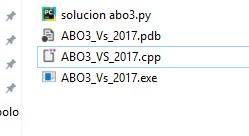
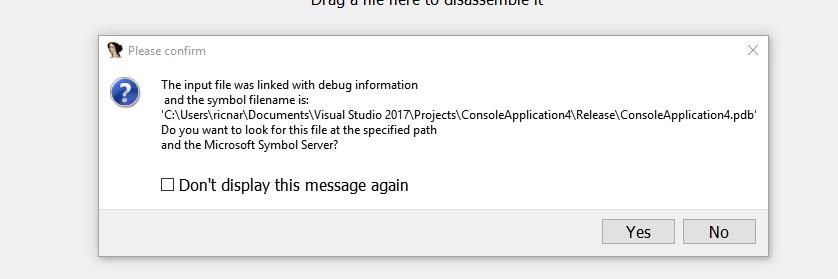
# EXPLOITING AND REVERSING USING FREE TOOLS (PART 8)

We have solved ABO2 in GHIDRA and now it is our turn to use IDA FREE to solve ABO3.

As in all ABOS the point is to run the calculator or some executable we want.



We open ABO3 in IDA FREE to analyze it.



Looking for the PDB with the symbols could show the following error.



## MS DIA SDK ERROR

In case you get this error you need to install the Visual C++ 2008 redistributable for 64 bits.

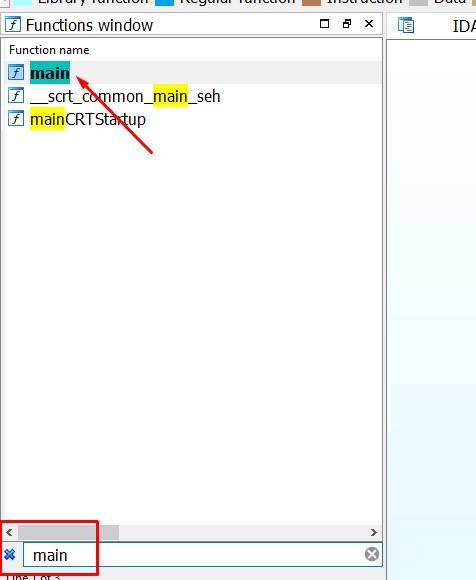
INSTALLING VISUAL C++ 2008 REDISTRIBUTABLE FOR 64 BITS.

The package can be downloaded from here.

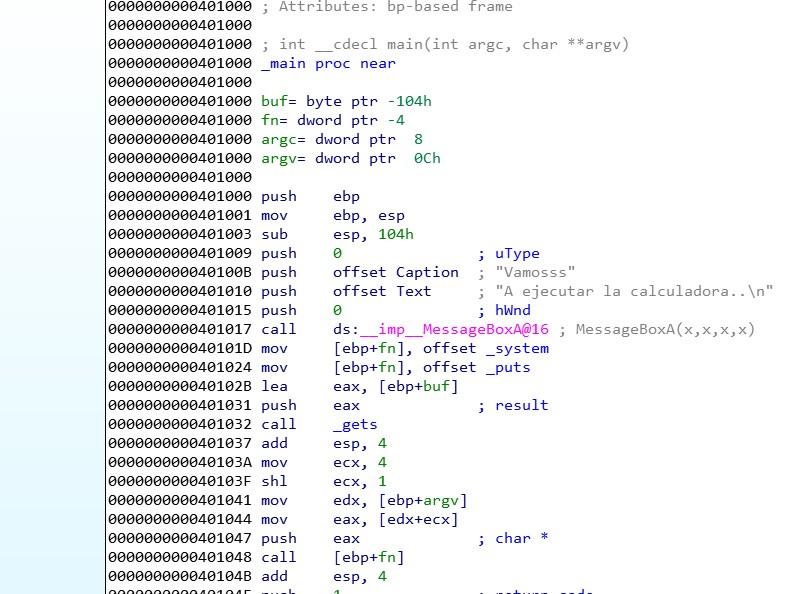
<https://www.microsoft.com/en>[-us/Download/confirmation.aspx?id=15336](https://www.microsoft.com/en-us/Download/confirmation.aspx?id=15336)

Once installed and restarted IDA we load ABO3.

When it asks us for the symbols, we open them and then in the function window we press CTRL + F and type **main** to search.

