

REVERSING WITH IDA PRO FROM SCRATCH

PART 30

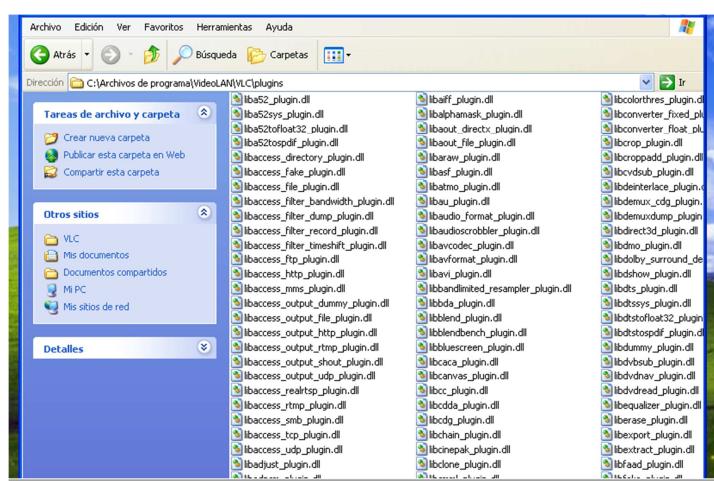
We will begin to try to solve the exercise proposed in part 29. This is a Diff of two consecutive versions of VLC and you have the CVE as a help with the information contained therein and that we almost always really have.

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.

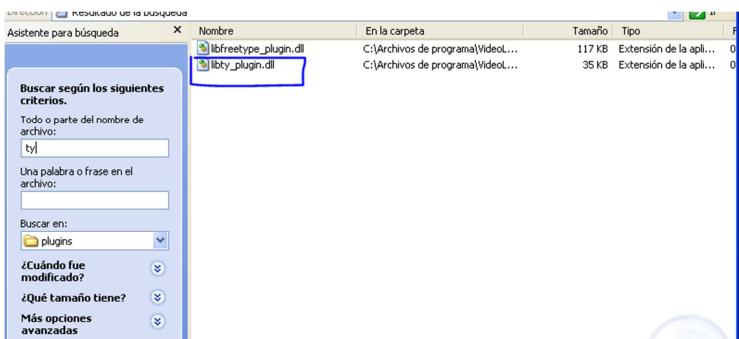
<https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2008-4654>

The screenshot shows the CVE-2008-4654 page on cve.mitre.org. The page has a header with links to Home, CVE IDs, About CVE, Compatible Products & More, Community, Blog, News, and Site Search. It also displays a total of 79949 CVE IDs. The main content area is titled 'CVE-ID' and shows 'CVE-2008-4654'. It includes a link to the National Vulnerability Database (NVD). Below this, there's a 'Description' section which states: 'Stack-based buffer overflow in the parse_master function in the Tivo demux plugin (modules/demux/tivo.c) in VLC Media Player 0.9.0 through 0.9.4 allows remote attackers to execute arbitrary code via a Tivo TY media file with a header containing a crafted size value.' There's also a 'References' section with a note that the list is not intended to be complete, and a single reference entry: 'BUGTRAQ:20081020 [TKADV2008-010] VLC media player Tivo ty Processing Stack Overflow'.

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

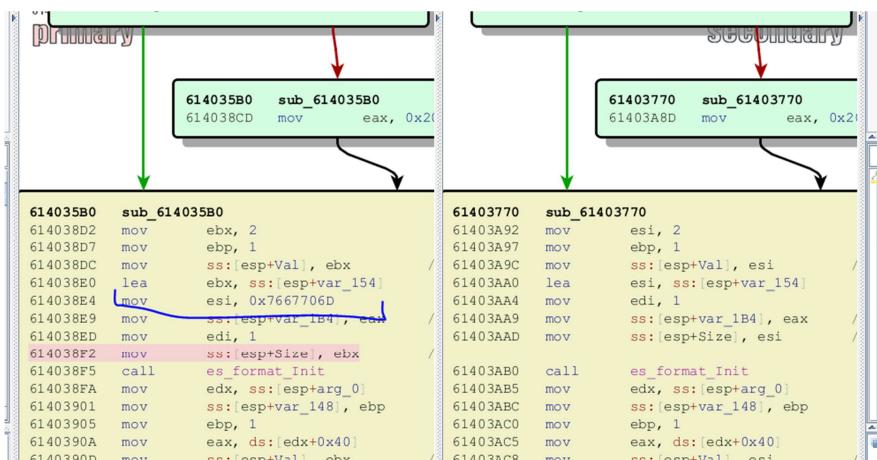


There is a libty_plugin.dll which seems to be quite suspicious. Let's do the diff on it.

	ID...	Ma...	St...	St...	P...	Se...	He...	A St...	En...	Im...	Ex...
similarity	confidence	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-----	61403580	sub_61403580_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.

```

0_61403770 - BinDiff
edition Search Window Help
x_61403590 vs sub_61403770
File Edit View Options Search Window Help
mov    ebp, 1
mov    ss: esp+Val , ebx
lea    ebx, ss: esp+var_154
mov    esi, 0x7667706D
mov    ss: esp+var_1B4 , eax
mov    edi, 1
mov    ss: esp+Size , ebx
call   es_format_Init
mov    edx, ss: esp+arg_0
mov    ss: esp+var_148 , ebp
mov    ebp, 1
mov    eax, ds: edx+0x40
mov    ss: esp+Val , ebx
mov    ss: esp+Size , eax
call   ds: eax
mov    ecx, ss: esp+var_180
mov    ds: ecx+4 , eax
mov    ss: esp+var_1B4 , esi
xor    esi, esi
mov    ss: esp+Val , edi
xor    edi, edi

```

```

0_61403770 - BinDiff
edition Search Window Help
x_61403590 vs sub_61403770
File Edit View Options Search Window Help
mov    ebp, 1
mov    ss: esp+Val , esi
lea    esi, ss: esp+var_154
mov    edi, 1
mov    ss: esp+var_1B4 , eax
mov    ss: esp+Size , esi
call   es_format_Init
mov    edx, ss: esp+arg_0
mov    ss: esp+var_148 , ebp
mov    ebp, 1
mov    eax, ds: edx+0x40
mov    ss: esp+Val , esi
mov    ss: esp+Size , eax
call   ds: eax
mov    ebx, ss: esp+var_180
mov    ecx, 0x7667706D
mov    ds: ebx+4 , eax
xor    ebx, ebx
mov    ss: esp+var_1B4 , ecx
mov    ss: esp+Val , edi
xor    edi, edi

```

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.

Function	Address	Instruction	Comment
sub_61403140	031A0	sub_61403140	
	031A7	mov eax, ss:[esp+arg_8]	// jumps to 0336E
	031AE	mov edi, ss:[esp+arg_8]	
	031B5	imul ecx, ds:[eax+4], 0x3E8	
	031BC	mov eax, 0x3E8	
	031C1	mul ds:[edi]	
sub_61403300	03300	sub_61403300	
	03367	mov edi, ss:[esp+arg_8]	// jumps to 0336E
	0336E	mov eax, 0x3E8	
	03373	mul ds:[edi]	
	03375	imul ecx, ds:[edi+4], 0x3E8	
	0337C	lea edx, ds:[ecx+edx]	
	0337F	mov ecx, edx	
	03381	mov edx, eax	

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.

primary			secondary		
61403140	sub_61403140		103300	sub_61403300	
6140333C	mov eax, ebp		1034F6	mov eax, ebp	
6140333E	call 0x61401AE0		1034F8	call 0x61401AE0	
61403343	mov ecx, ds:[edi+0xBE6C]		1034FD	mov eax, ds:[edi+0xBE6C]	
61403349	mov eax, 2				
6140334E	mov ss:[esp+var_78], eax // var_78				
61403352	shl ecx, bl 0x11		103503	shl eax, bl 0x11	
61403355	mov edx, ecx		103506	cdq	
61403357	sar edx, bl 0x1F		103507	mov ss:[esp+var_70], edx // var_70	
6140335A	mov ss:[esp+var_74], ecx // var_74		10350B	mov edx, 2	
6140335E	mov ss:[esp+var_70], edx // var_70		103510	mov ss:[esp+var_74], eax // var_74	
61403362	mov esi, ss:[ebp+0x3C]		103514	mov ss:[esp+var_78], edx // var_78	
61403365	mov ss:[esp+var_7C], esi // var_7C		103518	mov eax, ss:[ebp+0x3C]	
61403368	call stream_Control		10351B	ss:[esp+var_7C], eax // var_7C	
6140336D	test eax, eax		10351E	call stream_Control	
			103523	test eax, eax	

