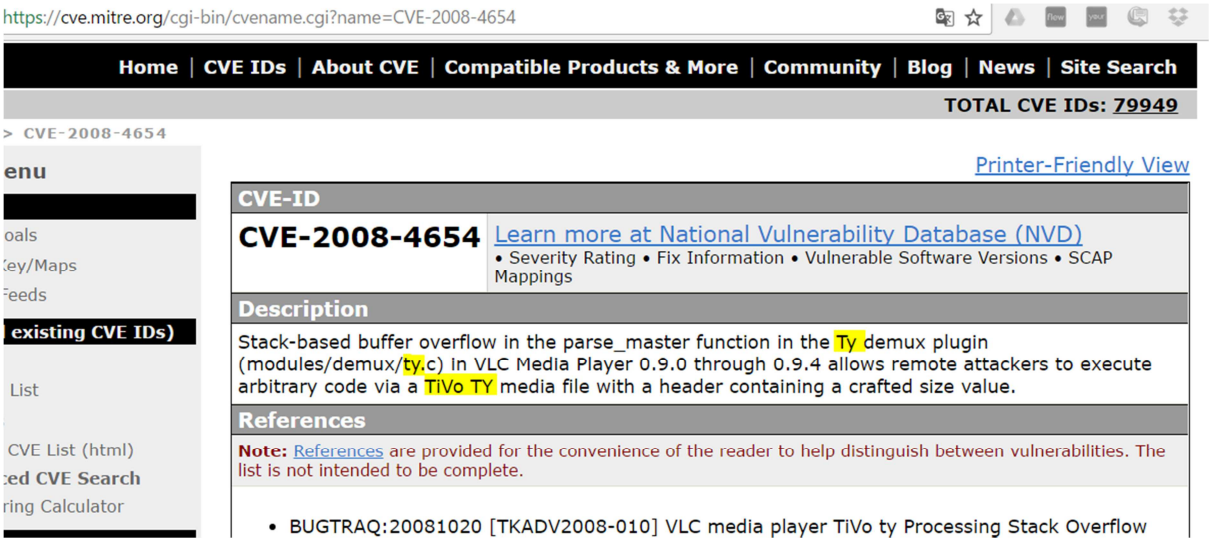
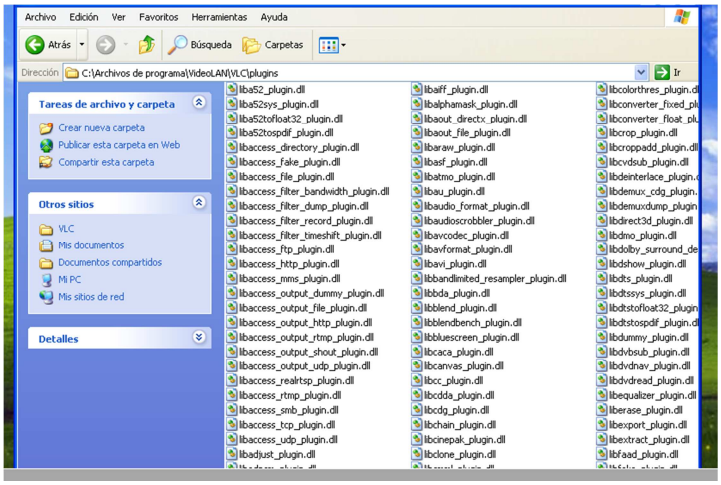
PART 30

We will begin to try to solve the exercise proposed in part 29. This is a Diff of two consecutive versions of VLC and you have the CVE as a help with the information contained therein and that we almost always really have.

After installing the vulnerable and patched versions in a virtual machine, I will look at the CVE information to see if it gives me a clue to help me not to diff too much.

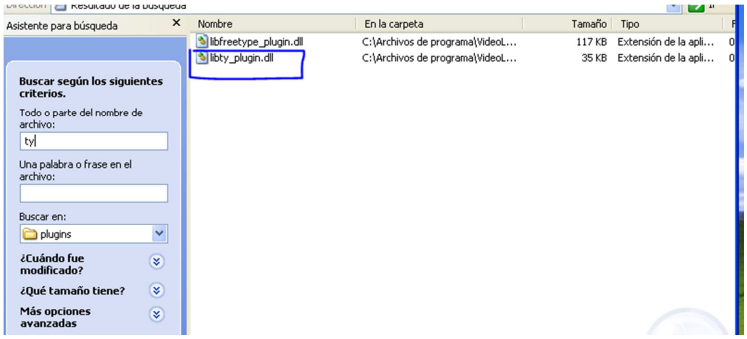


If I look at the folder where the VLC is installed, I see that it is organized to handle different extensions with a folder called plugins.

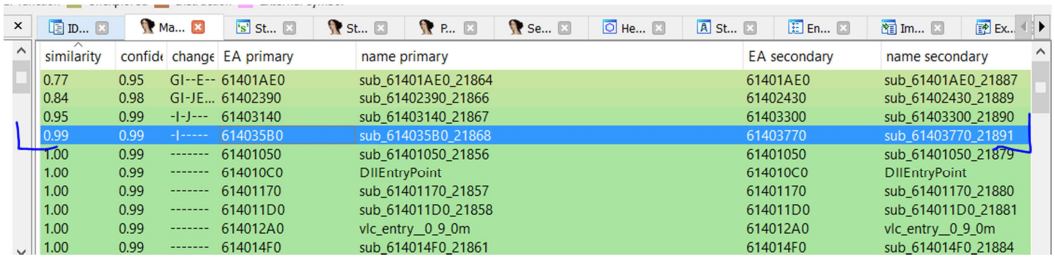


Let's see if by the name there is some that indicates that it works with the TIVO or TY format

.



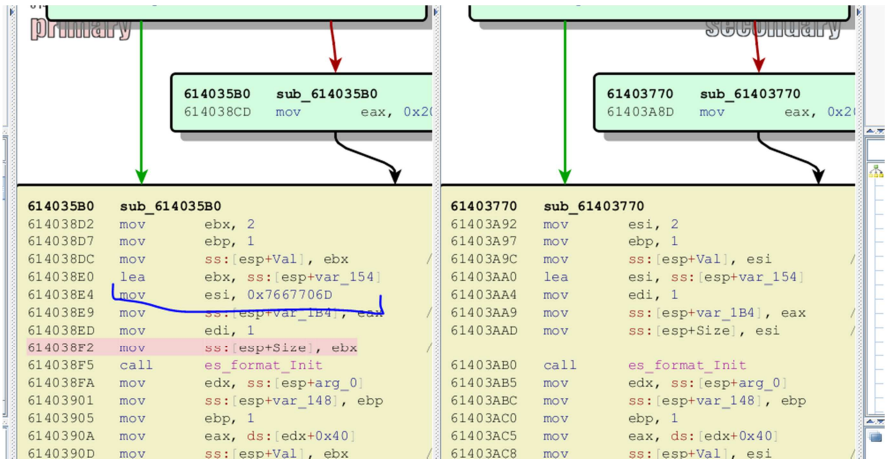
There is a libty_plugin.dll which seems to be quite suspicious. Let's do the diff on it.



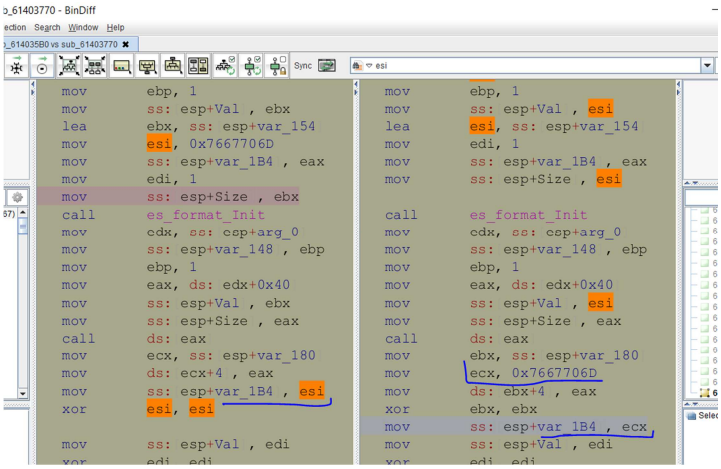
similarity	confidc	change	EA primary	name primary	EA secondary	name secondary
0.77	0.95	GI--E--	61401AE0	sub_61401AE0_21864	61401AE0	sub_61401AE0_21887
0.84	0.98	GI-JE...	61402390	sub_61402390_21866	61402430	sub_61402430_21889
0.95	0.99	-I-J---	61403140	sub_61403140_21867	61403300	sub_61403300_21890
0.99	0.99	-I-----	614035B0	sub_614035B0_21868	61403770	sub_61403770_21891
1.00	0.99	-------	61401050	sub_61401050_21856	61401050	sub_61401050_21879
1.00	0.99	-------	614010C0	DllEntryPoint	614010C0	DllEntryPoint
1.00	0.99	-------	61401170	sub_61401170_21857	61401170	sub_61401170_21880
1.00	0.99	-------	614011D0	sub_614011D0_21858	614011D0	sub_614011D0_21881
1.00	0.99	-------	614012A0	vlc_entry__0_9_0m	614012A0	vlc_entry__0_9_0m
1.00	0.99	-------	614014F0	sub_614014F0_21861	614014F0	sub_614014F0_21884

We see that there are four functions changed. What we usually do is to have a bird's eye view of it first, marking the most suspicious for later if we start to reverse those only more deeply.

We look for a patch that prevents a stack overflow. We see, for example, this function changed.

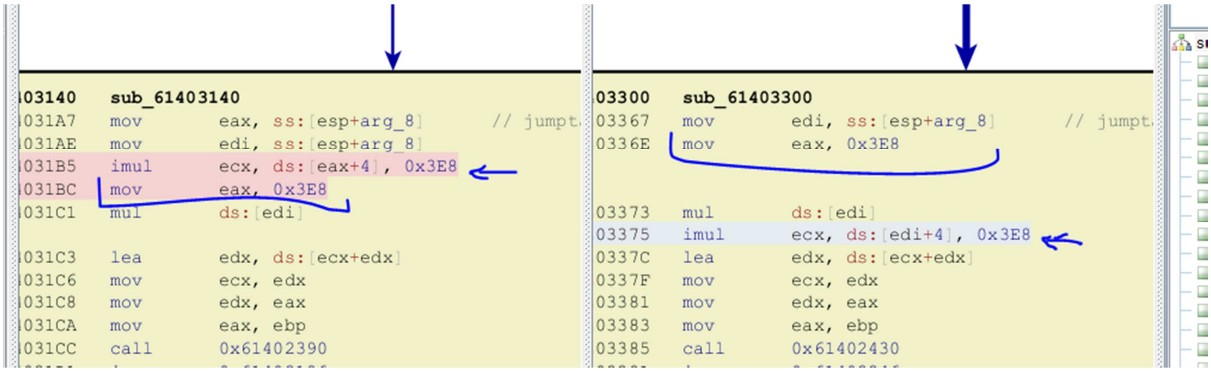


That does not seem to prevent anything. Only one address to ESI. There can be no problem with that.

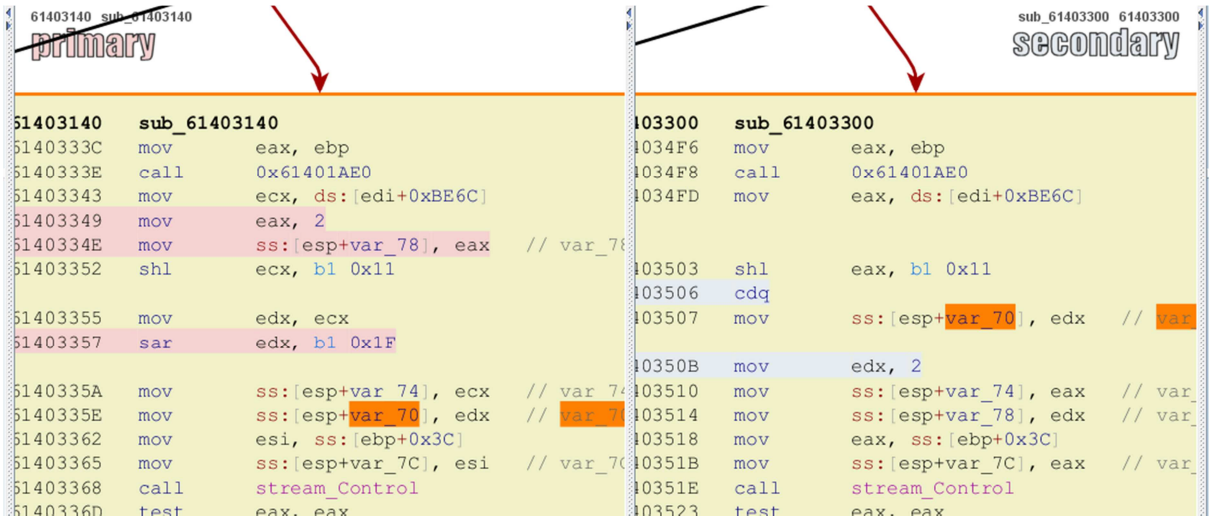


We see that it is only a change in the order that does not affect. In the vulnerable, it moved that address to ESI which it stored in var_1b4 and in the patched, it moves the address to ECX and saves it in var_1b4, nothing around here.

In the next, the same. Many changes but sometimes, It uses another record for the same effect, but it is the same, changes of order, this down does not affect anything.



We see some changes in the way var_70 calculates, but I do not see it reads or uses it anywhere in the function and it is a local variable, neither is it passed as an argument nor compared. We will take it into account very minimally. For now, it is not a priority.

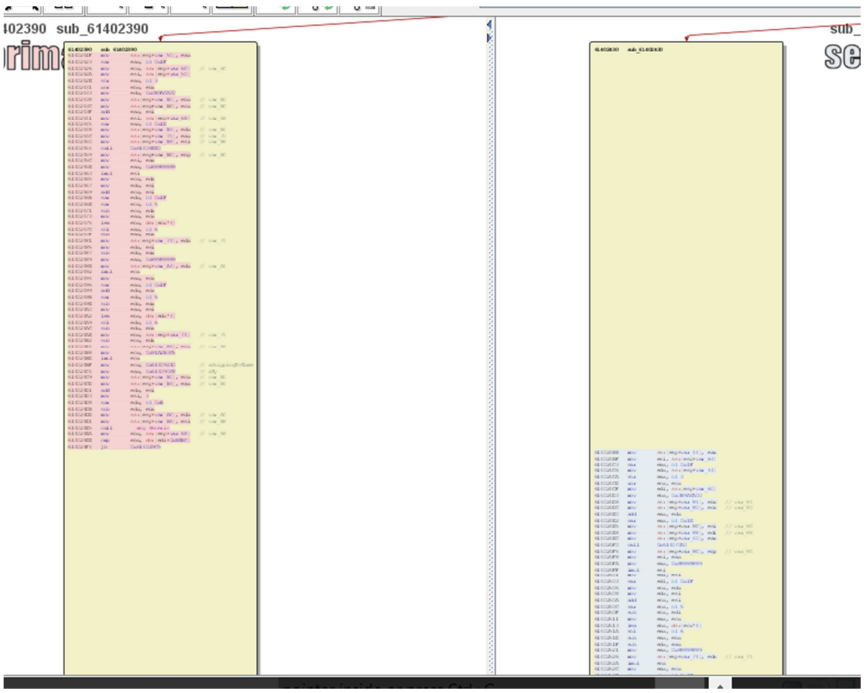


That function has nothing more. This is not seen as a candidate. Let's continue with the bird's view.