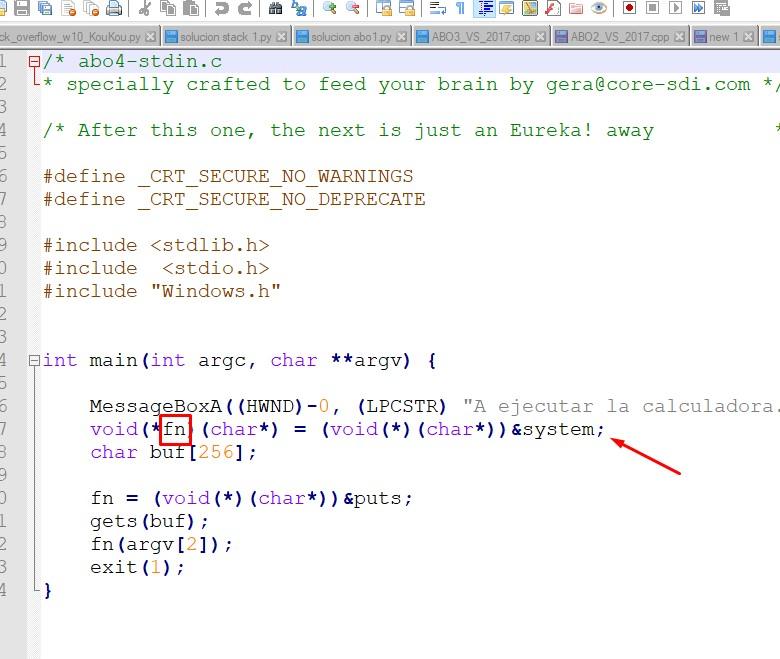


Go there with a double click on the result found.



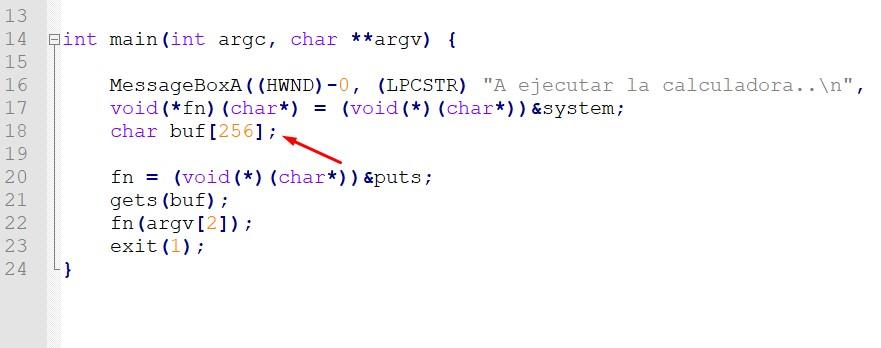
In IDA FREE we don't have a decompiler, only the PRO version has it, maybe the source code is in the CPP file and if we don't have the CPP, we can decompile it with the x64dbg.

## SOURCE CODE



We see an fn variable in the stack of the pointer to function type, it is initialized with the address of the **system()** function, which is used to execute a console command, if we pass it the name of an executable, if it exists and it can find it, it will execute it.

There is a 256 bytes buffer called **buf**.



When the program starts, the **fn** variable changes to save the address of the function **puts()**, which is used to print a text in the console.

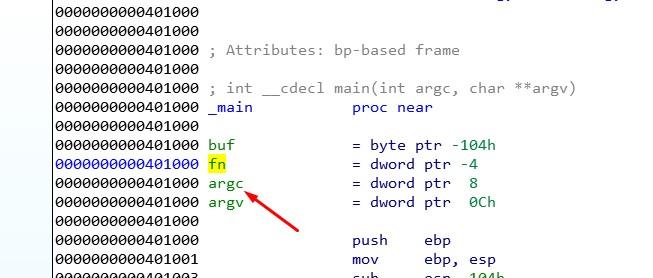
Then it calls **gets()** to fill the buffer called **buf** and finally it calls to execute the function whose address is saved in **fn**, which if there is no overflow will be **puts()**, it will print the text we entered as the second argument (**argv[2]**) and it would leave the main without ever reaching the return address.

Well, the idea is that as the data entry is by means of **gets()** we can overflow **buf** and step the value of the variable **fn**, so when we execute in **fn(argv[2])**, we can jump where we want.

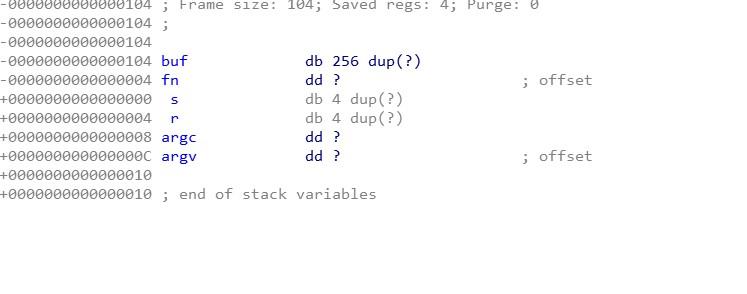
Now let's do the static reversing in IDA as if we had not seen the source code.

## STATIC ANALYSIS OF ABO3

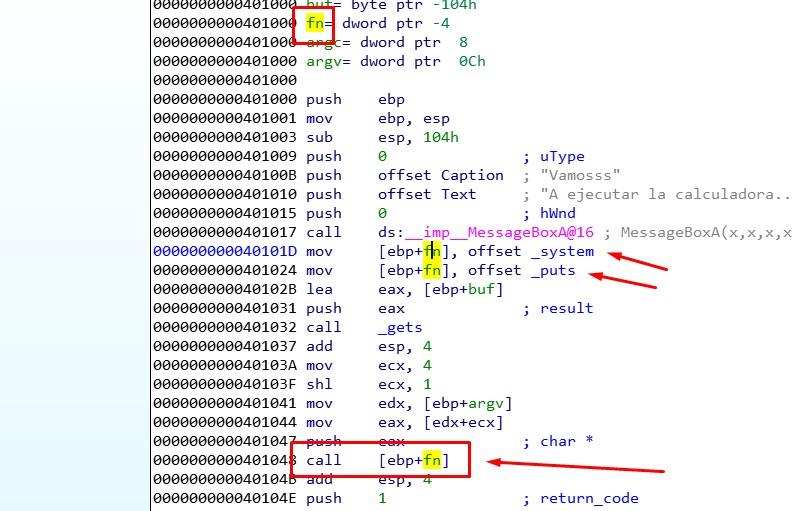
By double-clicking on any variable in the function definition



We see the **static representation of the stack**.