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

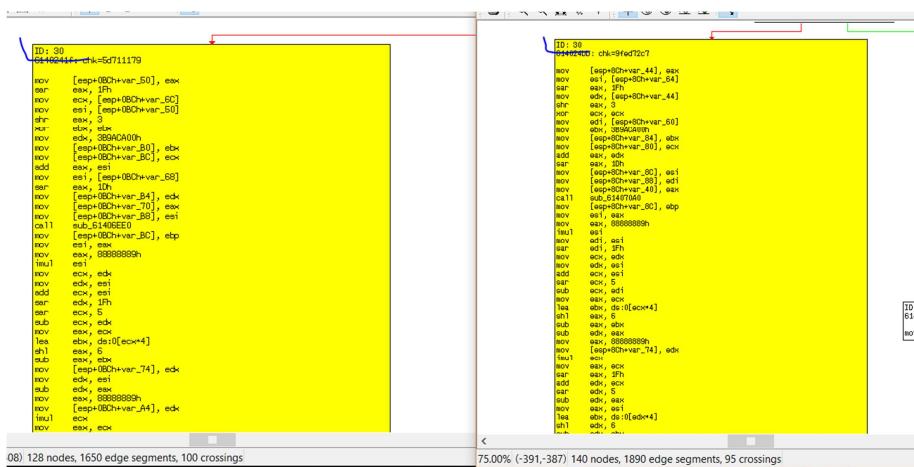
The next one is very messy. We will see if we can accommodate it a little.

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

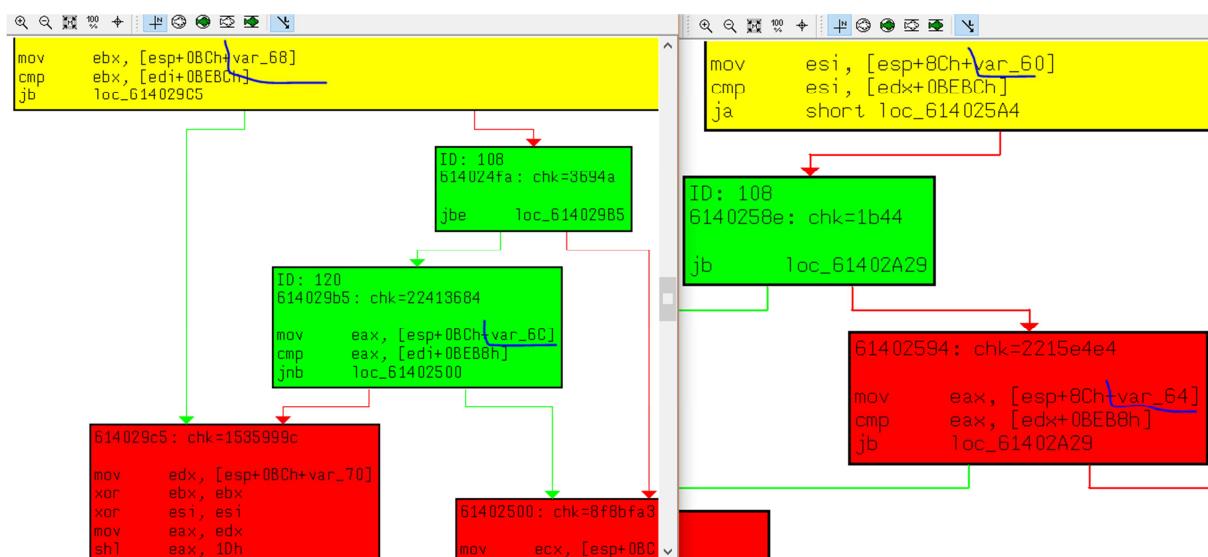
There are blocks that look very different when being matched badly but when we match them well...

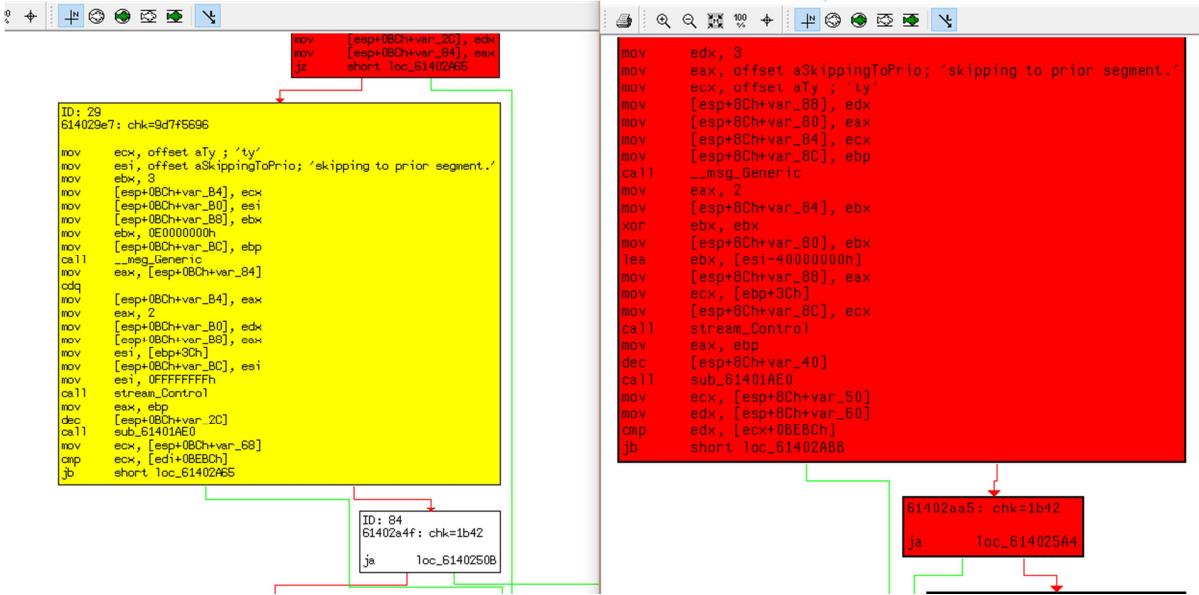
If I look at them, they look the same now, but they look a little badly drawn.

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

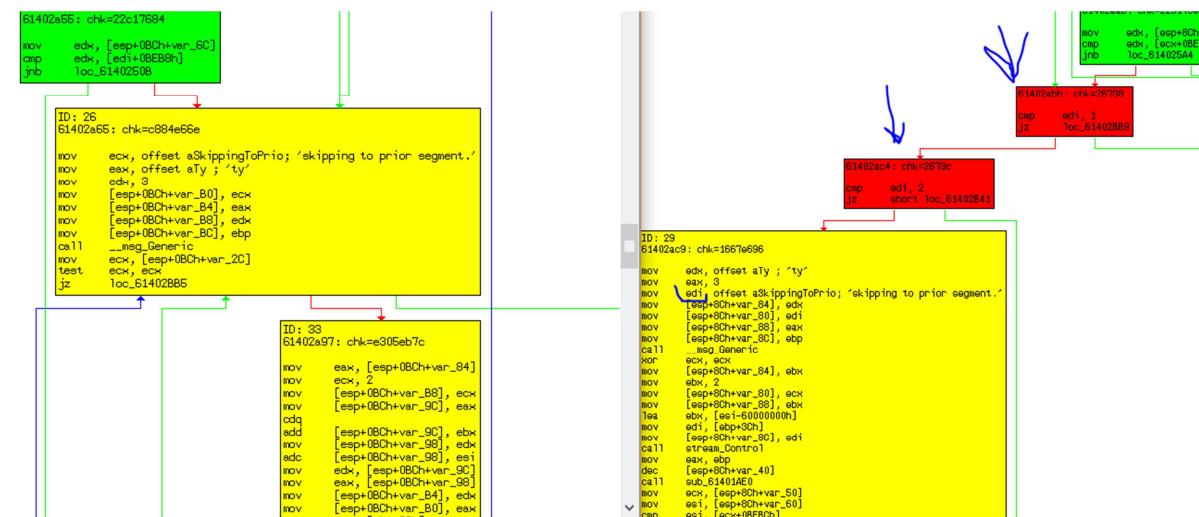


Let's look at the block IDs.