The next one is very messy. We will see if we can accommodate it a little.

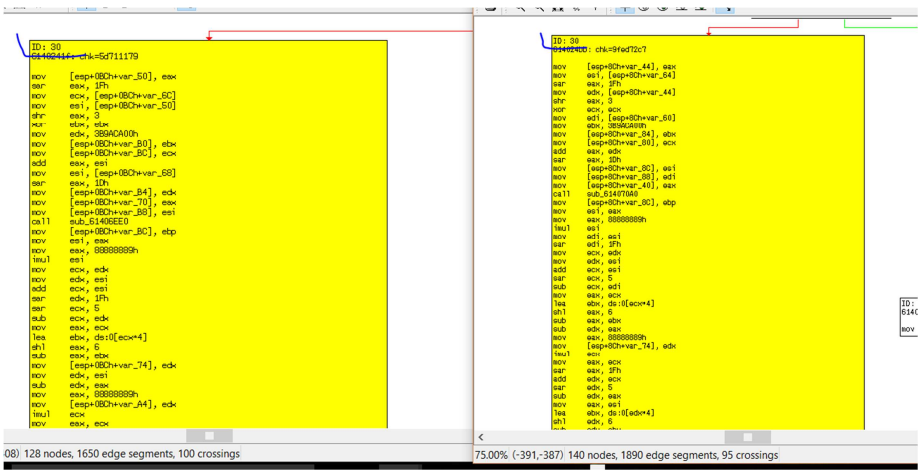
Right-clicking - **delete matches** in the unmatched blocks, then marking the ones that should match and selecting **add basic block match**.

There are blocks that look very different when being matched badly but when we match them well…

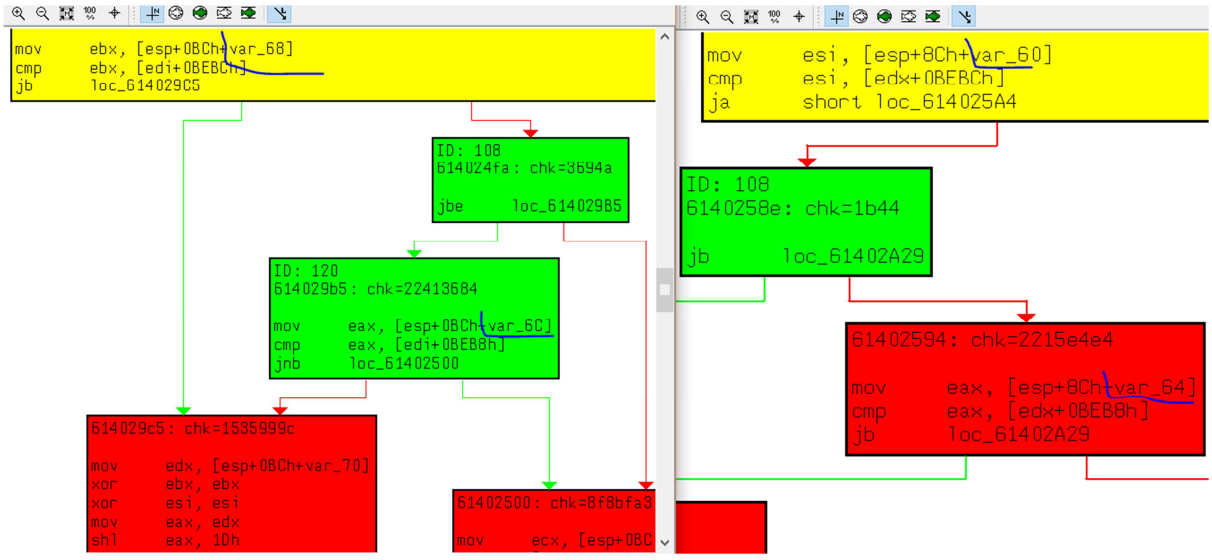


If I look at them, they look the same now, but they look a little badly drawn.

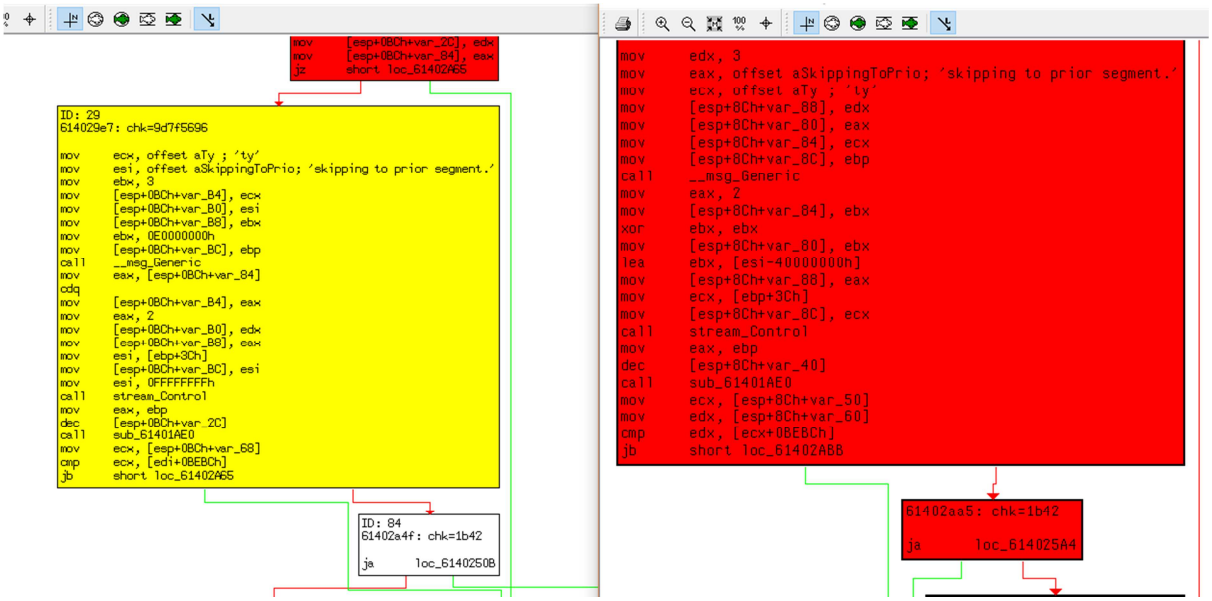
Sometimes, there are functions that are very unarmed. It is advisable to use Turbodiff too, so that they do not disarm so much there.



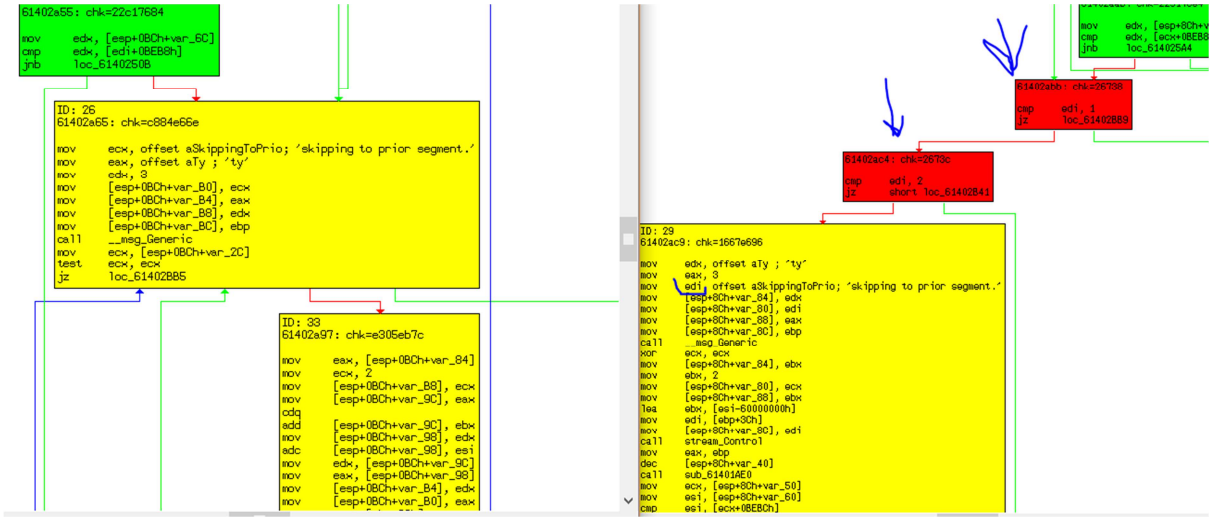
Let's look at the block IDs.



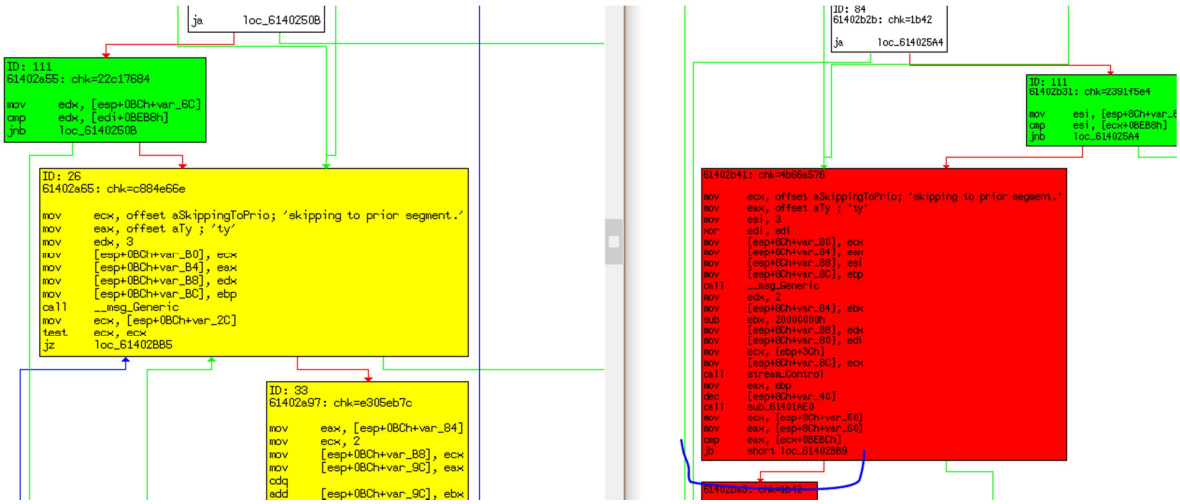
We see that the variables are spread but the comparisons are similar (68-6c and in the new 60-64) There is also no change of sign in the comparison (JB for JL or something, it keeps equal) If the destinations are inverted in the comparison inversion of JA by JB, it is the same. It does not change anything. Let's keep looking.



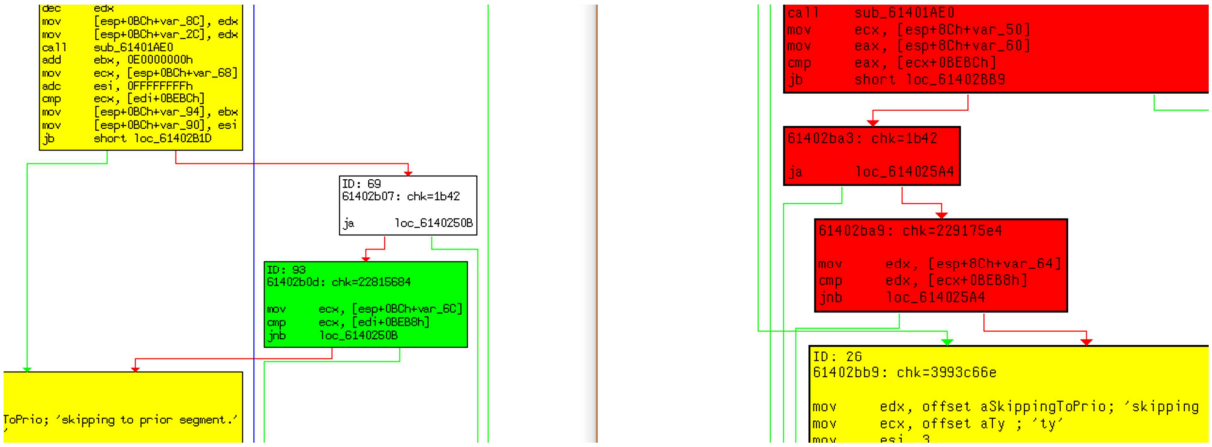
At bird's eye view here, it looks pretty similar. There is no change of sign in the comparison. Minor changes.



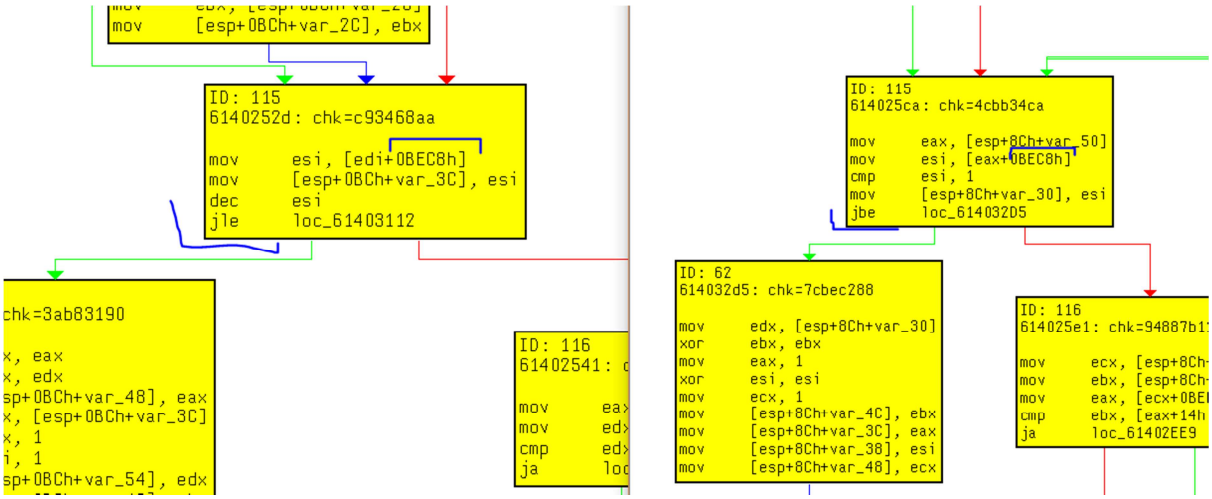
There, we see a couple of filters that the old one does not have. As EDI is overwritten just later, so they can be aggregated cases. We leave them marked but it is not seen as suspicious of overflow.