There we see **buf**, that is a byte array variable (db= BYTES) whose length is 256 bytes and below it is **fn**, that as it is a 32 bits pointer variable its length is 4 bytes or (dd= DWORD)



There we see all the accesses to the variable **fn**. First we save the direction of the **system** function, in the next line we save the direction of **puts** and finally we jump to execute the function whose value is saved in **fn**, if there is no overflow it will jump to **puts**.

And what will be the **puts** argument?

**argv** is an array of pointers, each one pointing to a console argument.

**argv[0]** points to the name of the executable.

**argv[1]** points to the first argument.

**argv[2]** points to the second argument and so on.



If I run it from a console with those arguments.

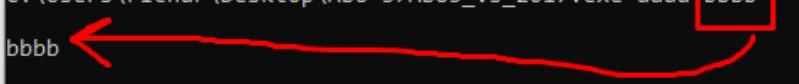
### ABO3\_Vs\_2017.exe aaaa bbbb

**argv[0]** points to the name of the executable =ABO3\_Vs\_2017.exe

**argv[1]** points to the first argument = aaaa

**argv[2]** points to the second argument= **bbbb**

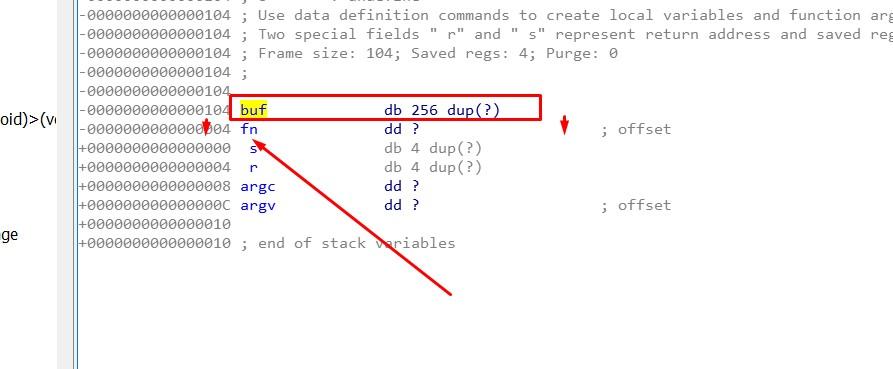
Therefore, the argument that is passed to the function **puts** is controlled by us, we see that finally after pressing ENTER, to pass the pause of the program, it prints **bbbb** or the argv **[2].**



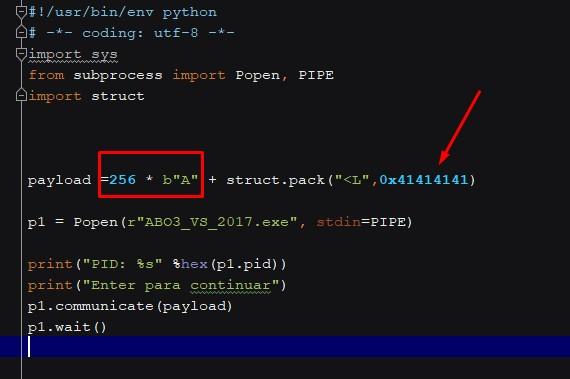
So, the argument, whatever function is stored in **fn**, will be the **argv[2]** and will be controlled by us.

If we were to make a script.

As the idea is to fill **buf** and overflow it to overwrite **fn**.

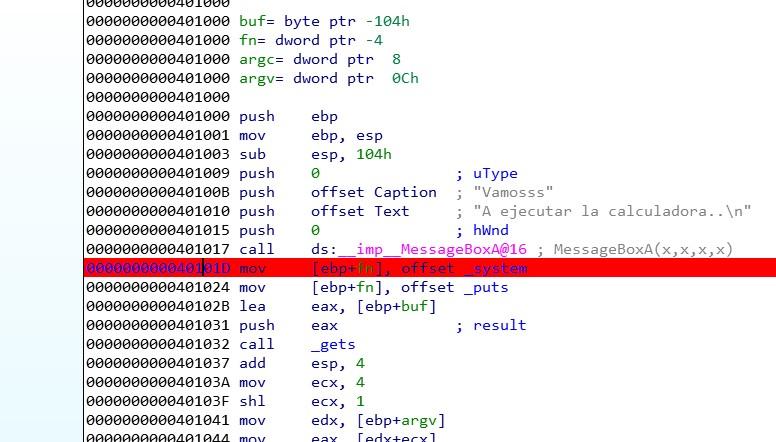


We will have to send 256 bytes to fill **buf** and then the 4 bytes that will step on **fn**.