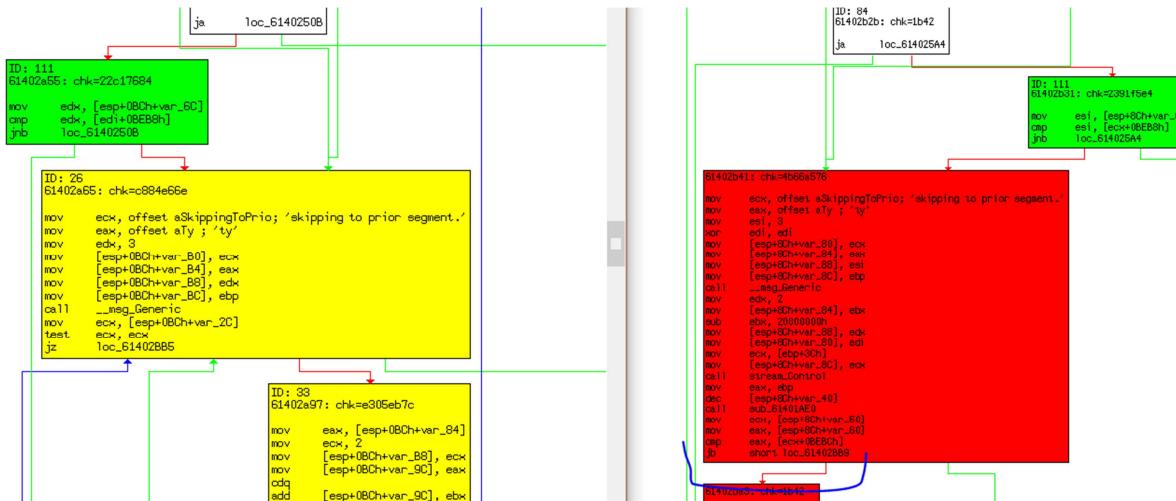




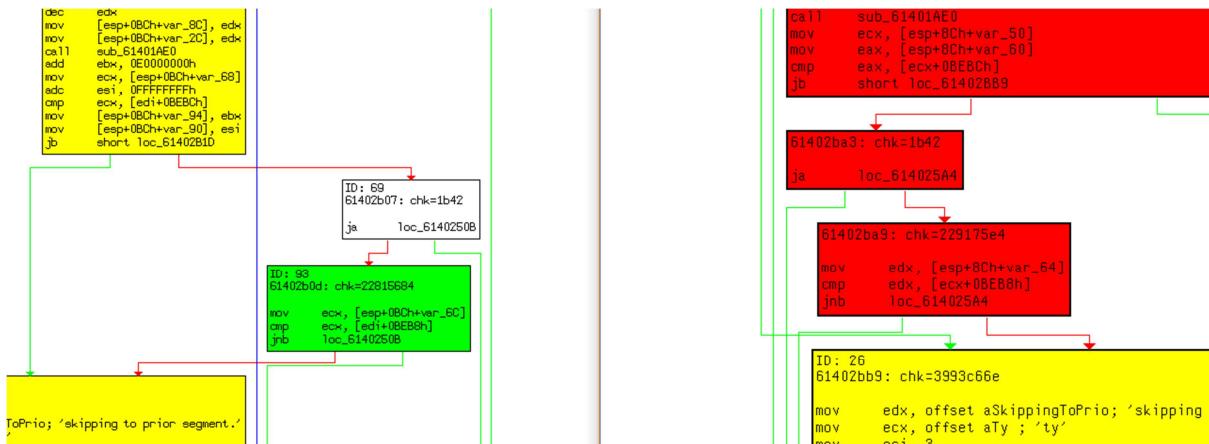
At bird's eye view here, it looks pretty similar. There is no change of sign in the comparison. Minor changes.



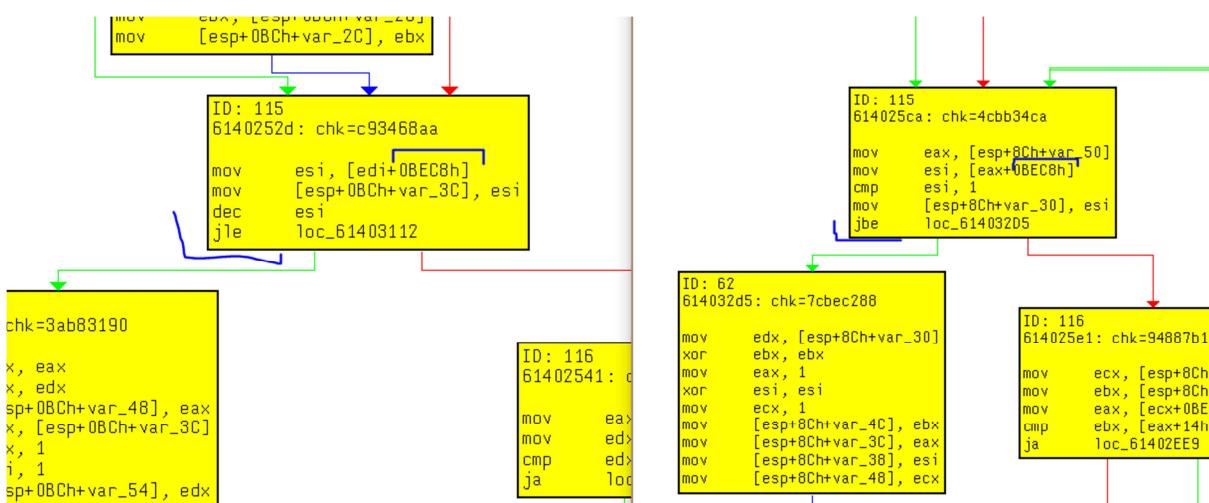
There, we see a couple of filters that the old one does not have. As EDI is overwritten just later, so they can be aggregated cases. We leave them marked but it is not seen as suspicious of overflow.



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.



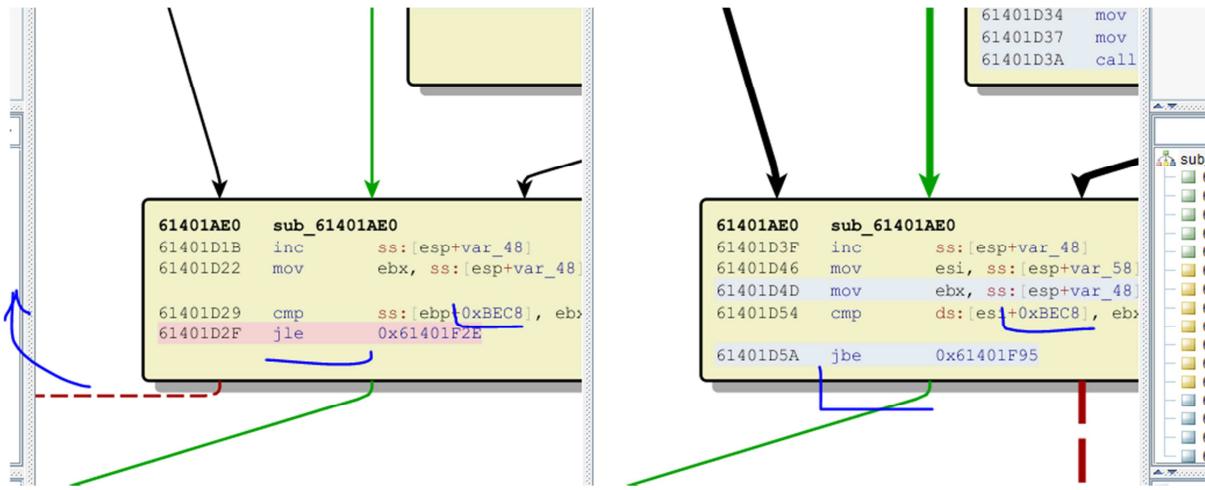
Recall that we are not reversing in depth just looking for something that catches our attention as very likely.



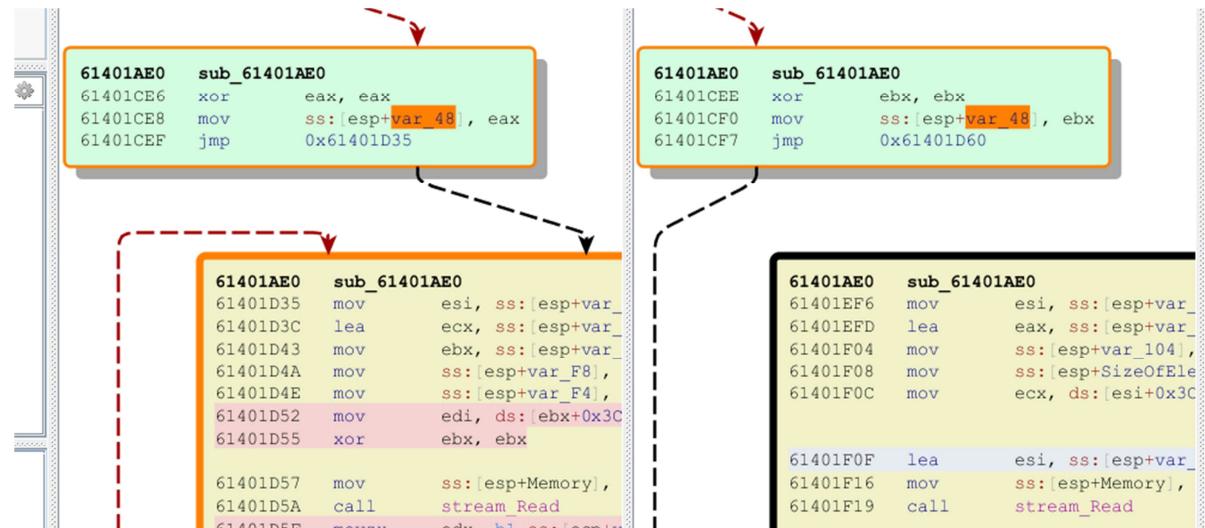
Here, if you see something very possible that affects, a field of the structure that compares, and in one makes a decision with JLE and in the other with JBE that is a change of sign and we point it as possible.