We see the JB there. In the old, it is a little lower. There are changes here to study later if we find nothing else.



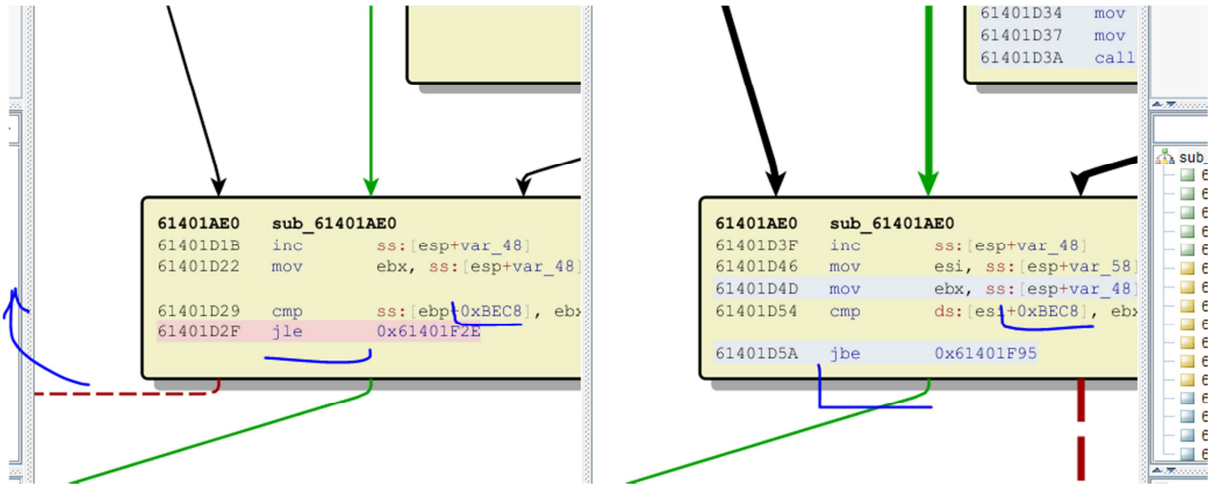
Recall that we are not reversing in depth just looking for something that catches our attention as very likely.



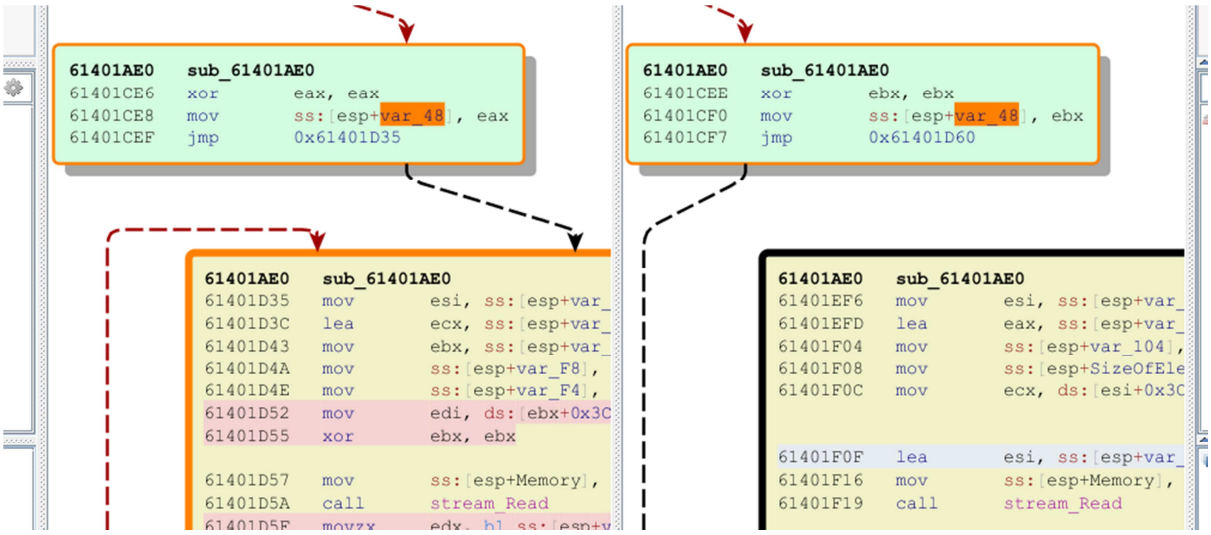
Here, if you see something very possible that affects, a field of the structure that compares, and in one makes a decision with JLE and in the other with JBE that is a change of sign and we point it as possible.

It is a very complex function. We will analyze it later. We saw something very possible. We note it and look at the last one a little.

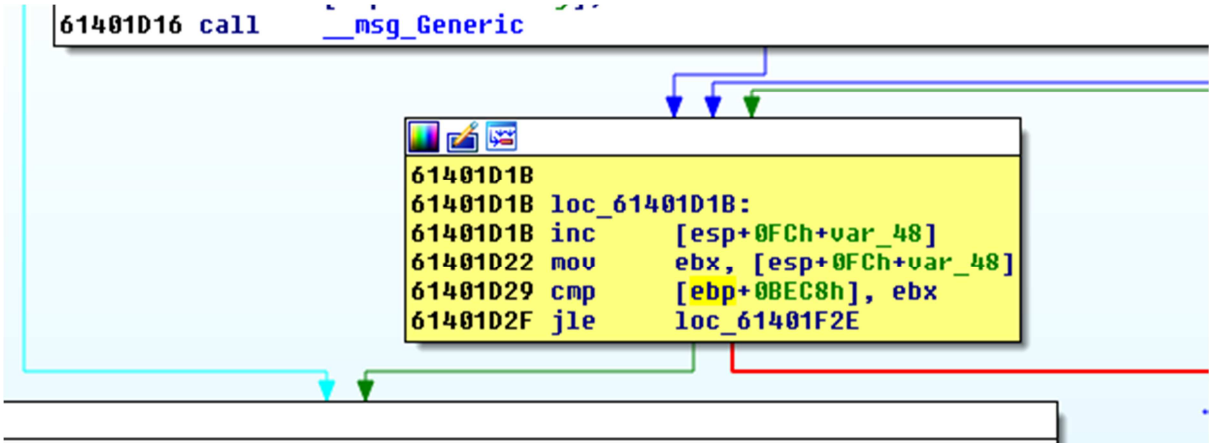
In this function, the same field of structure affects and it is easier to see.



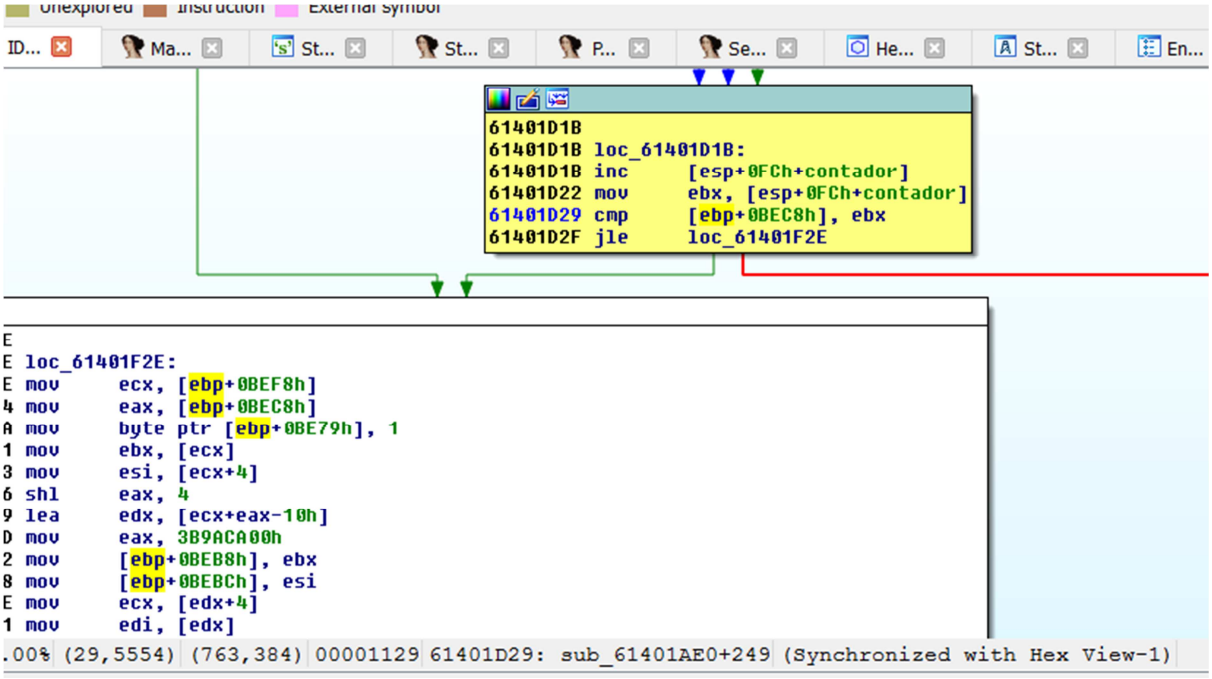
It is a loop and the value that decides the output is a counter that is in var_48 and is incremented and compared to the maximum.



Before entering the LOOP, the counter is reset to zero. I think this is easier to start reversing than the previous, although both may be the culprit, we always start with the easiest, hehe, the latter.

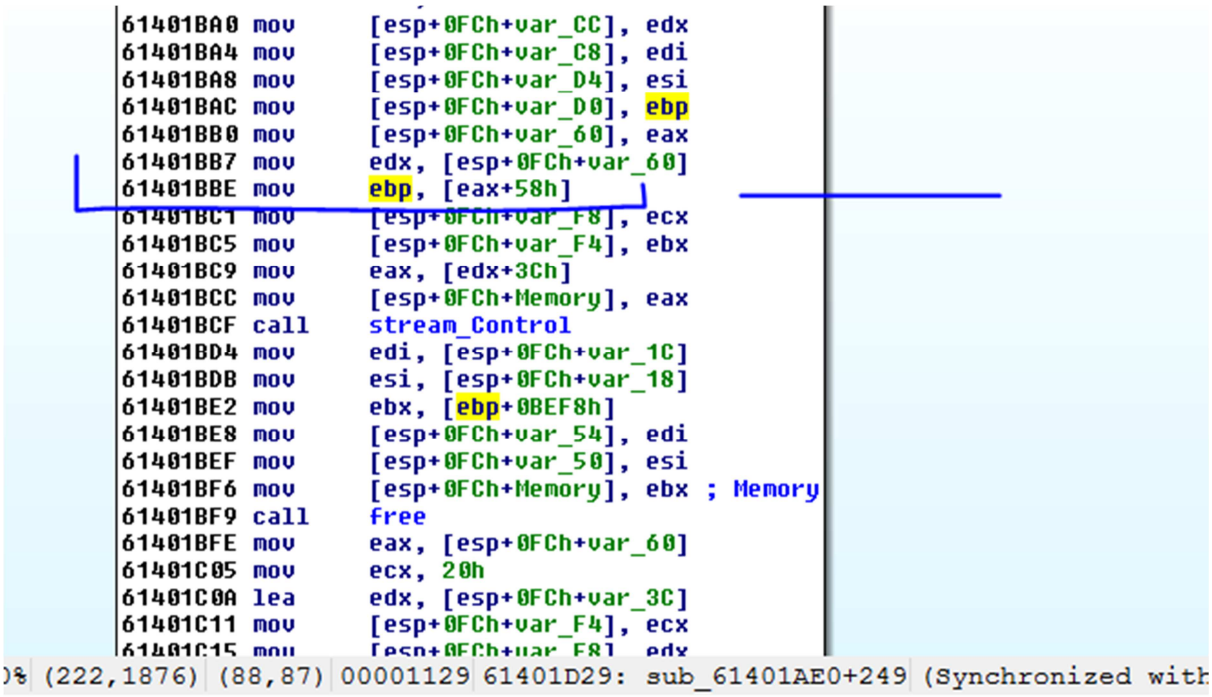


Let's start with patience because it looks complex, we rename var_48 as **contador** that means **counter** in English.

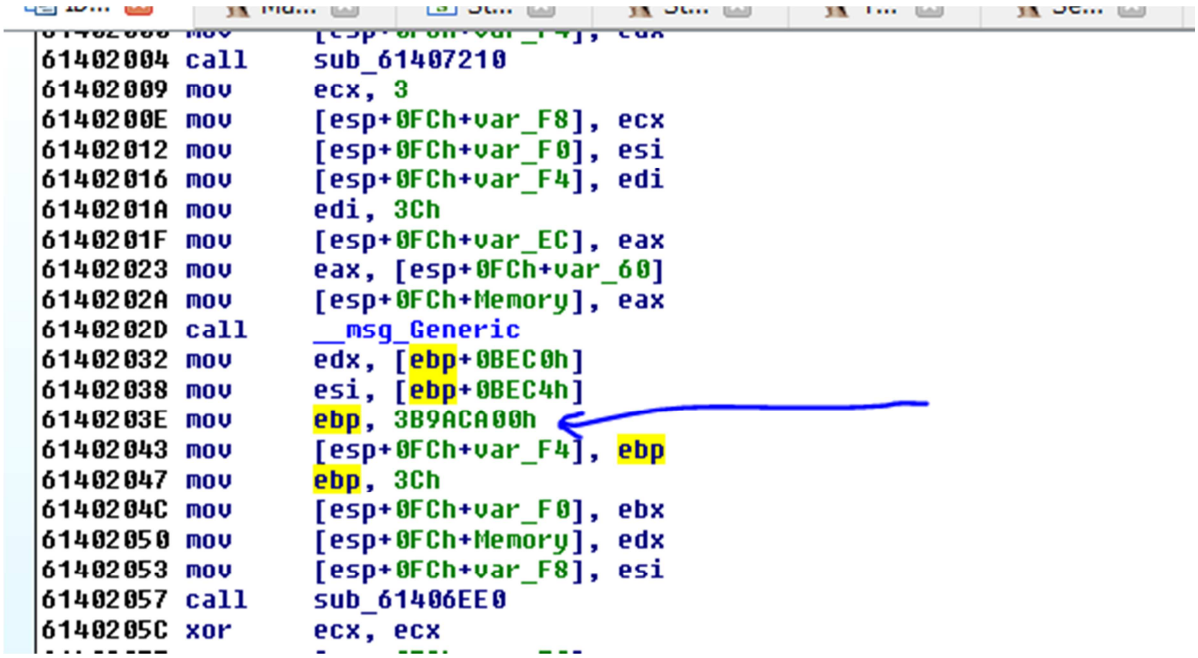


Obviously, EBP is the address of the structure, if we see in most of the function it stays the same, starting from EBP + XXXX the fields of it.

EBP takes the value from here.