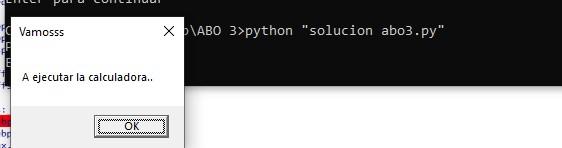


That would jump to execute at 0x41414141, let's try.

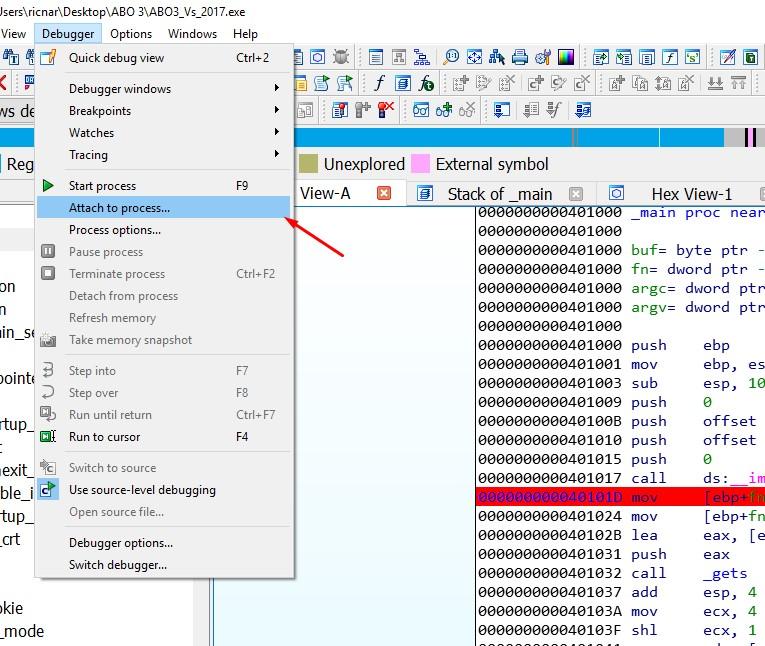
Let's put a breakpoint when returning from the MessageBoxA

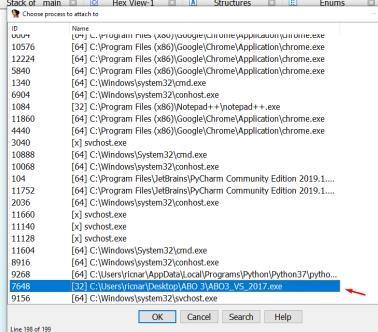


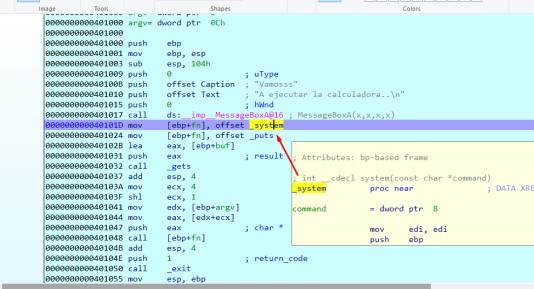
Let's run the script from a console.



Let's attach the IDA.

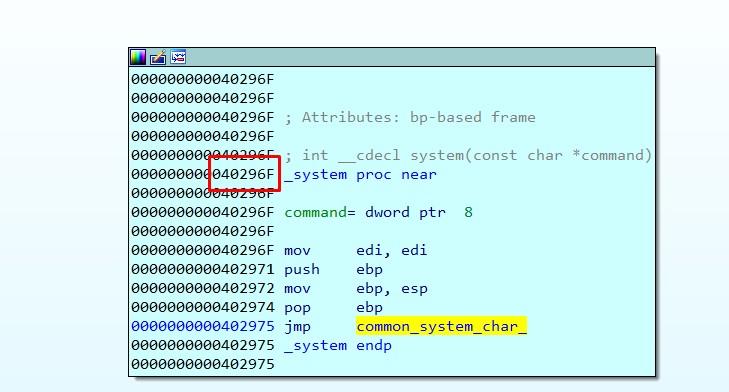




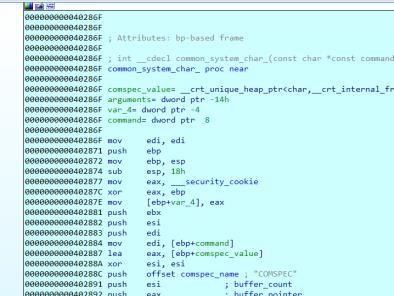


Once the message is accepted it stops and we see how it's going to save the **system** address in **fn**.