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.

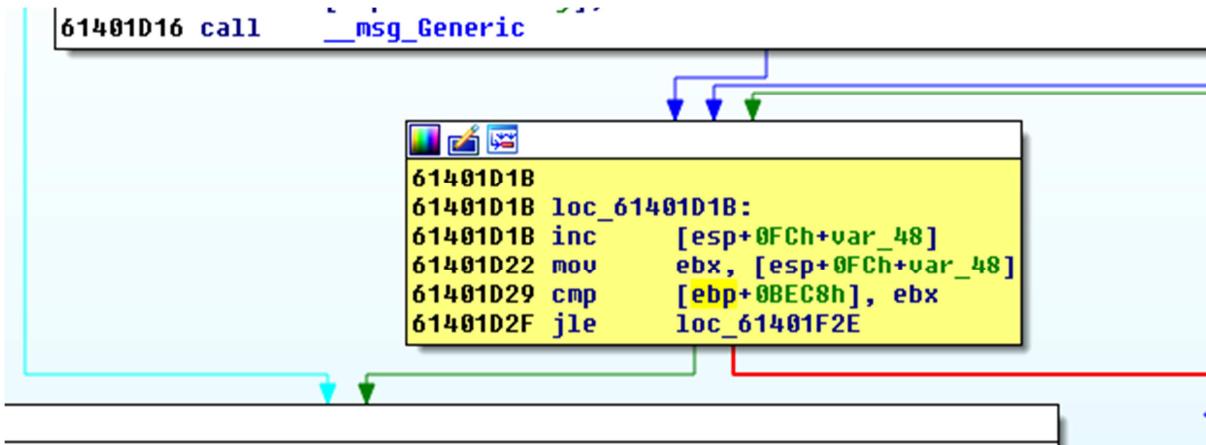
In this function, the same field of structure affects and it is easier to see.



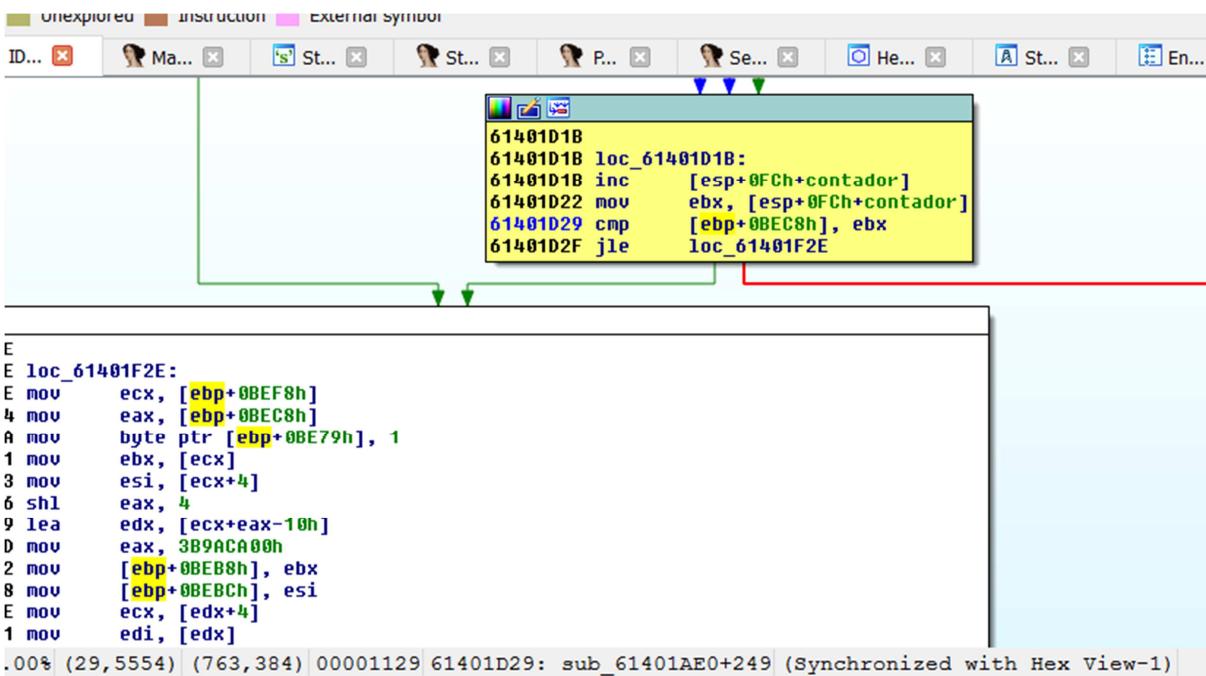
It is a loop and the value that decides the output is a counter that is in var_48 and is incremented and compared to the maximum.



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.

```

61401BA0 mov [esp+0FCh+var_CC], edx
61401BA4 mov [esp+0FCh+var_C8], edi
61401BA8 mov [esp+0FCh+var_D4], esi
61401BAC mov [esp+0FCh+var_D0], ebp
61401BB0 mov [esp+0FCh+var_60], eax
61401BB7 mov edx, [esp+0FCh+var_60]
61401BBE mov ebp, [eax+58h]
61401BC1 mov [esp+0FCh+var_F8], ecx
61401BC5 mov [esp+0FCh+var_F4], ebx
61401BC9 mov eax, [edx+3Ch]
61401BCC mov [esp+0FCh+Memory], eax
61401BCF call stream_Control
61401BD4 mov edi, [esp+0FCh+var_1C]
61401BDB mov esi, [esp+0FCh+var_18]
61401BE2 mov ebx, [ebp+0BEF8h]
61401BE8 mov [esp+0FCh+var_54], edi
61401BEF mov [esp+0FCh+var_50], esi
61401BF6 mov [esp+0FCh+Memory], ebx ; Memory
61401BF9 call free
61401BFE mov eax, [esp+0FCh+var_60]
61401C05 mov ecx, 20h
61401C0A lea edx, [esp+0FCh+var_3C]
61401C11 mov [esp+0FCh+var_F4], ecx
61401C15 mnu [esp+0FCh+var_F81], edx

```

(222,1876) (88,87) 00001129 61401D29: sub_61401AE0+249 (synchronized with)

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

```

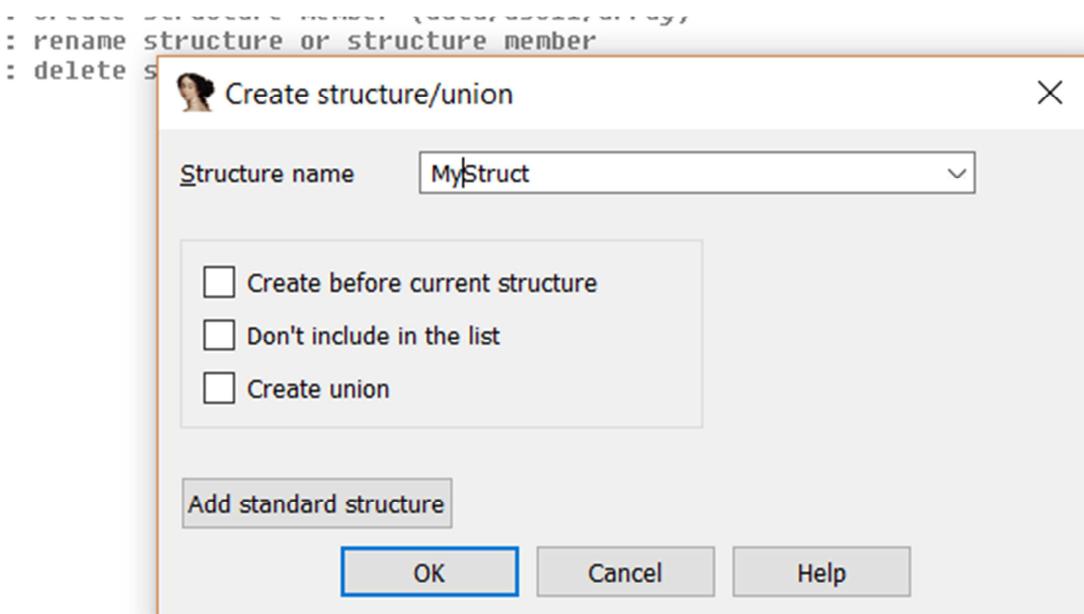
61402000 mov [esp+0FCh+var_40], ecx
61402004 call sub_61407210
61402009 mov ecx, 3
6140200E mov [esp+0FCh+var_F8], ecx
61402012 mov [esp+0FCh+var_F0], esi
61402016 mov [esp+0FCh+var_F4], edi
6140201A mov edi, 3Ch
6140201F mov [esp+0FCh+var_EC], eax
61402023 mov eax, [esp+0FCh+var_60]
6140202A mov [esp+0FCh+Memory], eax
6140202D call __msg_Generic
61402032 mov edx, [ebp+0BECE0h]
61402038 mov esi, [ebp+0BECE4h]
6140203E mov ebp, 3B9ACA00h
61402043 mov [esp+0FCh+var_F4], ebp
61402047 mov ebp, 3Ch
6140204C mov [esp+0FCh+var_F0], ebx
61402050 mov [esp+0FCh+Memory], edx
61402053 mov [esp+0FCh+var_F8], esi
61402057 call sub_61406EE0
6140205C xor ecx, ecx