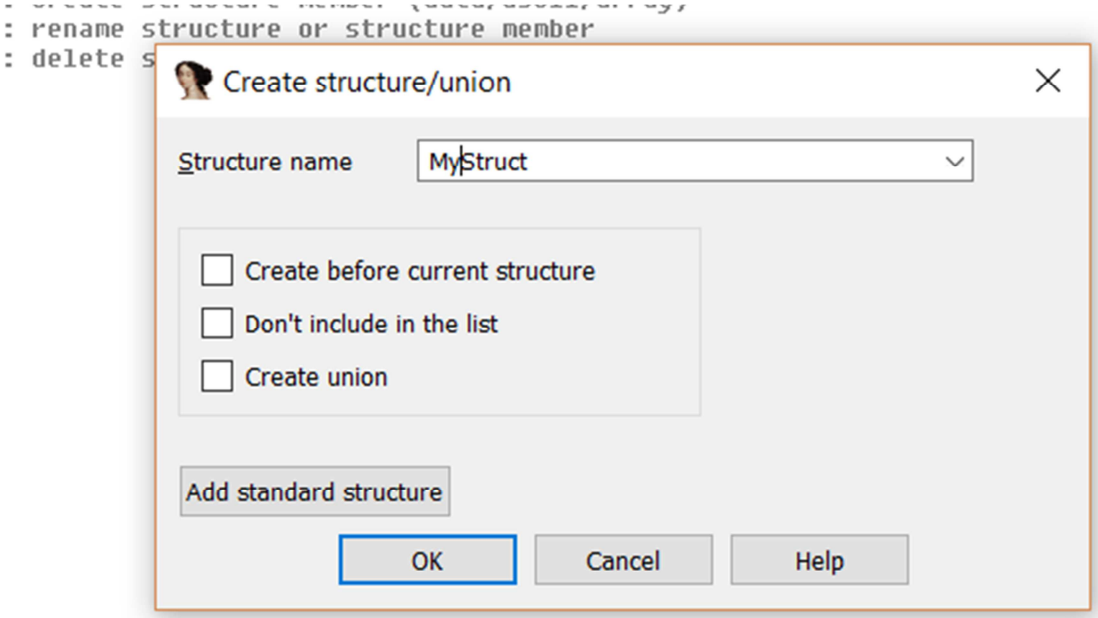
From there, EBP is the address of the structure and changes here.

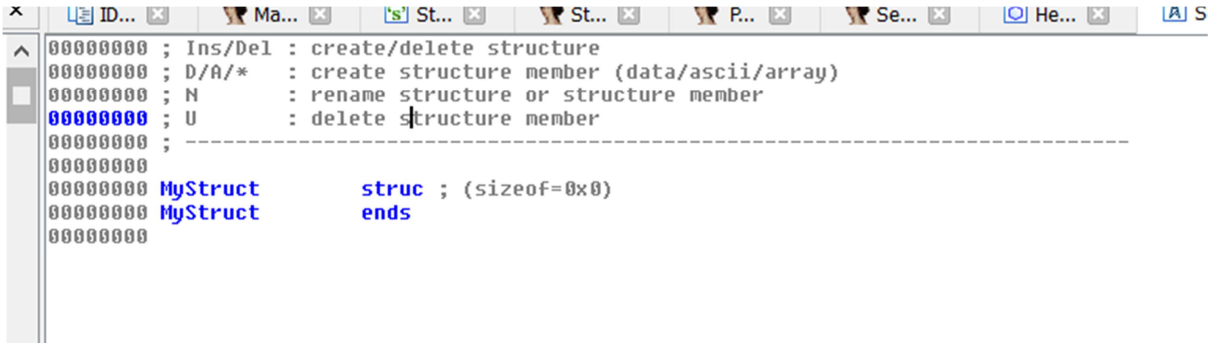


There, EBP changes its value, meaning that between both directions, EBP is the address of the structure.

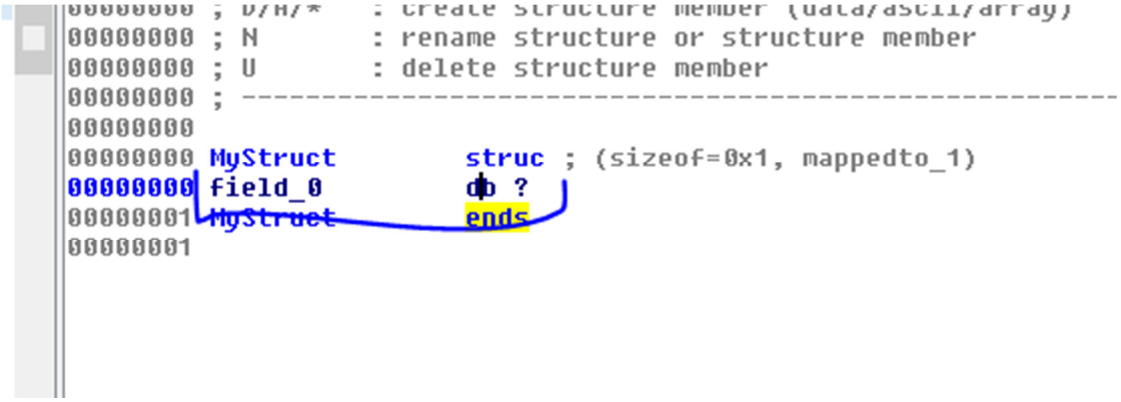
We see that it is a very large structure. There are 0xbexx fields which are a big structure, let's do it, I think most of the fields are 0xbexx so we can make a long structure 0xbf00 that covers those we see to enlarge or shrink. There is time.

I go to the structures tab and press INSERT to add one.



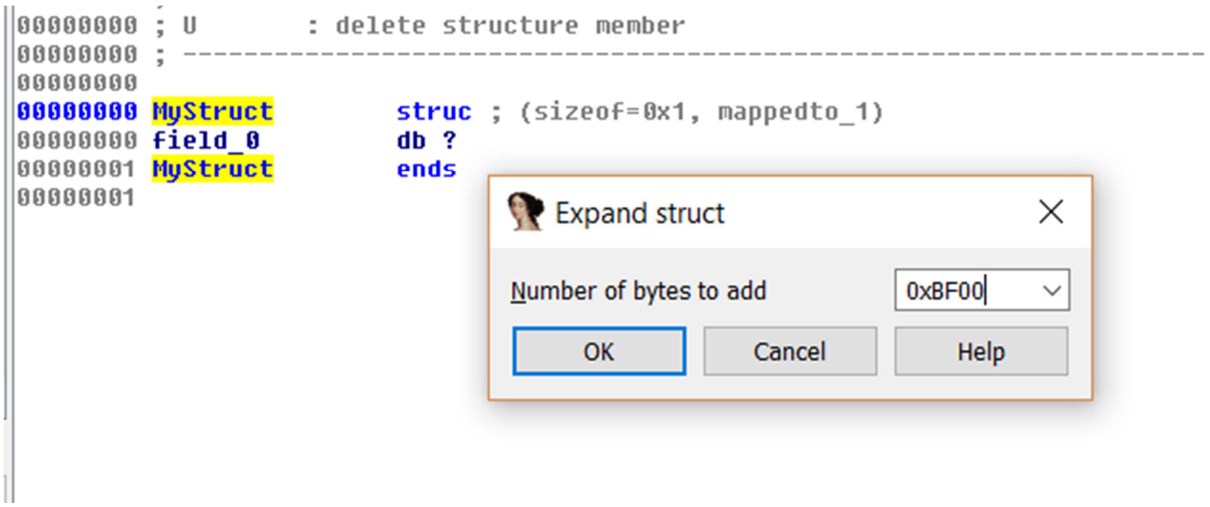


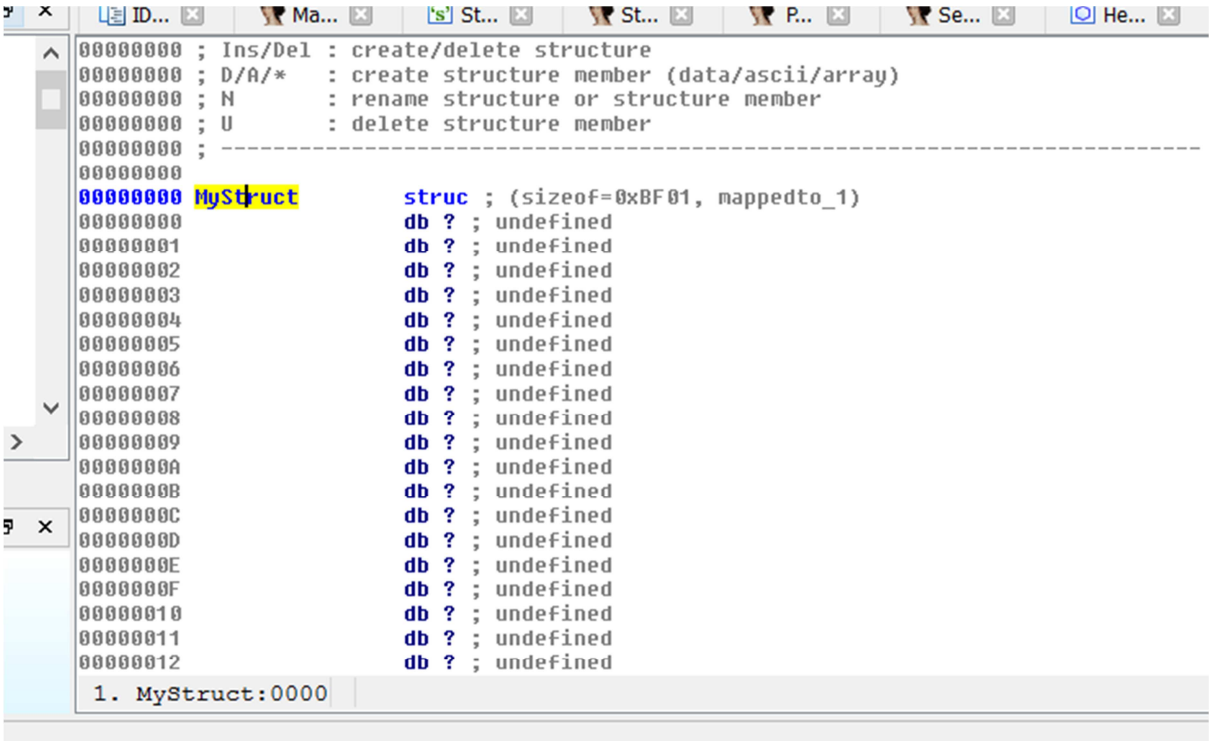
On **ends**, I press D to add a one-byte field.



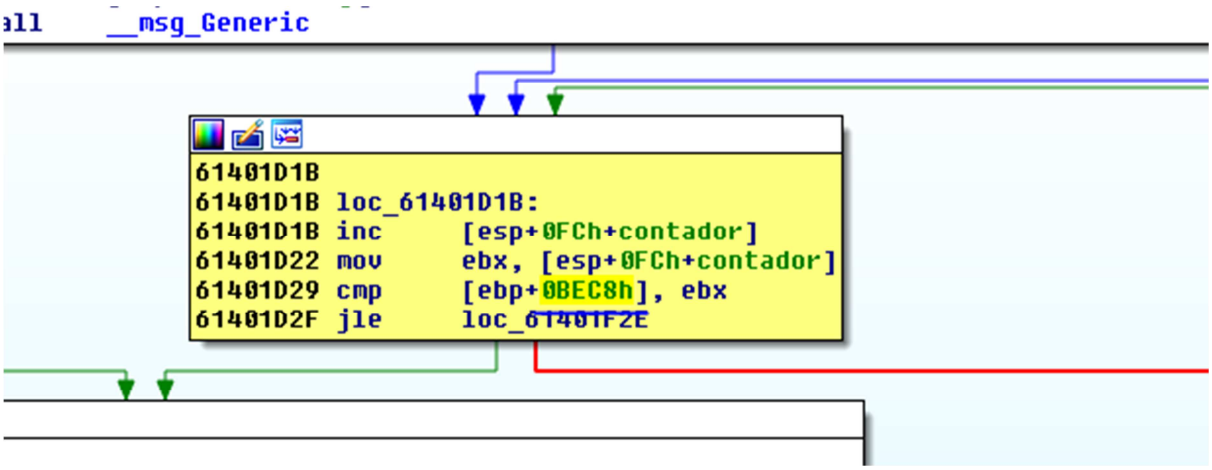
I right click EXPAND STRUCT TYPE.

I add 0xBF00 for one more byte, nobody dies, hehe.


00000000 ; U : delete structure member
00000000 ; ----------
00000000
00000000 MyStruct struc ; (sizeof=0x1, mappedto_1)
00000000 field_0 db ?
00000001 MyStruct ends
00000001
Expand struct
Number of bytes to add
0xBF00
OK
Cancel
Help


ID... Ma... St... St... P... Se... He...
00000000 ; Ins/Del : create/delete structure
00000000 ; D/A/* : create structure member (data/ascii/array)
00000000 ; N : rename structure or structure member
00000000 ; U : delete structure member
00000000 ; ----------
00000000
00000000 MyStruct struc ; (sizeof=0xBF01, mappedto_1)
00000000 db ? ; undefined
00000001 db ? ; undefined
00000002 db ? ; undefined
00000003 db ? ; undefined
00000004 db ? ; undefined
00000005 db ? ; undefined
00000006 db ? ; undefined
00000007 db ? ; undefined
00000008 db ? ; undefined
00000009 db ? ; undefined
0000000A db ? ; undefined
0000000B db ? ; undefined
0000000C db ? ; undefined
0000000D db ? ; undefined
0000000E db ? ; undefined
0000000F db ? ; undefined
00000010 db ? ; undefined
00000011 db ? ; undefined
00000012 db ? ; undefined
1. MyStruct:0000

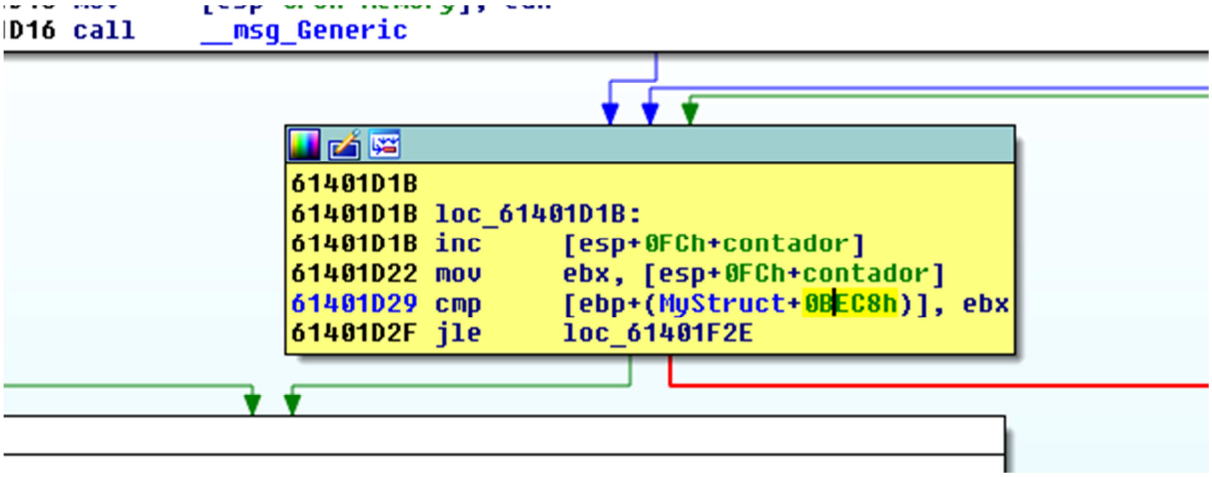
There, it is.


all __msg_Generic
61401D1B
61401D1B loc_61401D1B:
61401D1B inc [esp+0FCh+contador]
61401D22 mov ebx, [esp+0FCh+contador]
61401D29 cmp [ebp+0BEC8h], ebx
61401D2F jle loc_61401F2E

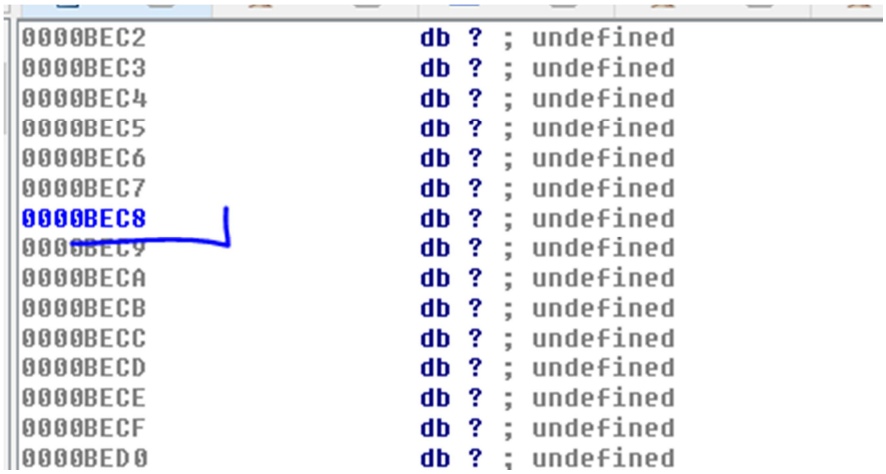
Obviously, if it were possible that the 0x0bec8 field is negative, for example: 0xFFFFFFFF, it will be less than the positive values (1, 2, etc.) that the counter takes since the sign is considered and the loop will repeat itself much more than thought.