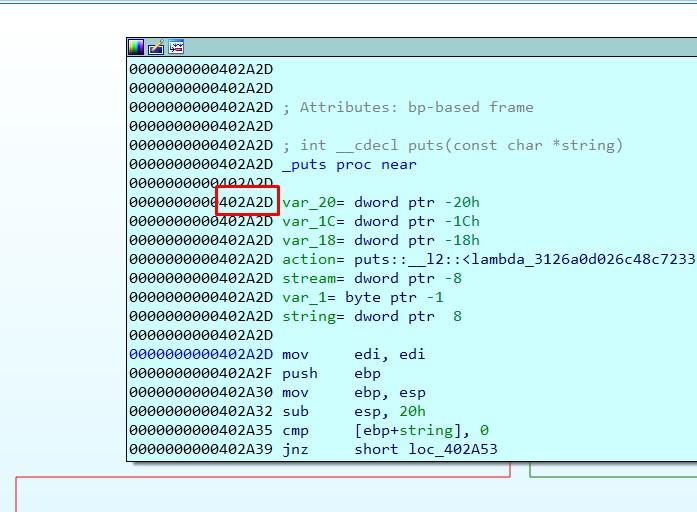
If we double click on **system,** it will take us to the address of the function.



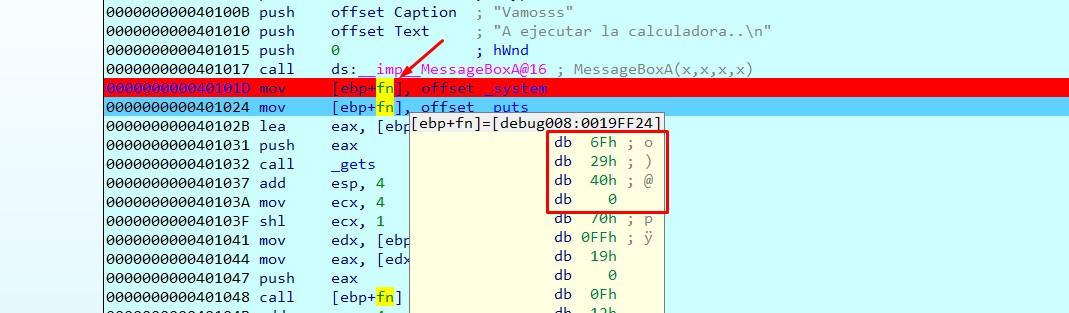
As the executable is compiled with the embedded DLLs, **system** is inside the same executable, in this case it does not belong to a dll.



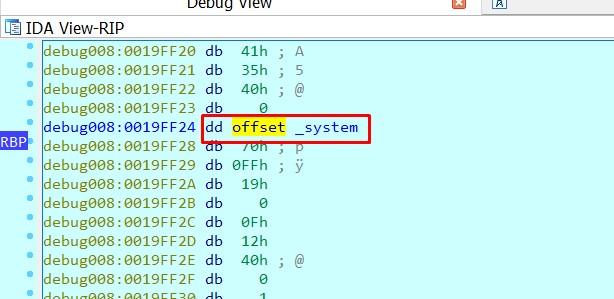
So, when I compile it I keep a copy of the **system** API in the same executable, the same happens with **puts**.



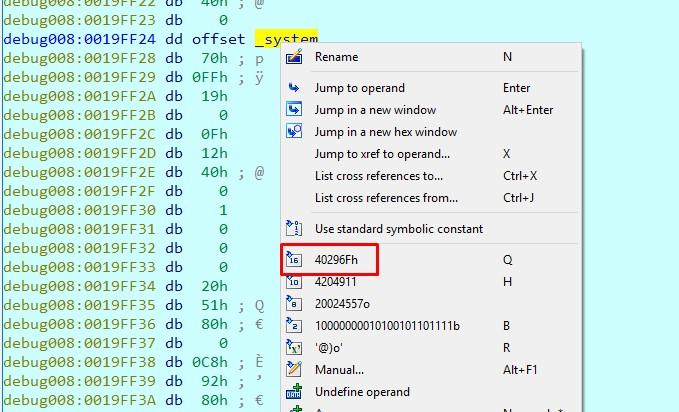
If we press f8 and pass the mouse over **fn**, I see that I save the **system()** address.



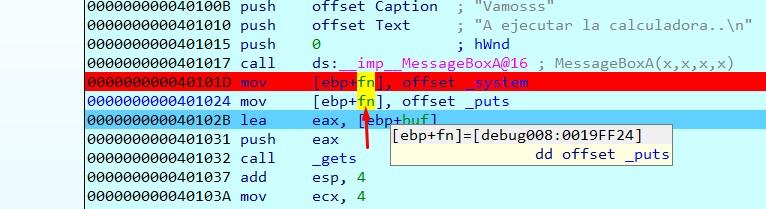
If I double-click on fn, I can set it to be a DWORD, press the D key until it becomes a DWORD and it will show **system**.



There we see the system address with a right-click.