```

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.



```
x ID... Ma... St... St... P... Se... He... A S
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=0x0)
00000000 MyStruct    ends
00000000
```

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

```
00000000 ; D/H/* : CREATE STRUCTURE MEMBER (data/ascii/array)
00000000 ; N      : rename structure or structure member
00000000 ; U      : delete structure member
00000000 ;
00000000 ;
00000000 MyStruct    struc ; (sizeof=0x1, mappedto_1)
00000000 Field_0     db ?
00000001 MyStruct    ends
00000001
```

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      struct ; (sizeof=0x1, mappedto_1)
00000000 Field_0        db ?
00000001 MyStruct      ends
00000001
```

Expand struct X

Number of bytes to add

OK Cancel Help

The screenshot shows the IDA Pro interface. The assembly view at the top displays:

```
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 MyStruct      struct ; (sizeof=0xBFO1, 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
```

The search results window below shows:

```
1. MyStruct:0000
```

There, it is.

The screenshot shows the Immunity Debugger interface. A yellow callout box highlights a section of assembly code:

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

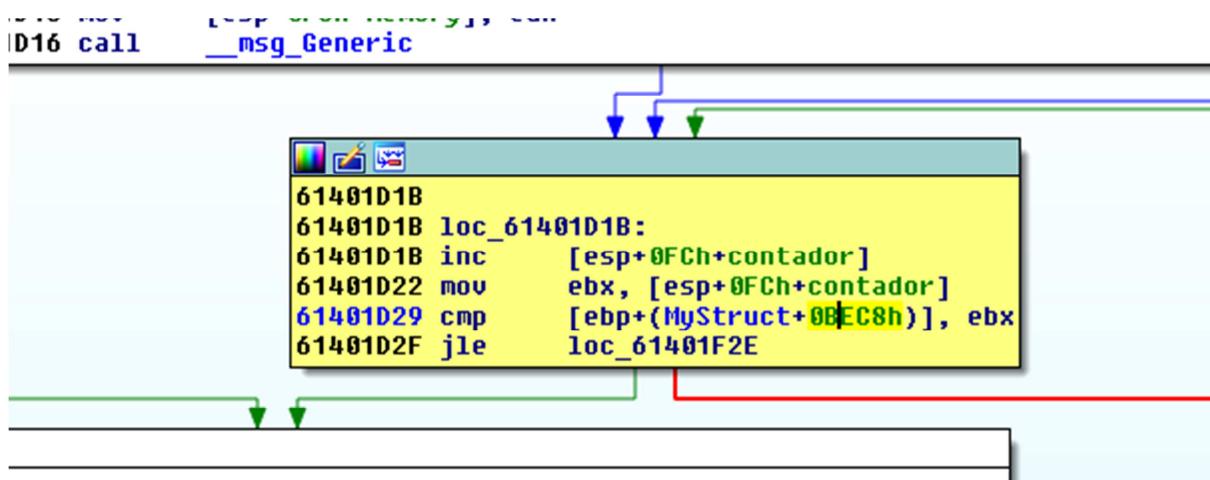
Annotations include:

- A blue bracket above the first four instructions (inc, mov, cmp, jle) with three green arrows pointing down to each instruction.
- A red bracket below the last two instructions (cmp, jle) with one green arrow pointing down to the cmp instruction.

Obviously, if it were possible that the 0xbec8 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.



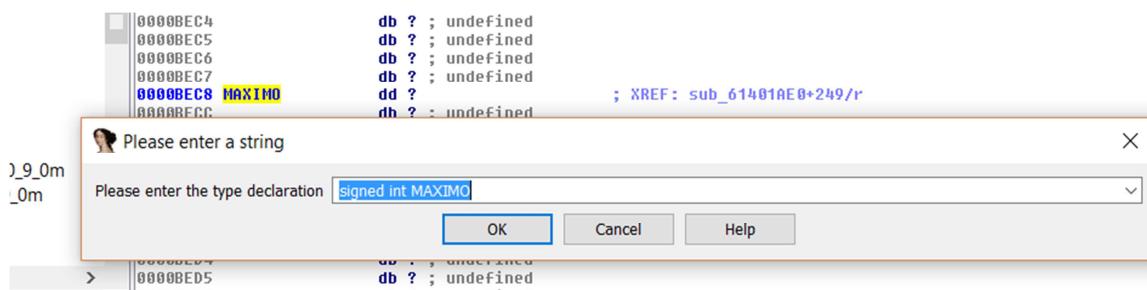
And I should go to the structure at 0bec8 and create a DWORD field as that is what it is.

0000BEC2	db ? ; undefined
0000BEC3	db ? ; undefined
0000BEC4	db ? ; undefined
0000BEC5	db ? ; undefined
0000BEC6	db ? ; undefined
0000BEC7	db ? ; undefined
0000BEC8	db ? ; undefined
0000BEC9	db ? ; undefined
0000Beca	db ? ; undefined
0000Becb	db ? ; undefined
0000Becc	db ? ; undefined
0000Becd	db ? ; undefined
0000Bece	db ? ; undefined
0000Becf	db ? ; undefined
0000Bed0	db ? ; undefined

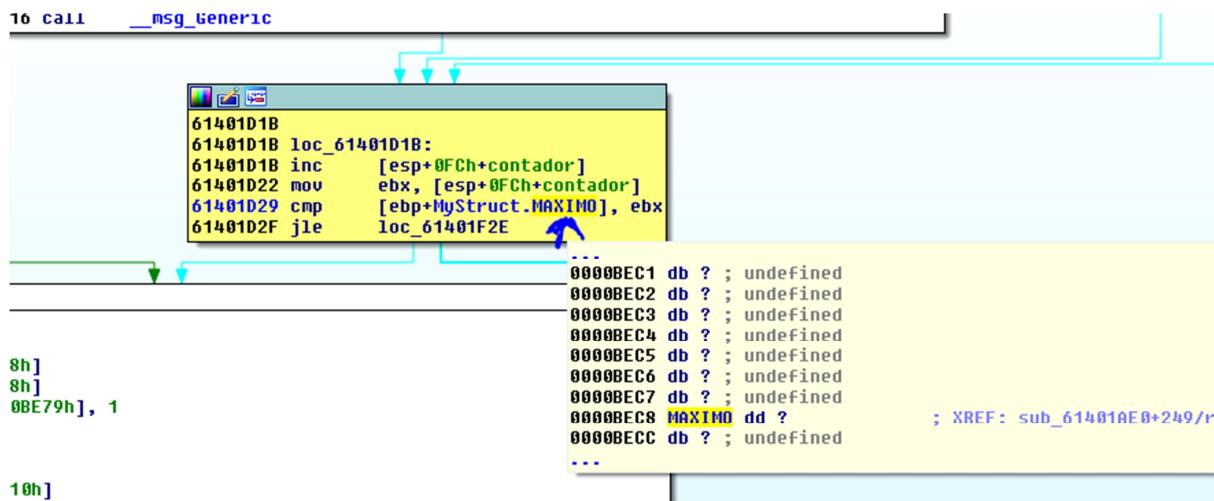
I press D there several times until it shows DD.

0m		
	0000BEC2	db ? ; undefined
	0000BEC3	db ? ; undefined
	0000BEC4	db ? ; undefined
	0000BEC5	db ? ; undefined
	0000BEC6	db ? ; undefined
	0000BEC7	db ? ; undefined
	0000BEC8 Field_BEC8	dd ? ; undefined
	0000BEC9	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.