I can rename the field as MAXIMO since it is assumed to be the maximum value that the loop should repeat before exiting.

Press T.



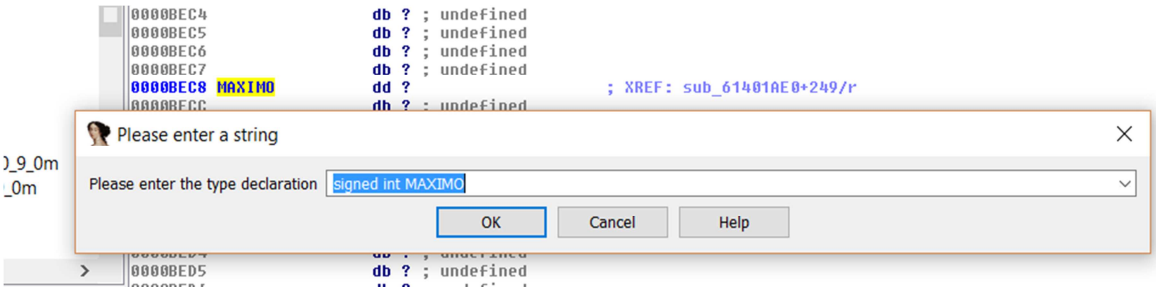
And I should go to the structure at 0xbec8 and create a DWORD field as that is what it is.



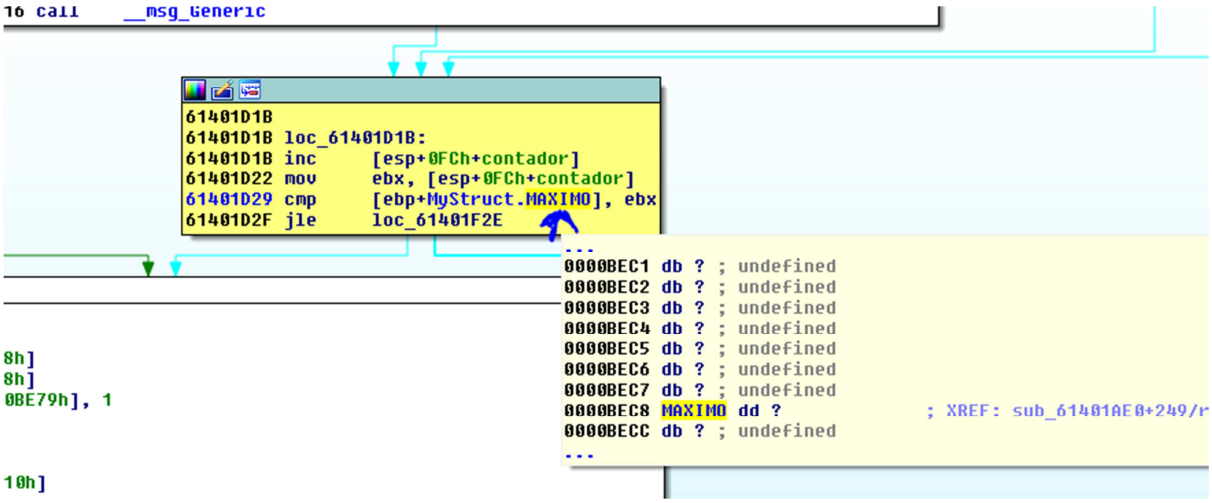
I press D there several times until it shows DD.

```
0000BEC2                db ? ; undefined
0000BEC3                db ? ; undefined
0000BEC4                db ? ; undefined
0000BEC5                db ? ; undefined
0000BEC6                db ? ; undefined
0000BEC7                db ? ; undefined
0000BEC8 field_BEC8     dd ?                    ; XREF: sub_61401AE0+249/r
0000BECC                db ? ; undefined
0000BECD                db ? ; undefined
0000BECE                db ? ; undefined
0000BECF                db ? ; undefined
0000BED0                db ? ; undefined
```

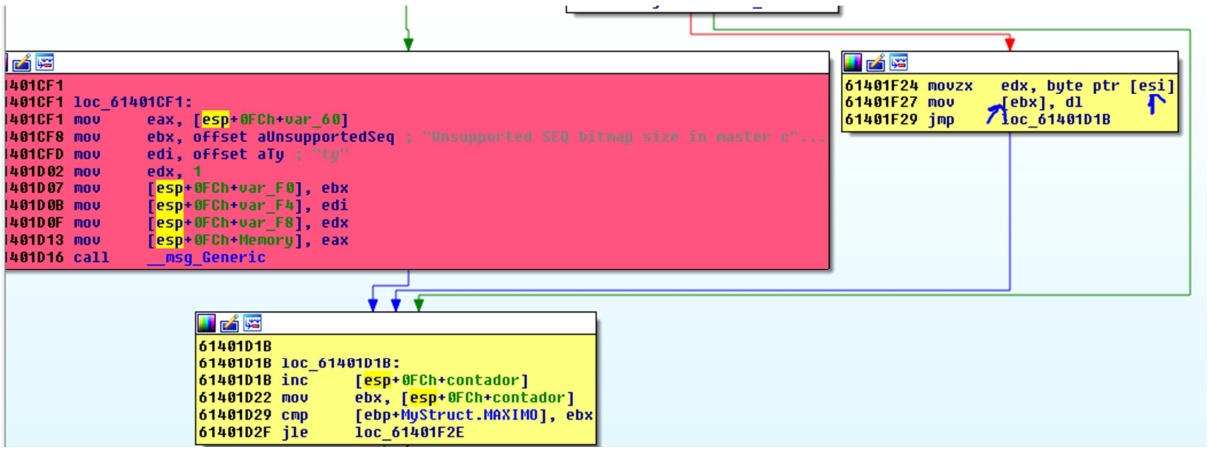
I rename it as MAXIMO.



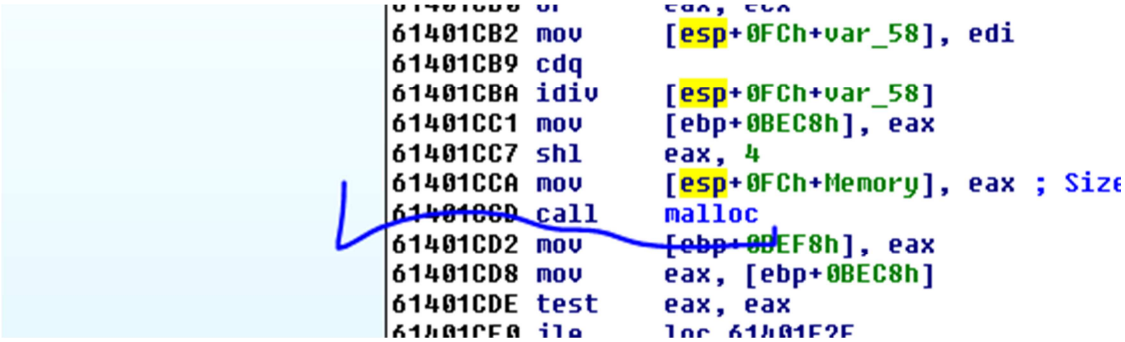
I can press Y and change the type since I know it's SIGNED INT, by the JLE that compares it.



It looks better. I put the **signed int**, but it will not affect much, except that I use the Hex Rays decompiler which I will not do for now, but I like to accommodate things well.



I'll continue studying.



We see that there is a **malloc**. This is the function used to reserve a buffer dynamically, not in the stack, but in memory.

malloc

Visual Studio 2015 | Otras versiones

Para obtener la documentación más reciente de Visual Studio 2017 RC, consulte Documentación de Visual

Asigna bloques de memoria.

Sintaxis

```
void *malloc(
   size_t size
);
```

Parámetros

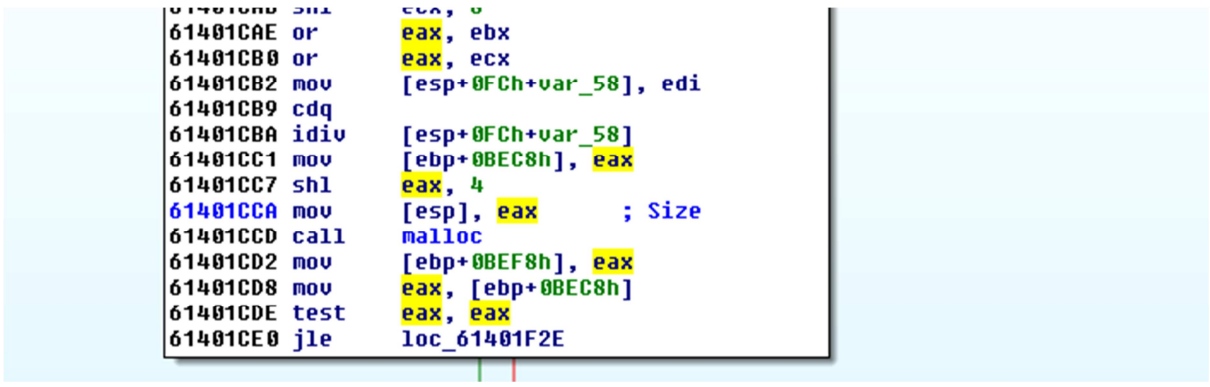
`size`

Bytes a asignar.

Valor devuelto

`malloc` devuelve un puntero void al espacio asignado, o `NULL` si no hay disponible memoria suficiente. P

You pass a single argument that is the size to reserve or size. Here, the program uses the method that we saw to save in the stack instead of pushing. We know that if we click right, we can fix the instruction.



There we see that the **size** argument is in EAX and comes from making several accounts.

```
61401C83 movzx   eax, [esp+0FCh+var_20]
61401C8B movzx   esi, [esp+0FCh+var_1F]
61401C93 movzx   ebx, [esp+0FCh+var_1D]
61401C9B movzx   ecx, [esp+0FCh+var_1E]
61401CA3 shl     eax, 18h
61401CA6 shl     esi, 10h
61401CA9 or      eax, esi
61401CAB shl     ecx, 8
61401CAE or      eax, ebx
61401CB0 or      eax, ecx
61401CB2 mov     [esp+0FCh+var_58], edi
61401CB9 cdq
61401CBA idiv    [esp+0FCh+var_58]
61401CC1 mov     [ebp+0BEC8h], eax
61401CC7 shl     eax, 4
61401CCA mov     [esp], eax      ; Size
61401CCD call    malloc
```

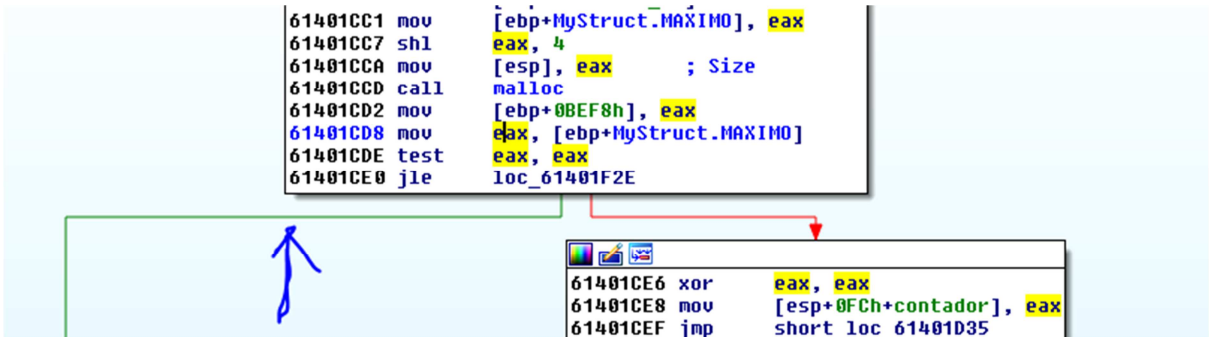
We see four variables of the byte type from which we perform the operations that create the size which makes SHL EAX, 4 at the end before passing it to malloc.

```
shl eax, 1   ;Equivalent to EAX*2
shl eax, 2   ;Equivalent to EAX*4
shl eax, 3   ;Equivalent to EAX*8
shl eax, 4   ;Equivalent to EAX*16
shl eax, 5   ;Equivlaent to EAX*32
shl eax, 6   ;Equivalent to EAX*64
shl eax, 7   ;Equivalent to EAX*128
shl eax, 8   ;Equivalent to EAX*256
```