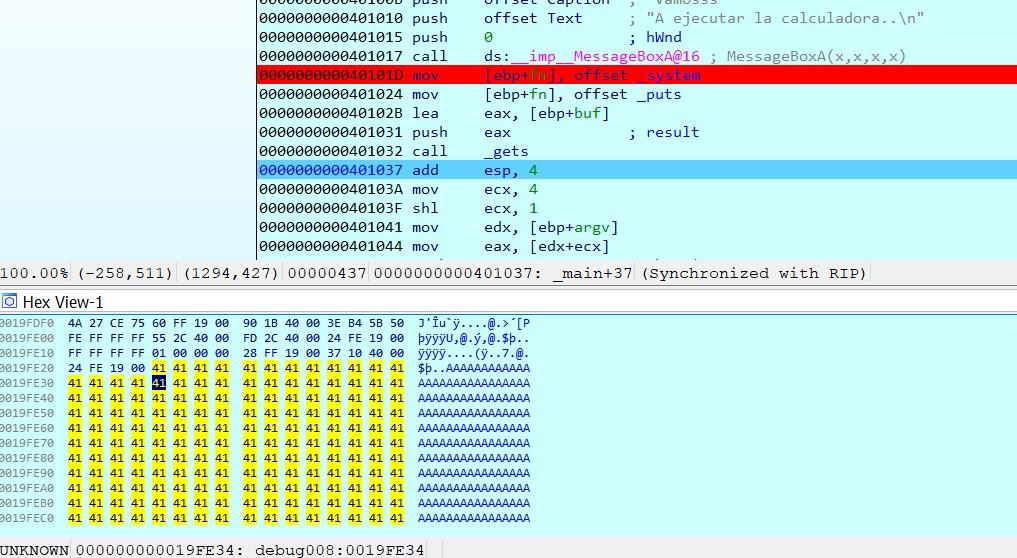
I press f8 again and save the **puts** address in **fn,** passing the mouse over it I will see it as DWORD, because the memory position where **fn** is was already changed.



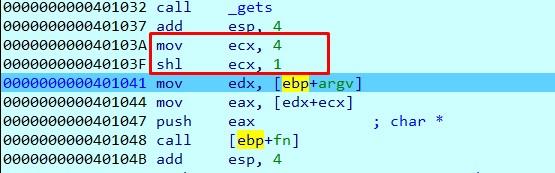
If I continue tracing, I will get the **gets()**.



EAX will have the address of **buf** and will fill it up when passing over the **gets()**.



I keep tracing.