```

401CF1
401CF1 loc_61401CF1:
401CF1 mov    eax, [esp+0FCh+var_60]
401CF8 mov    ebx, offset aUnsupportedSeq ; "Unsupported SEQ bitmap size in master c"...
401CFD mov    edi, offset aby ; "by"
401D02 mov    edx, 1
401D07 mov    [esp+0FCh+var_F0], ebx
401D08 mov    [esp+0FCh+var_F4], edi
401D0F mov    [esp+0FCh+var_F8], edx
401D13 mov    [esp+0FCh+Memory], eax
401D16 call   __msg_Generic

61401D18 loc_61401D18:
61401D18 inc    [esp+0FCh+contador]
61401D22 mov    ebx, [esp+0FCh+contador]
61401D29 cmp    [ebp+MyStruct.MAXIMO], ebx
61401D2F jle   loc_61401F2E

```

I'll continue studying.

```

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+0FCh+Memory], eax ; Size
61401CCD call   malloc
61401CD2 mov    [ebp+0BEC8h], eax
61401CD8 mov    eax, [ebp+0BEC8h]
61401CDE test   eax, eax
61401CF0 jne   loc_61401F2F

```

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.

```
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  
61401CD2 mov    [ebp+0BEF8h], eax  
61401CD8 mov    eax, [ebp+0BEC8h]  
61401CDE test   eax, eax  
61401CE0 jle    loc_61401F2E
```

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      ; Equivalent 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
61401C80 or     Leax, ecx
61401C82 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
61401C92 mntr   Tephn+0RFF8h1 eax

```

We see that the maximum negative values are filtered here.

```

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

```



```

61401CE6 xor    eax, eax
61401CE8 mov    [esp+0FCCh+contador], eax
61401CEF jmp   short loc 61401D35

```

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.

◀ **calloc**

Visual Studio 2015 | Otras versiones ▾

hl

Publicado: julio de 2016

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

Asigna una matriz en memoria con elementos inicializado a 0.

Sintaxis

```

void *calloc(
    size_t num,
    size_t size
);

```

Parámetros

num
Número de elementos.

size
Longitud en bytes de cada elemento.

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.

```

61401CA7 shl    esi, 8
61401CAA or     edx, esi
61401CAC mov    eax, edx
61401CAE xor    edx, edx
61401CB0 div    ebx
61401CB2 mov    ebx, [esp+10Ch+var_58]
61401CB9 mov    ecx, eax
61401CBB mov    [ebx+0BEF8h], eax
61401CC1 mov    eax, 10h
61401CC6 mov    [esp+10Ch+SizeOfElements], eax ; SizeOfElements
61401CCA mov    [esp], ecx ; NumOfElements
61401CCD call   calloc
61401CD2 test   eax, eax
61401CD4 mov    [ebx+0BEF8h], eax
61401CDA jz    loc_614021CD

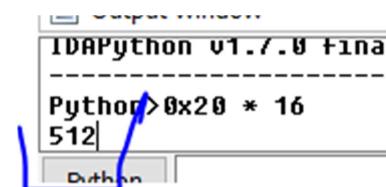
```

The screenshot shows the assembly view of the program. A red arrow points from the instruction `61401CE8 mov eax, [ebx+0BEF8h]` to the memory dump window below. A green arrow points from the instruction `61401CBB mov [ebx+0BEF8h], eax` to the same memory dump window. The memory dump window displays the value `614021CD`.

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:

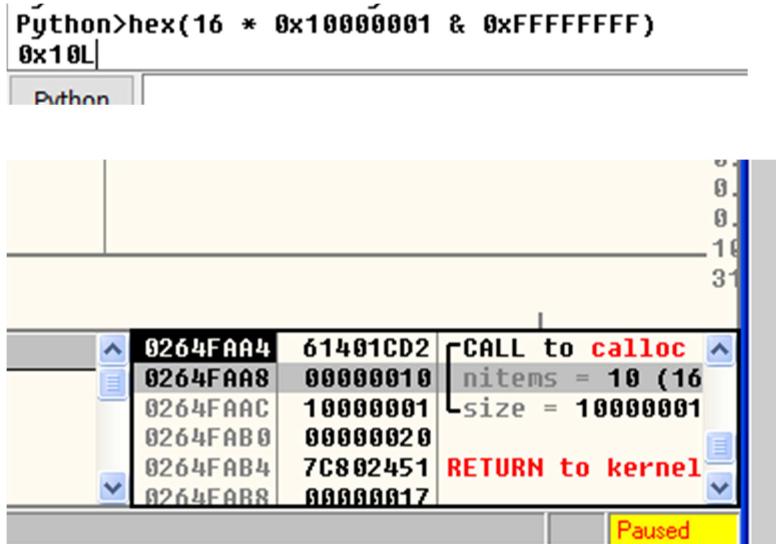


This will reserve 512 bytes and then copy 0x20 because it compares inside the loop and exits when the counter is greater than MAXIMO.

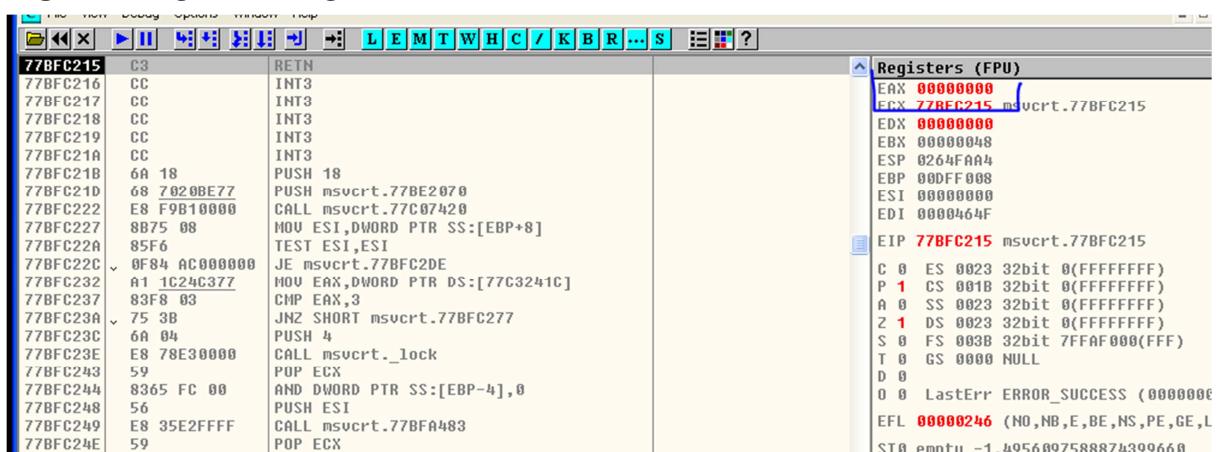
Now, what happens if the MAXIMO value is positive but multiplying it by 16 is smaller than the initial value?

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.

```

000BEF5          db ? ; undefined
000BEF6          db ? ; undefined
000BEF7          db ? ; undefined
000BEF8 BUFFER_HEAP dd ? ; offset
000BEFC          db ? ; undefined
000BEFD
000BEFE
0000BF00 Please enter a string
0000BF01 Please enter the type declaration char *BUFFER_HEAP[]
0000BF02
0000BF03

```

Please enter a string

Please enter the type declaration

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.

```

01401E0F or     ebx, ebx
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_98], 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_98]
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.

```

61401D5A call    stream_Read
61401D5F movzx  edx, [esp+0FCh+var_3C]
61401D67 movzx  eax, [esp+0FCh+var_3B]
61401D6F mov    edi, [esp+0FCh+contador]
61401D76 mov    [esp+0FCh+var_9C], edx
61401D7A xor    edx, edx
61401D7C mov    ecx, [esp+0FCh+var_9C]
61401D80 mov    [esp+0FCh+var_A4], eax
61401D84 shl    edi, 4
61401D87 mov    eax, [esp+0FCh+var_A4]
61401D8B mov    ebx, ecx
61401D8D mov    ecx, 0
61401D92 mov    esi, ecx
61401D94 movzx  ecx, [esp+0FCh+var_35]
61401D9C mnll  edx  eax

```

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.