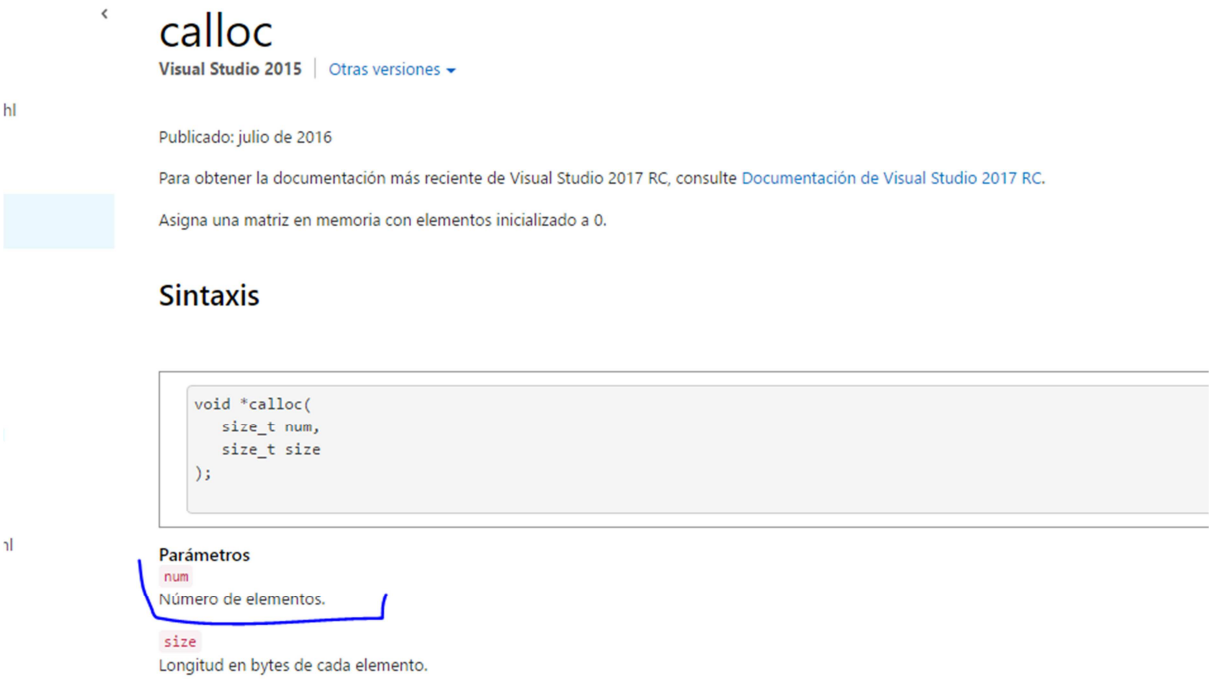
The SHL EAX, 4 byte shift is equivalent to EAX * 16, but before multiplying, it saves it in the MAXIMO variable. If I press T, I see that it is the same.

```
61401C80 add     edi, 8
61401C83 movzx   eax, [esp+0FCh+b1]
61401C8B movzx   esi, [esp+0FCh+b2]
61401C93 movzx   ebx, [esp+0FCh+b3]
61401C9B movzx   ecx, [esp+0FCh+b4]
61401CA3 shl     eax, 18h
61401CA6 shl     esi, 10h
61401CA9 or      eax, esi
61401CAB shl     ecx, 8
61401CAE or      eax, ebx
61401CB0 or      eax, ecx
61401CB2 mov     [esp+0FCh+var_58], edi
61401CB9 cdq
61401CBA idiv    [esp+0FCh+var_58]
61401CC1 mov     [ebp+MyStruct.MAXIMO], eax
61401CC7 shl     eax, 4
61401CCA mov     [esp], eax      ; Size
61401CCD call    malloc
```

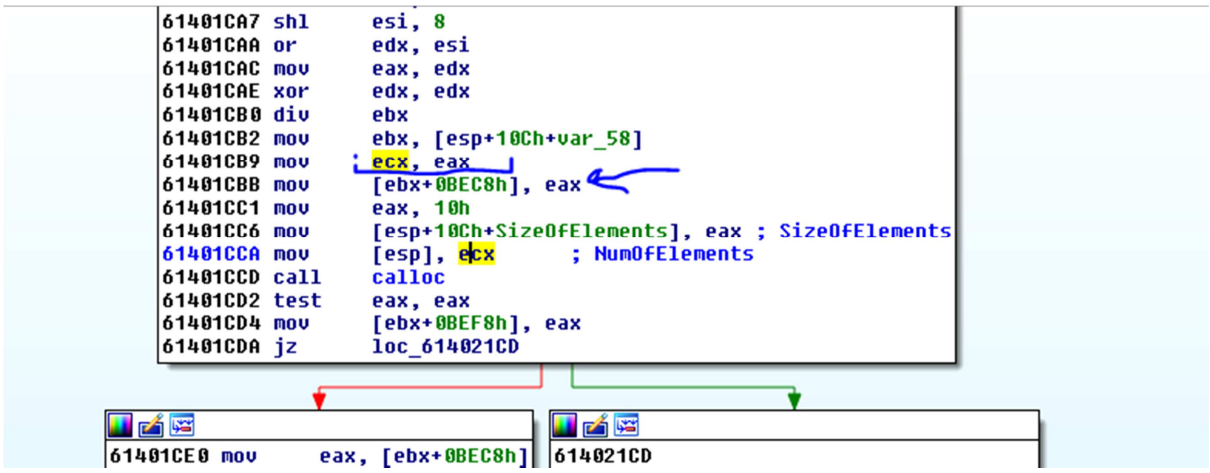
We see that the maximum negative values are filtered here.



So the subject is not a negative value of MAXIMO because it is filtered. In the patched function, we see that it does not do the SHL or it does not multiply by 16 directly the value of the account. It uses it as calloc size.



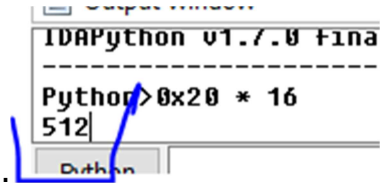
We see that instead of multiplying by 16 what it does is to pass it to calloc that has one argument and the size of each element that is 0x10, whereby multiplication is done by the size API by the size of each element.



MALLOC or CALLOC is used to reserve memory. The addresses it returns are variables. It will not always give us an area with the same memory address. Later, we will study the heap or how to reserve memory, but for now, it will give us a zone of memory to work with the size we ask it.

In the vulnerable, it multiplies MAXIMUM by 16 before calculating the size and then passes it to malloc. In the patched, it does not and it passes the MAXIMO to calloc directly and the multiplication is internally calculated by 16 which is the size of each element.

The problem is that if MAX is, for example, 0x20 bytes and it multiplies it by 16, it will be equal to:

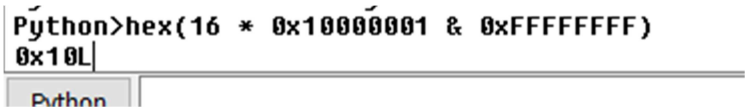


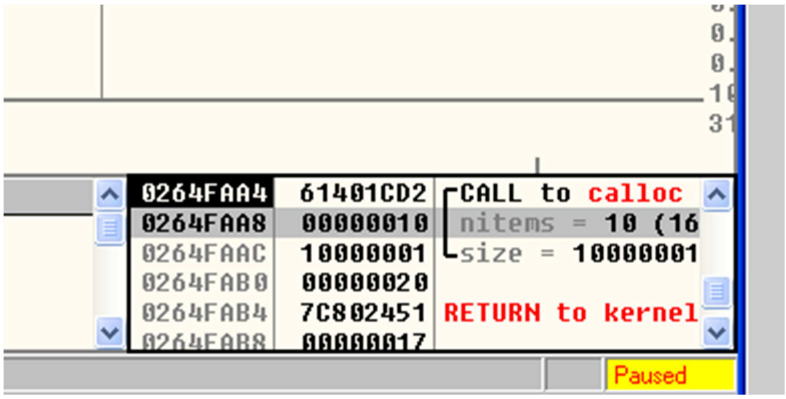
This will reserve 512 bytes and then copy 0x20 because it compares inside the loop and exits when the counter is greater than MAXIMO.

Now, what happens if the MAXIMO value is positive but multiplying it by 16 is smaller than the initial value?

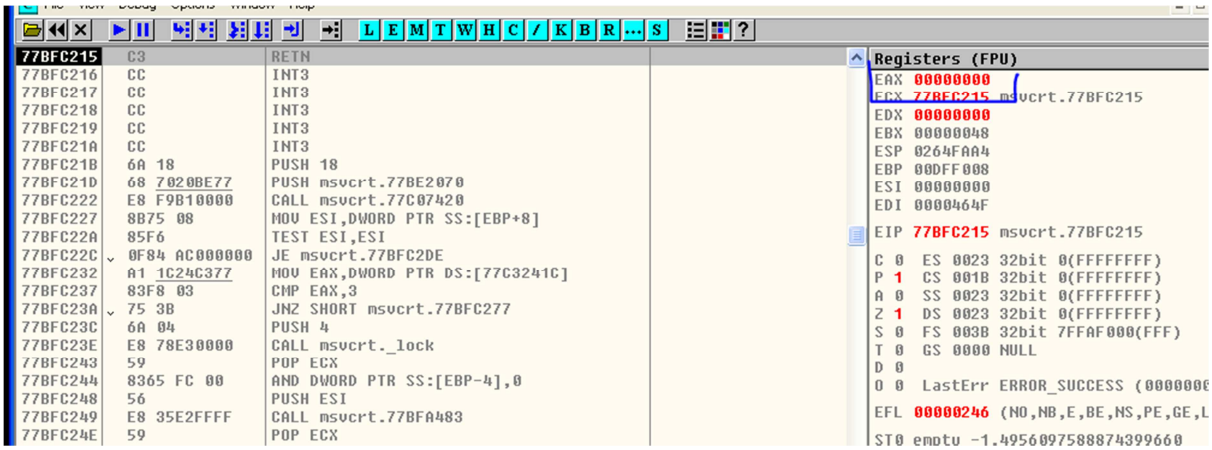
If MAXIMO is 0x10000001 when multiplying by 16 is 0x10 with which only 0x10 bytes will be reserved and when it is copied in the loop, it will write 1 at a time until the counter reaches 0x10000001 which overflows the buffer

copying more than the reserved or of the buffer size which is the definition of overflow. Although, in this case, it is not a stack overflow but a heap overflow. While in the parched, calloc does not allow the internal multiplication to be turned around and be the result smaller than the MAXIMO, for which a possible vulnerability is repaired.





There, I will prove that passing the same values to malloc, it returns zero or does not alloc and return any reserved memory address while doing it with malloc, it allocated 0x10 which it realy worked and wrote 0x10000001 causing BUFFER OVERFLOW.



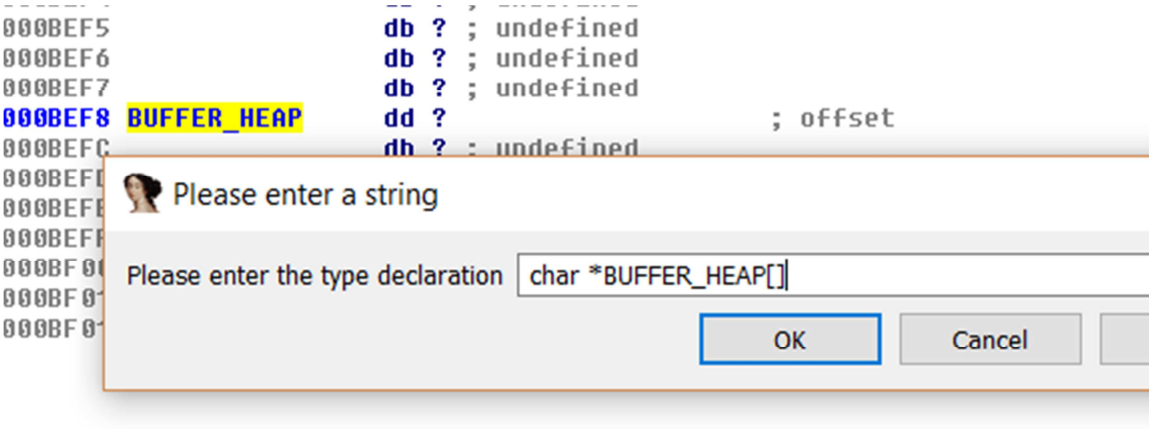
Let's continue analyzing to see what else there is.

```
61401CAB shl     ecx, 8
61401CAE or      eax, ebx
61401CB0 or      eax, ecx
61401CB2 mov     [esp+0FCh+var_58], edi
61401CB9 cdq
61401CBA idiv    [esp+0FCh+var_58]
61401CC1 mov     [ebp+MyStruct.MAXIMO], eax
61401CC7 shl     eax, 4
61401CCA mov     [esp], eax      ; Size
61401CCD call    malloc
61401CD2 mov     [ebp+0BEF8h], eax
61401CD8 mov     eax, [ebp+MyStruct.MAXIMO]
61401CDE test    eax, eax
61401CE0 jle     loc_61401F2E
```

There it stores the address of the reserved buffer in the heap, I can rename it, for that I will go to structures and press D until it is a DWORD.

```
0000BEF2                db ? ; undefined
0000BEF3                db ? ; undefined
0000BEF4                db ? ; undefined
0000BEF5                db ? ; undefined
0000BEF6                db ? ; undefined
0000BEF7                db ? ; undefined
0000BEF8 field_BEF8     dd ?
0000BEFC                db ? ; undefined
0000BEFD                db ? ; undefined
0000BEFE                db ? ; undefined
0000BEFF                db ? ; undefined
0000BF00 field_0        db ?
0000BF01 MyStruct       ends
0000BF01
```

I rename it.