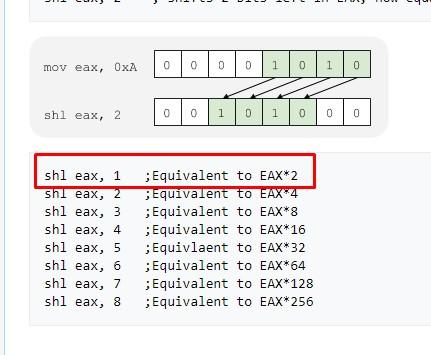


I move 4 to ECX and then

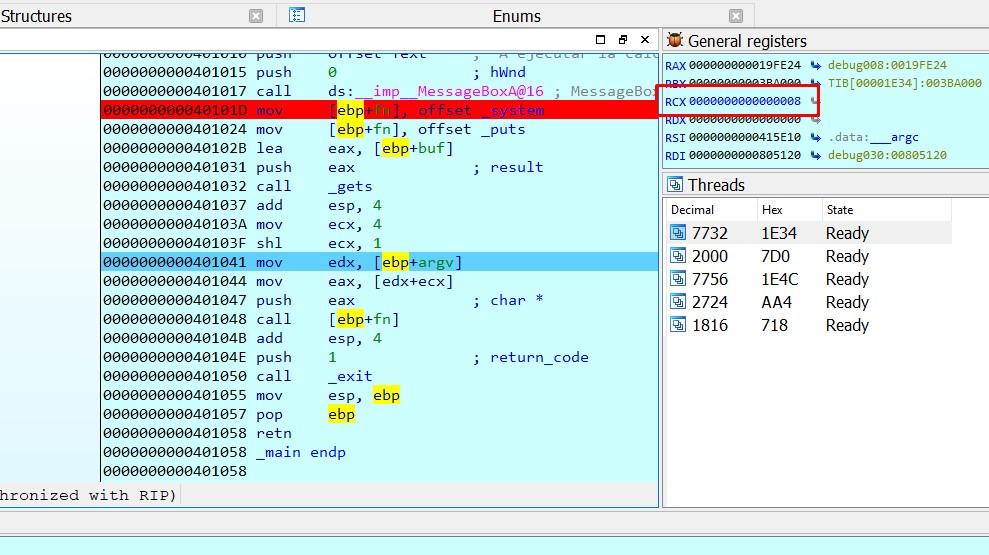
### SHL ECX, 1

<https://www.aldeid.com/wiki/X86-assembly/Instructions/shl>

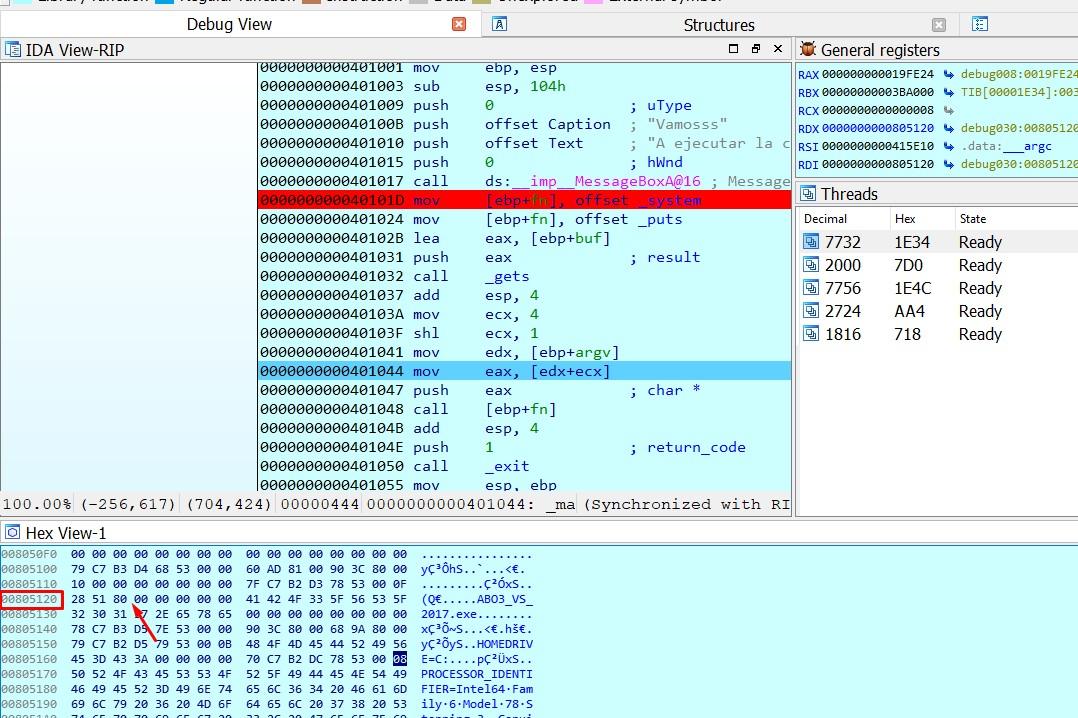
Which is equivalent to multiplying ECX by 2.



So ECX will be worth 8 after multiplication by 2.

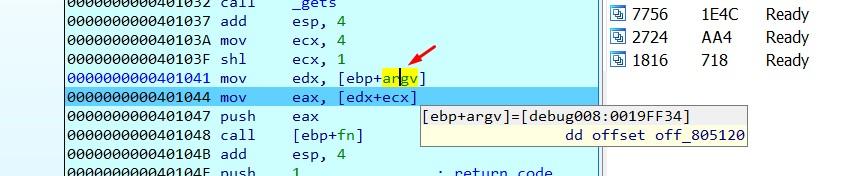


**argv** is moved to EDX; we already saw that it is an array of pointers so each field will be 4 bytes.



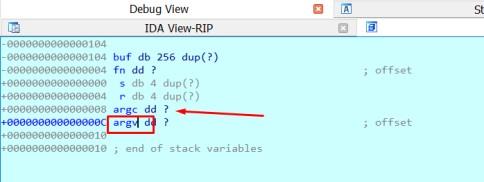
As we don't pass arguments **argv** only has one field, which is a pointer to the name of the executable.

The number of **argv** fields matches the value of **argc**, which is the number of arguments passed.

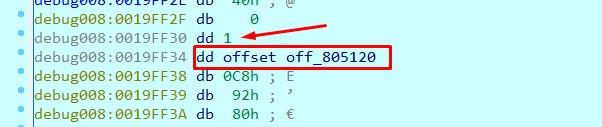


As we know both are also arguments of the **main** function, **argc** is above **argv** because of the order in which the arguments were passed in the stack when **main** was called

If we double click on argv in the function definition, we will go to the **static representation of the stack** and we will see above **argv**, the variable **argc**.

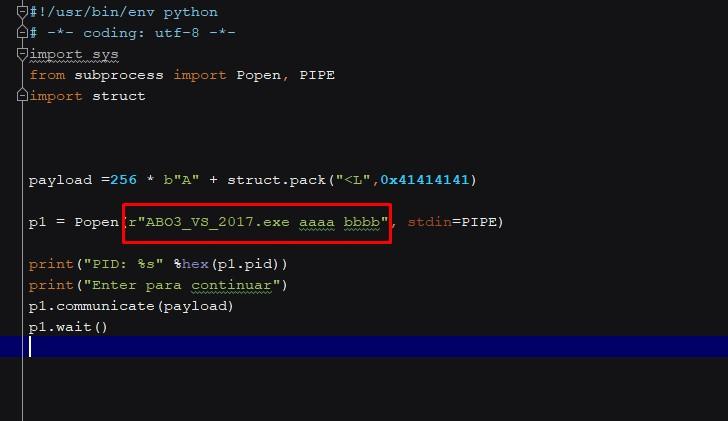


If I double click on the **argv** variable, but the code takes me to the memory address where it is when I'm debugging.



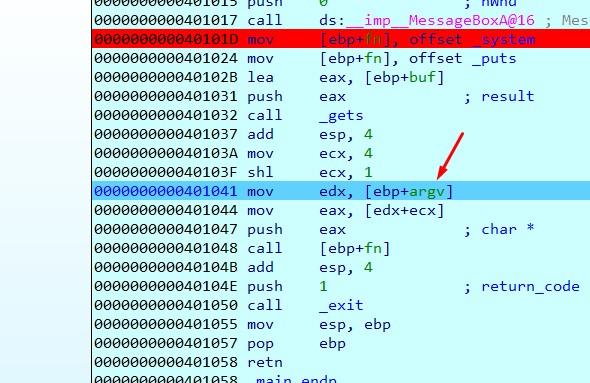
and above it will be **argc** which is 1.