```

61401F09 rep movsd
61401F0B mov    ebx, edi
61401F0D jz     short loc_61401F1B

61401F0F movzx  edi, word ptr [esi]
61401F12 add    esi, 2
61401F15 mov    [ebx], di
61401F18 add    ebx, 2

61401F1B loc_61401F1B:
61401F1B test   dl, 1
61401F1E jz     loc_61401D1B

61401F24 movzx  edx, byte ptr [esi]
61401F27 mov    [ebx], dl
61401F2A inc    ebx
61401F2D inc    esi
61401F2E inc    edi

570,5007 (114,58) 00001315 61401F15: sub 61401AE0+435 (Synchronized with

```

We see that destination address is coming from here.

Unexplored Instruction External symbol

```
61401EDF mov     eax, [esp+0FCh+var_34]
61401EE6 lea     edi, [ecx+0Ch]
61401EE9 sub     edx, 4
61401E9C lea     esi, [esp+0FCh+var_30]
61401EF3 mov     [ecx+8], eax
61401EF6 lea     esi, [esi+0]
61401EF9 lea     edi, [edi+0]

61401F00
61401F00 loc_61401F00:
61401F00 mov     ecx, edx
61401F02 shr     ecx, 2
61401F05 test    dl, 2
61401F08 cld
61401F09 rep     mousd
61401F0B mov     ebx, edi
61401F0D jz     short loc_61401F1B
```

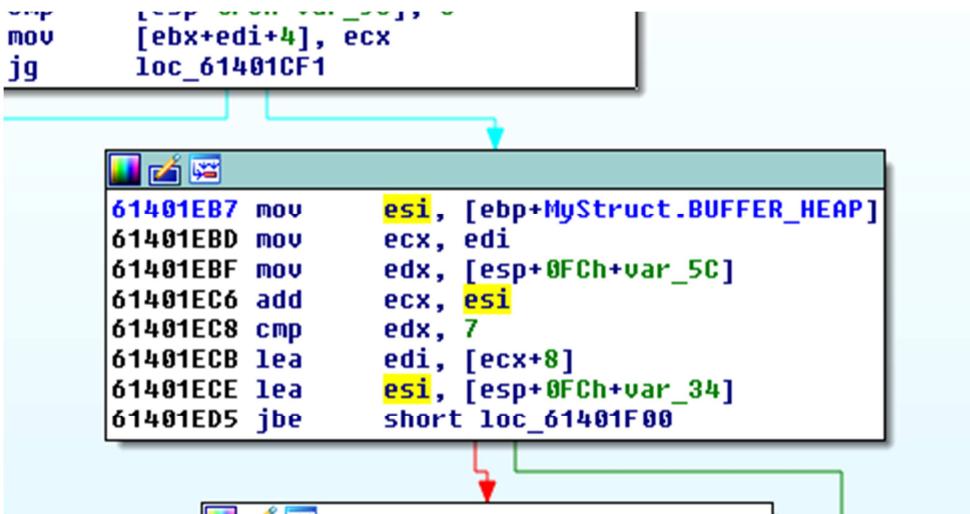
ECX + 0c is equal to EDI, so we'll look for where ECX comes from.

ECX leaves here.

```
loc_61401CF1
61401EB7 mov     esi, [ebp+0BEF8h]
61401EBD mov     ecx, edi
61401EBF mov     edx, [esp+0FCh+var_5C]
61401EC6 add     ecx, esi
61401EC8 cmp     edx, 7
61401ECB lea     edi, [ecx+8]
61401ECE lea     esi, [esp+0FCh+var_34]
61401ED5 jbe     short loc_61401F00
```

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.

```

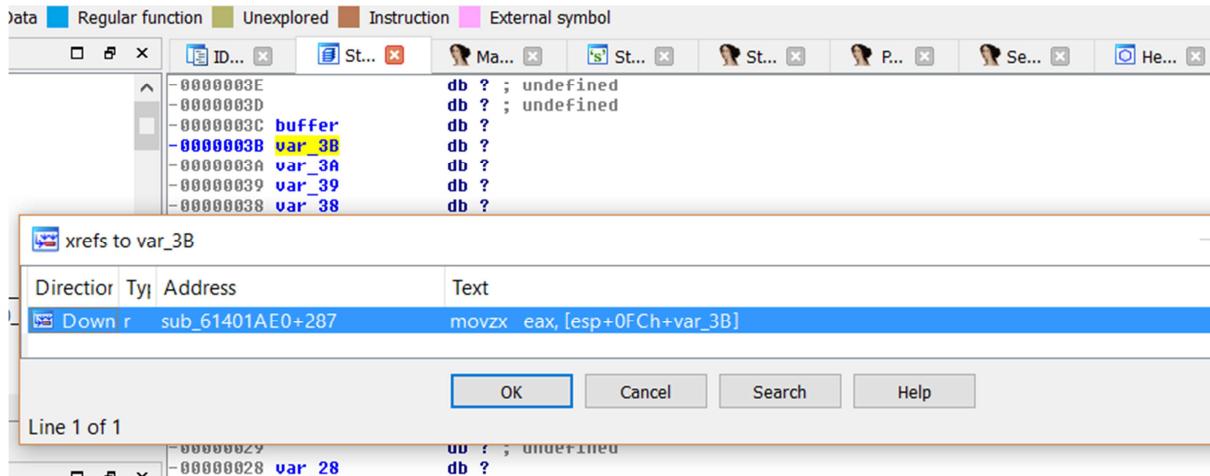
61401D35
61401D35 loc_61401D35:
61401D35 mov     esi, [esp+0FCh+valor2]
61401D3C lea     ecx, [esp+0FCh+buffer]
61401D43 mov     ebx, [esp+0FCh+var_60]
61401D4A mov     [esp+4], ecx
61401D4E mov     [esp+8], esi
61401D52 mov     edi, [ebx+3Ch]
61401D55 xor     ebx, ebx
61401D57 mov     [esp], edi
61401D5A call    stream_Read
61401D5F movzx   edx, [esp+0FCh+buffer]
61401D67 movzx   eax, [esp+0FCh+var_3B]
61401D6F mov     edi, [esp+0FCh+contador]
61401D76 mov     [esp+0FCh+var_9C], edx
61401D7A xor     edx, edx
61401D7B ...

```

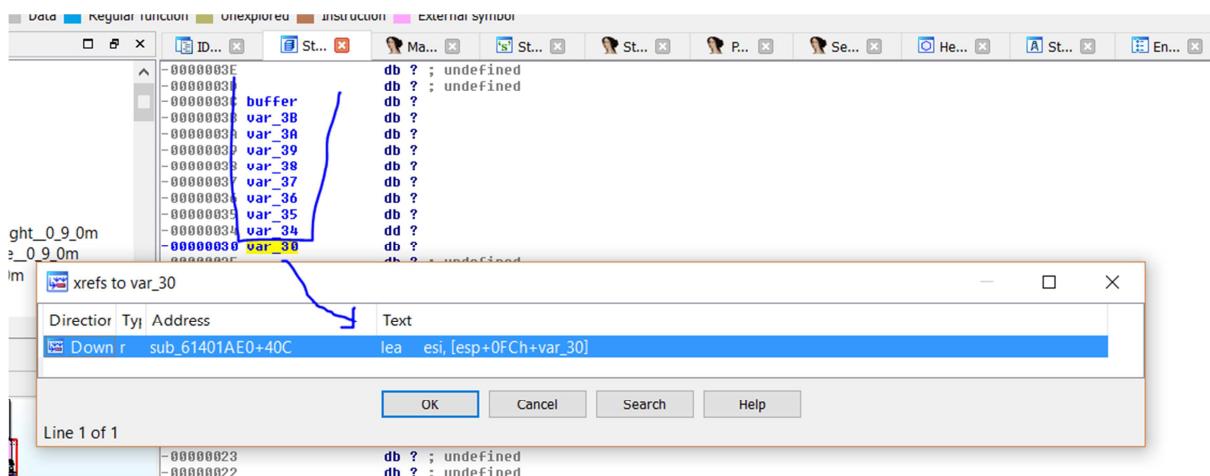
I see there are calls to a stream_Read function. I could read a part of the file to a temporary buffer.

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

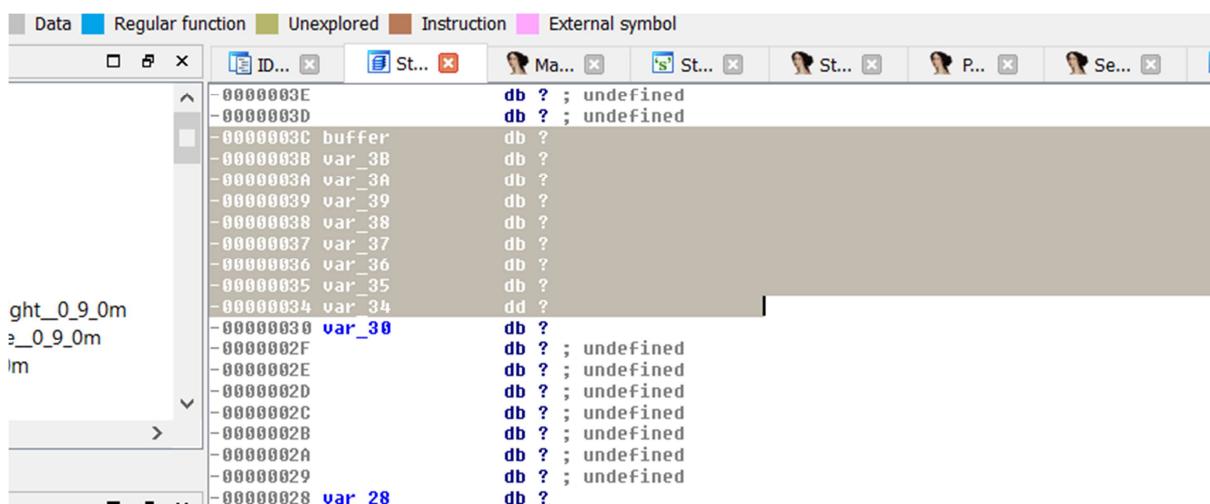
And the variables that follow are part of the buffer because it is never stored in them, it is only read, so it is sure to fill the variables below when filling the buffer.



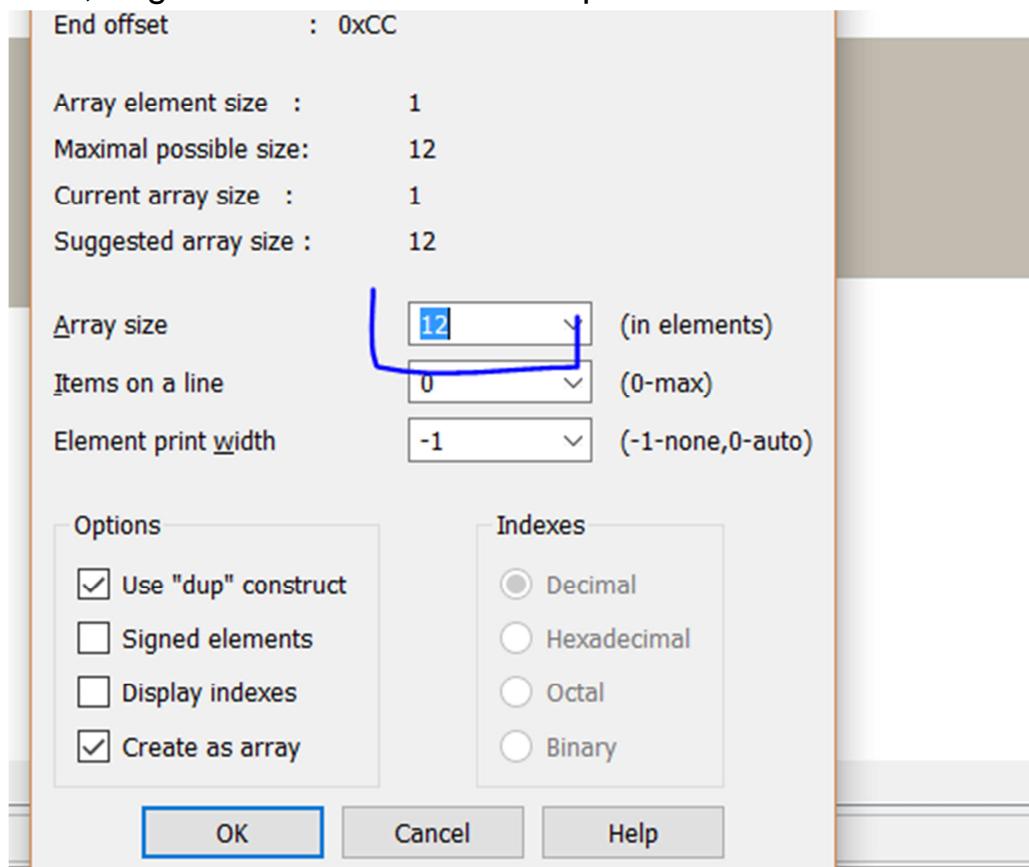
So the buffer continues here.



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.

The screenshot shows the Immunity Debugger assembly view with three code snippets highlighted by green boxes and connected by arrows:

- Top snippet (instruction 61401EEC):