I do this with Y which is a pointer variable, which stores the address of the buffer I reserved in the heap, as I do not know what's there, I put it as an array of characters, i.e. a byte buffer but I can change it if I see Is another thing.

```
0000BEF5                db ? ; undefined
0000BEF6                db ? ; undefined
0000BEF7                db ? ; undefined
0000BEF8 BUFFER_HEAP    dd ?                    ; offset
0000BEFC                db ? ; undefined
0000BEFD                db ? ; undefined
0000BEFE                db ? ; undefined
0000BEFF                db ? ; undefined
0000BF00 field_0        db ?
0000BF01 MyStruct       ends
0000BF01
```

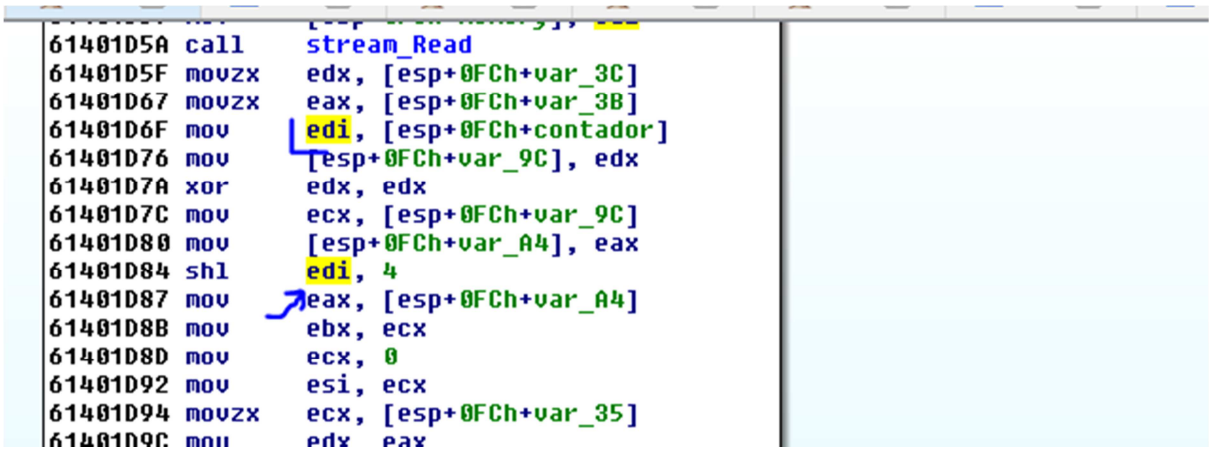
In the IDA language, it is an OFFSET or an address that points to something.

```
61401CB2 mov     [esp+0FCh+var_58], edi
61401CB9 cdq
61401CBA idiv    [esp+0FCh+var_58]
61401CC1 mov     [ebp+MyStruct.MAXIMO], eax
61401CC7 shl     eax, 4
61401CCA mov     [esp], eax      ; Size
61401CCD call    malloc
61401CD2 mov     [ebp+MyStruct.BUFFER_HEAP], eax
61401CD8 mov     eax, [ebp+MyStruct.MAXIMO]
61401CDE test    eax, eax
61401CE0 jle     loc_61401F2E
```

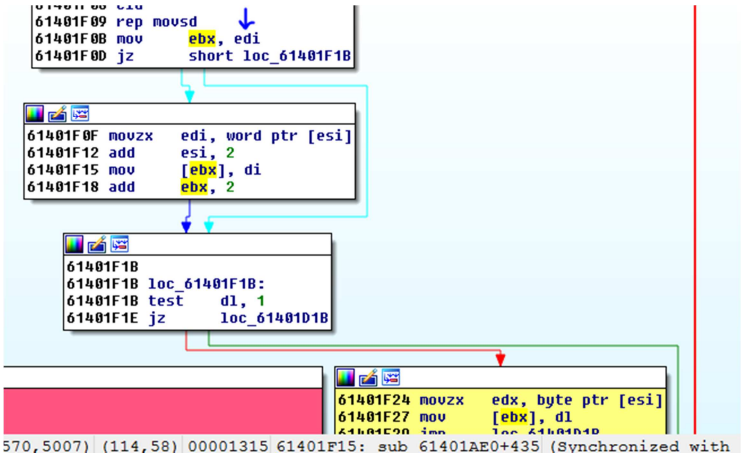
As we saw, malloc reserves the memory space of the size that I ask and returns the address of that buffer which I keep as any address in a variable of the type pointer.

```
61401E6F or      ebx, edx
61401E71 or      ecx, eax
61401E73 xor     edx, edx
61401E75 mov     [esp+0FCh+var_94], ecx
61401E79 movzx   esi, [esp+0FCh+var_36]
61401E81 mov     [esp+0FCh+var_90], ebx
61401E85 mov     ebx, [ebp+MyStruct.BUFFER_HEAP]
61401E8B mov     [esp+0FCh+var_D4], esi
61401E8F mov     eax, [esp+0FCh+var_D4]
61401E93 shld    edx, eax, 8
61401E97 shl     eax, 8
61401E9A or      ecx, eax
61401E9C mov     [ebx+edi], ecx
61401E9F mov     ecx, [esp+0FCh+var_90]
61401EA3 or      ecx, edx
61401EA5 cmp     [esp+0FCh+var_5C], 8
61401EAD mov     [ebx+edi+4], ecx
61401EB1 jg      loc_61401CF1
```

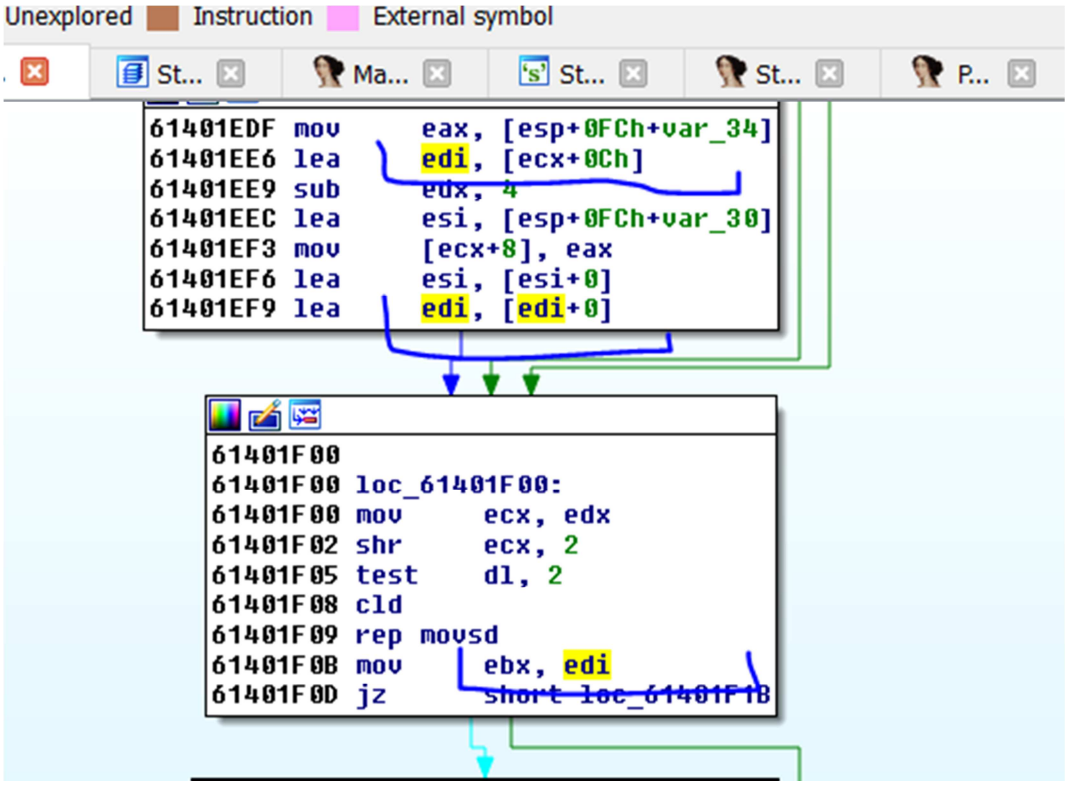
There, we see inside the loop that it takes the address and writes. It adds EDI that is the same counter by 16.



Inside the LOOP, it also writes in this other EBX that changed. That comes from that EDI. We would have to see where it is writing here.

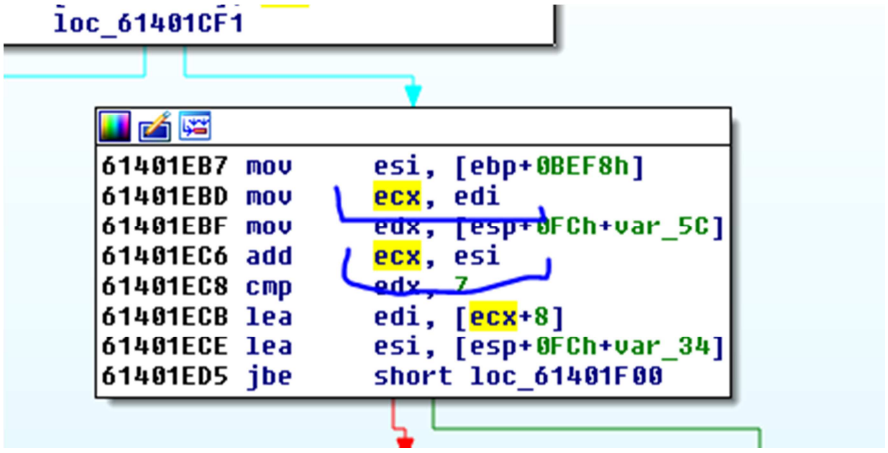


We see that destination address is coming from here.



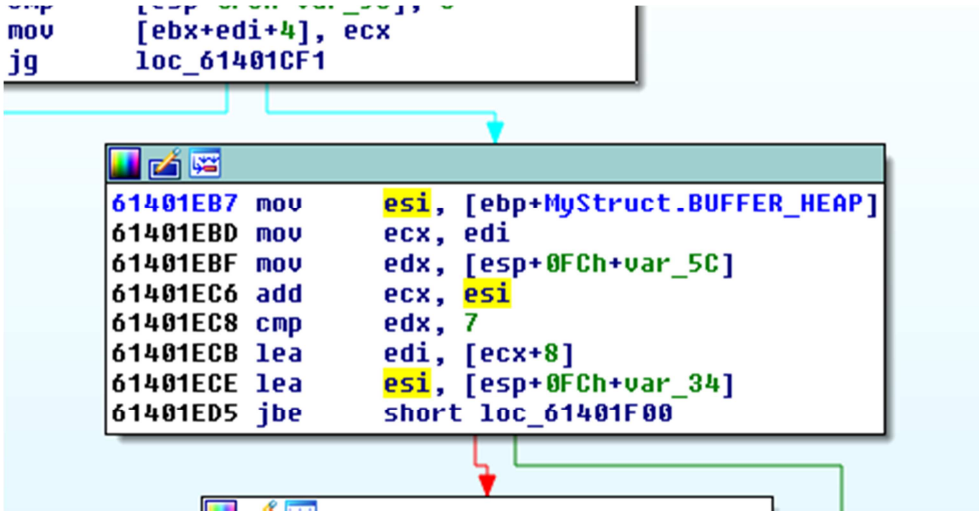
ECX + 0c is equal to EDI, so we'll look for where ECX comes from.

ECX leaves here.

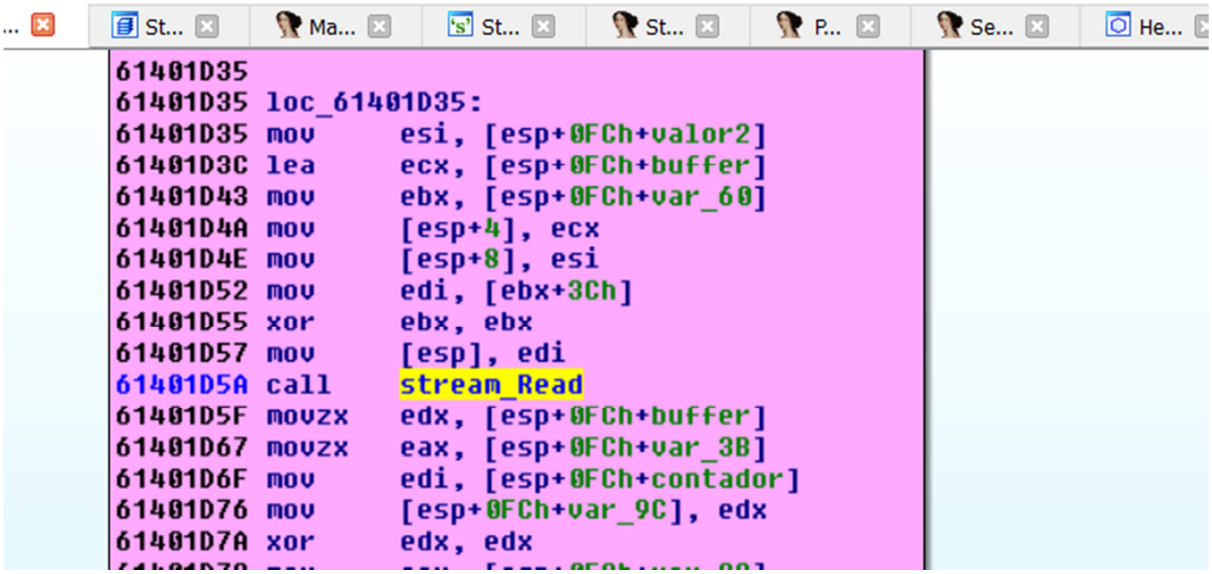


This moves the value of EDI to ECX and adds ESI to it.

If I press T.



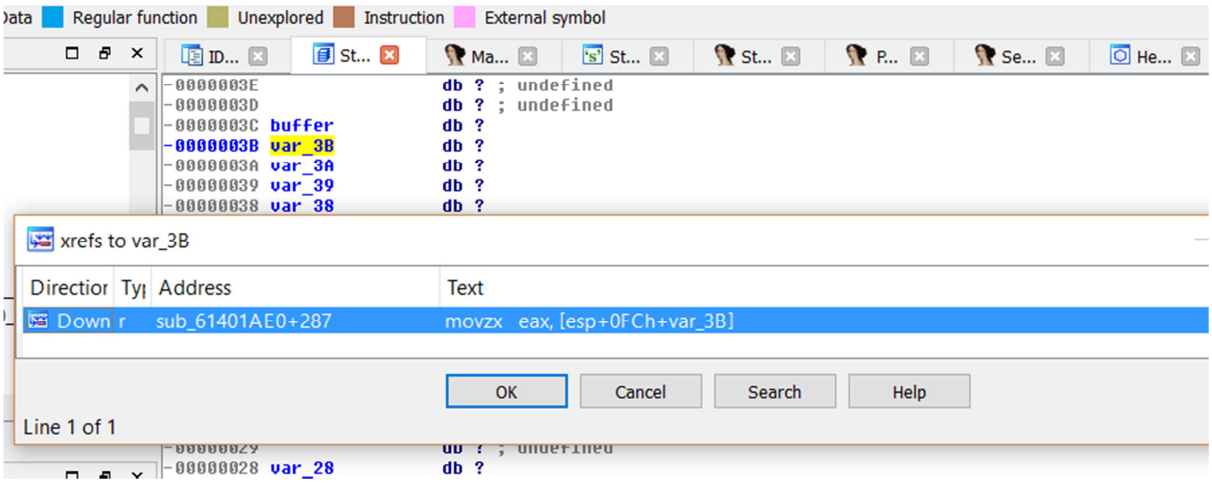
I see that ESI is the address of the buffer in the HEAP and it adds ECX that comes from EDI that is the counter, so in this function there are heap overflows since as we saw MAXIMO can be a value bigger than the size that was allocated and it overflows. There is another modification that goes a little unnoticed.



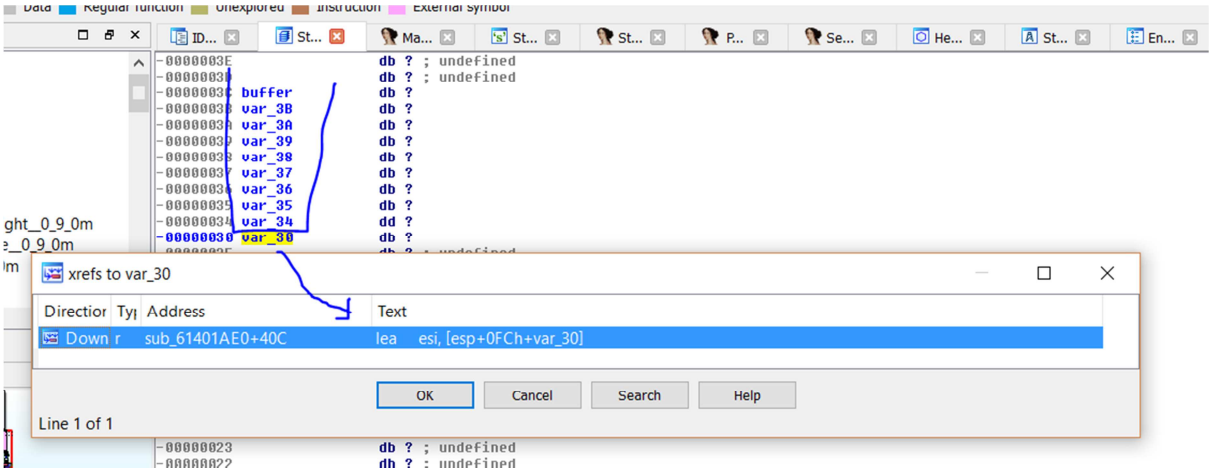
I see there are calls to a stream_Read function. I could read a part of the file to a temporary buffer.

We see that there is a LEA, so it will be a buffer in the stack.

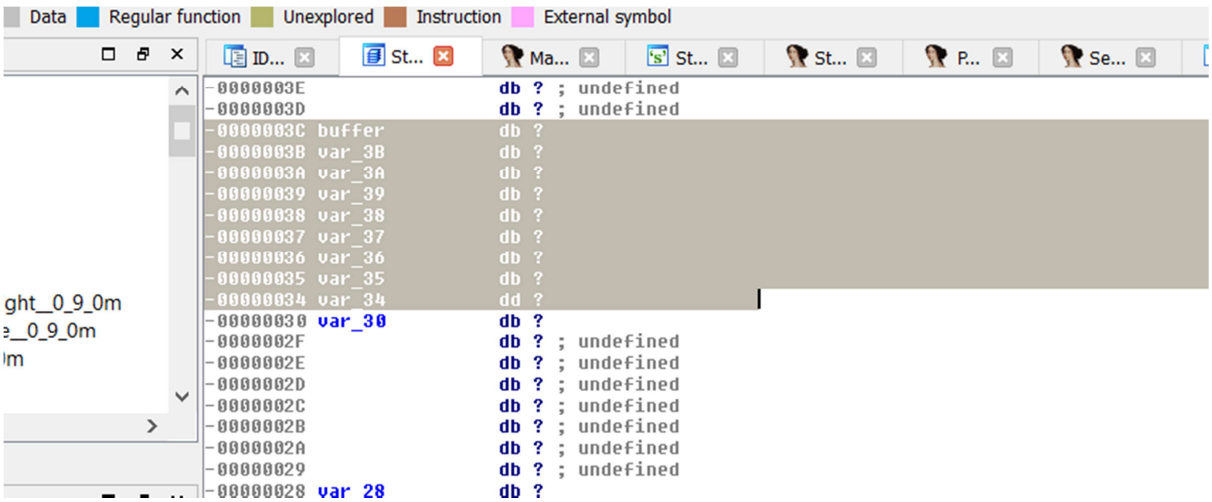
And the variables that follow are part of the buffer because it is never stored in them, it is only read, so it is sure to fill the variables below when filling the buffer.



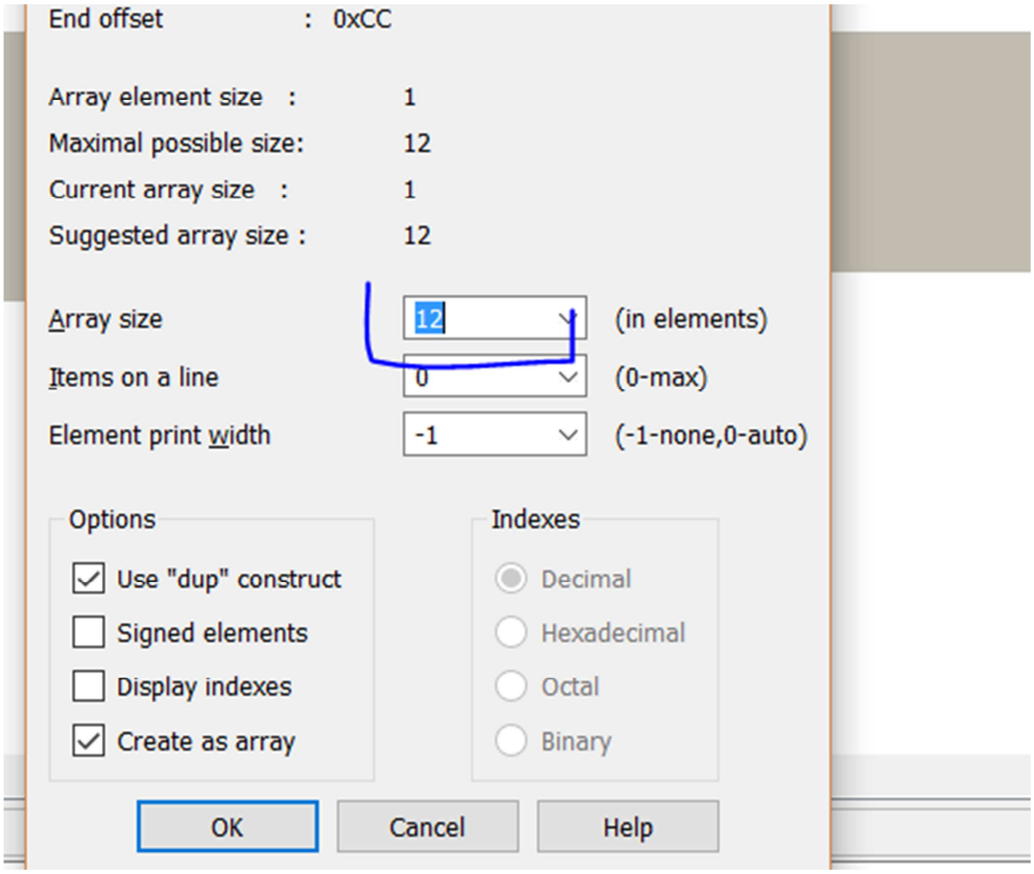
So the buffer continues here.



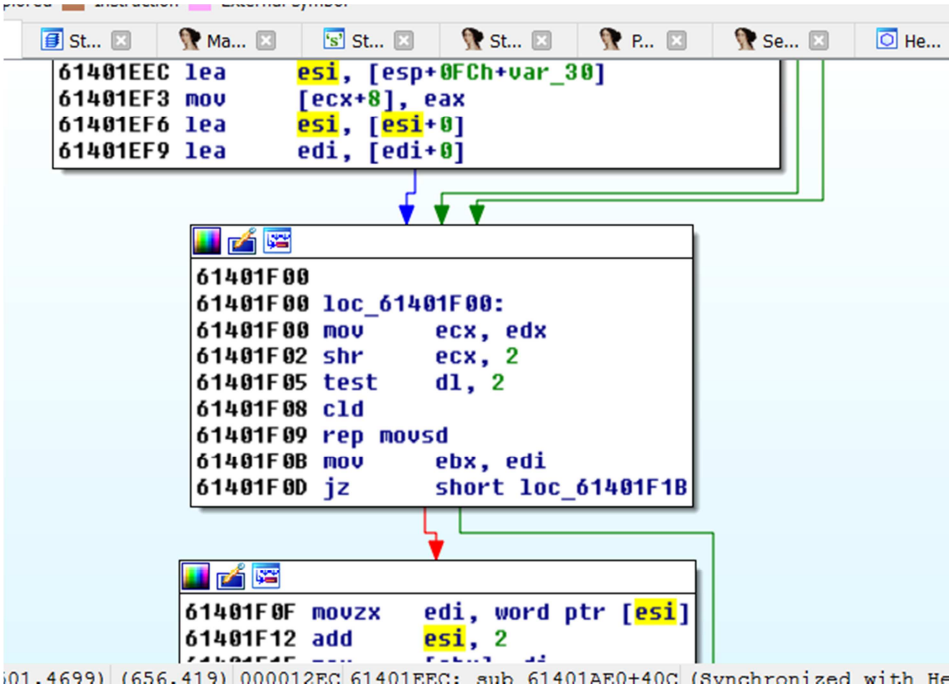
Since var_30 already has reference as another buffer, so I will mark it to see the size of the buffer.



Now, I right click ARRAY and accept.



I see the size is 12.



I see that the next buffer does not fill it but it reads bytes from there, so it's part of the same buffer above because it cannot read bytes from it if it has no reference where it fills.

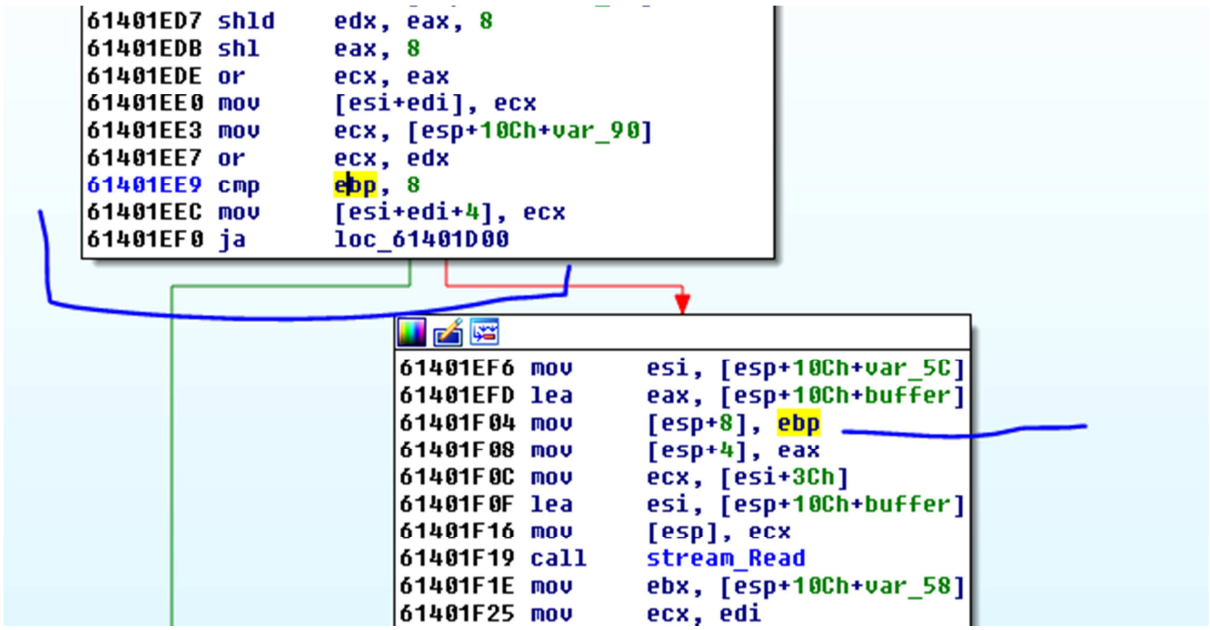
So let's look well. If we keep looking, we see that the buffer continues down here. All other intermediate variables have read access only, so they are initialized in the same buffer.

```
-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
-00000011                db ? ; undefined
-00000010                db ? ; undefined
-0000000F                db ? ; undefined
-0000000E                db ? ; undefined
-0000000D                db ? ; undefined
-0000000C                db ? ; undefined
-0000000B                db ? ; undefined
-0000000A                db ? ; undefined
-00000009                db ? ; undefined
-00000008                db ? ; undefined
-00000007                db ? ; undefined
-00000006                db ? ; undefined
```

Now if even var_1c is another buffer that is used to fill in another call to stream_Control.

The issue is that it is a small buffer, only 32 bytes, if you can pass a large value, it will overflow.

Here we see the patch on the new one on that value check that if it is greater than 8 it will not go to stream_Read.

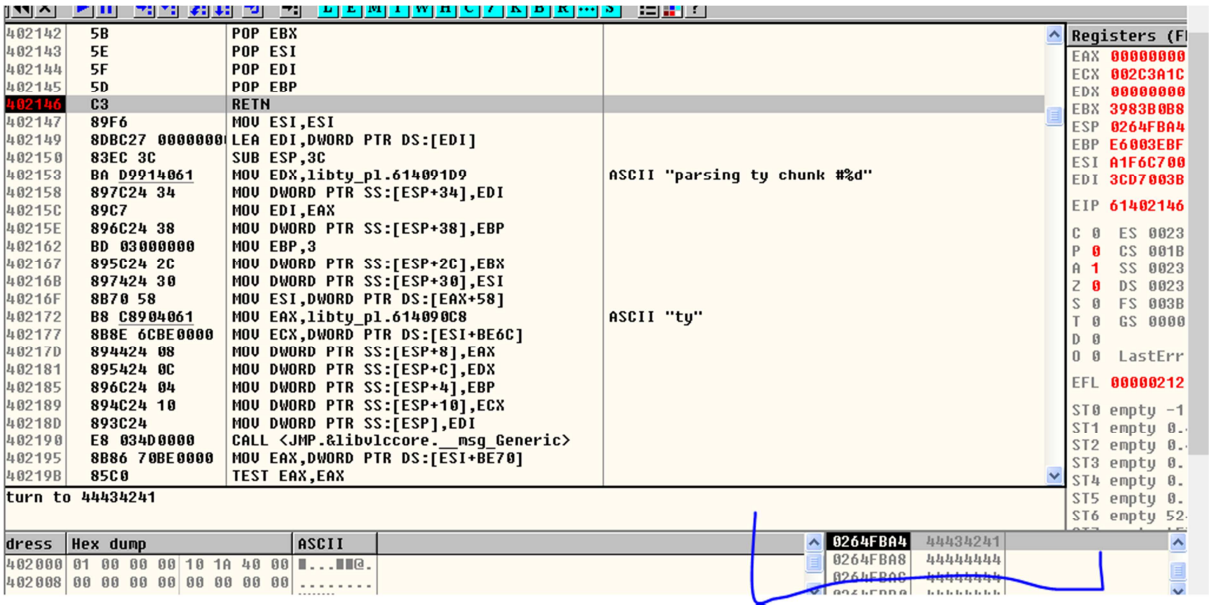


There, this is surely inside stream_Read. There will be some memcpy copying DWORDS. So it compares if it is greater than 8, because if you copy more than 8 dwords, it will be 8 * 4 greater than 32 which is the length of the buffer, so putting there a value greater than 8 we will have a stack overflow.

Let's look for a .ty file to try.

https://samples.libav.org/TiVo/test-dtivo-junkskip.ty%2B

I converted that sample into a POC to produce a stack overflow, changing the value that is filtered and adjusting some more that are around, so that it reaches the point of the stream_Read.



There, I tested it on an OLLYDBG on XP that I have for testing, but it works. This jumps to execute the address 0x**44434241** that I placed in the file.

The next exercise is to take the original file and modify it by setting a POC like the one I did to produce the stack overflow. It's simple because it's all analyzed. Debugging a little, you'll make it.

Ricardo Narvaja

Translated by: @IvinsonCLS