```
61401EEC lea    esi, [esp+0FCh+var_30]
```
- Middle snippet (instruction 61401F00):

```
61401F00 loc_61401F00:
```

```
61401F00 mov    ecx, edx
```

```
61401F02 shr    ecx, 2
```

```
61401F05 test   dl, 2
```

```
61401F08 cld
```

```
61401F09 rep    movsd
```

```
61401F0B mov    ebx, edi
```

```
61401F0D jz     short loc_61401F1B
```
- Bottom snippet (instruction 61401F0F):

```
61401F0F movzx  edi, word ptr [esi]
```

```
61401F12 add    esi, 2
```

Annotations:

- A green box surrounds the first snippet (61401EEC).
- Three green arrows point from the end of the first snippet to the start of the middle snippet (61401F00).
- A red arrow points from the end of the middle snippet to the start of the bottom snippet (61401F0F).

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.

```
1EE3  mov      ecx, ss:[esp+var_90]
1EE7  or       ecx, edx
1EE9  cmp      ebp, bl 8
1EEC  mov      ds:[esi+edi+4], ecx
1EF0  ja       0x61401D00
```

```

61401ED7 shld    edx, eax, 8
61401EDB shl     eax, 8
61401EDE or      ecx, eax
61401EE0 mov     [esi+edi], ecx
61401EE3 mov     ecx, [esp+10Ch+var_90]
61401EE7 or      ecx, edx
61401EE9 cmp     ebp, 8
61401EEC mov     [esi+edi+4], ecx
61401EF0 ja     loc_61401D00

```



```

61401EF6 mov     esi, [esp+10Ch+var_5C]
61401efd lea     eax, [esp+10Ch+buffer]
61401F04 mov     [esp+8], ebp
61401F08 mov     [esp+4], eax
61401F0C mov     ecx, [esi+3Ch]
61401F0F lea     esi, [esp+10Ch+buffer]
61401F16 mov     [esp], ecx
61401F19 call    stream_Read
61401F1E mov     ebx, [esp+10Ch+var_58]
61401F25 mov     ecx, edi

```

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 dwds, 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`.

The screenshot shows the OllyDbg debugger interface. The assembly pane displays the following code:

```

402142 5B      POP EBX
402143 5E      POP ESI
402144 5F      POP EDI
402145 5D      POP EBP
402146 C3    RETN
402147 89F6    MOU ESI,ESI
402149 8DBC27 00000000 LEA EDI,DWORD PTR DS:[EDI]
402150 83EC 3C SUB ESP,3C
402153 BA D9914061 MOU EDX,libty_pl.61409109
402158 897C24 34 MOU DWORD PTR SS:[ESP+34],EDI
40215C 89C7    MOU EDI,EAX
40215E 896C24 38 MOU DWORD PTR SS:[ESP+38],EBP
402162 BD 03000000 MOU EBP,3
402167 895C24 2C MOU DWORD PTR SS:[ESP+2C],EBX
40216B 897424 30 MOU DWORD PTR SS:[ESP+30],ESI
40216F 8870 58 MOU ESI,DWORD PTR DS:[EAX+58]
402172 B8 C8904061 MOU EAX,libty_pl.614090C8
402177 888E 6CBE0000 MOU ECX,DWORD PTR DS:[ESI+BE6C]
40217D 894424 08 MOU DWORD PTR SS:[ESP+8],EAX
402181 895424 0C MOU DWORD PTR SS:[ESP+C],EDX
402185 896C24 04 MOU DWORD PTR SS:[ESP+4],EBP
402189 894C24 10 MOU DWORD PTR SS:[ESP+10],ECX
40218D 893C24    MOU DWORD PTR SS:[ESP],EDI
402190 E8 024D0000 CALL <JMP.&libvlccore._msg_Generic>
402195 8886 70BE0000 MOU EAX,DWORD PTR DS:[ESI+BE70]
40219B 85C0    TEST EAX,EAX

```

The registers pane shows:

Registers (F)
EAX 00000000
ECX 002C3A1C
EDX 00000000
EBX 3983B0B8
ESP 0264FB04
EBP E6003EBF
ESI A1F6C700
EDI 3CD7003B
EIP 61402146
C 0 ES 0023
P 0 CS 001B
A 1 SS 0023
Z 0 DS 0023
S 0 FS 003B
T 0 GS 0000
D 0
0 0 LastErr
EFL 00000212
ST0 empty -1
ST1 empty 0..
ST2 empty 0..
ST3 empty 0..
ST4 empty 0..
ST5 empty 0..
ST6 empty 52..

The memory dump pane shows the byte sequence at address 0x44434241:

dress	Hex dump	ASCII
402000	01 00 00 00 10 1A 40 00	...@.
402008	00 00 00 00 00 00 00 00

There, I tested it on an OllyDbg on XP that I have for testing, but it works. This jumps to execute the address **0x44434241** 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