So, I have to pass the arguments in the script and launch it again.

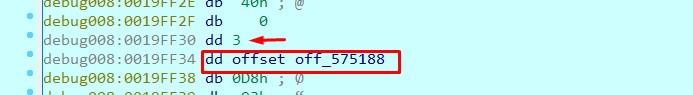


The console arguments are passed after the name of the executable, separated by spaces.

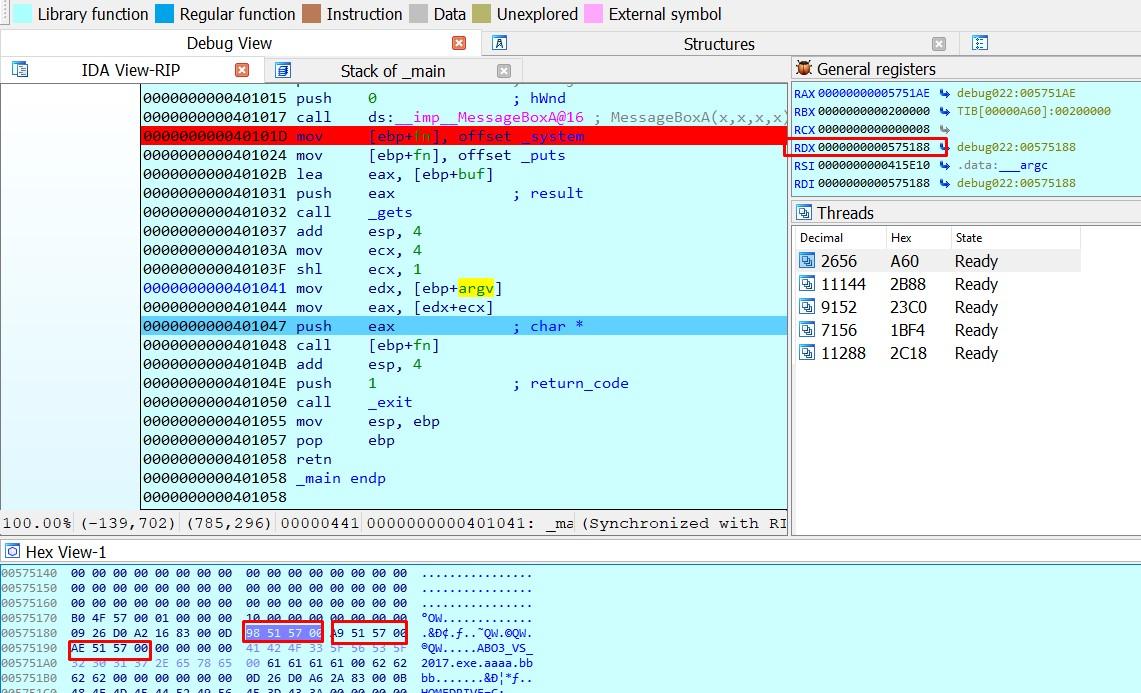
Let's run it again and get to the same point where we were.



I double click on **argv** in the code.



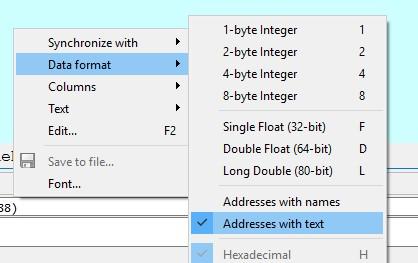
I already transformed them to DWORD by pressing D several times and I see that **argc** is 3 because I passed three arguments, so now **argv** is an array of three-pointers to each argument string.



If in the HEX VIEW I click RIGHT - SYNCHRONIZE WITH RDX.



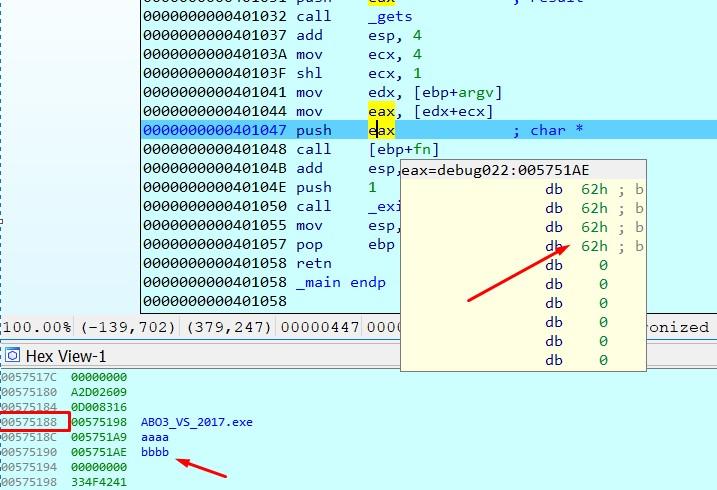
And then I change it to DATA FORMAT -ADDRESSES WITH TXT.



I can see the array of pointers to strings, each pointer points to an argument, being 0, the name of the executable, 1 the first argument **aaaa** and 2 the second argument **bbbb**.

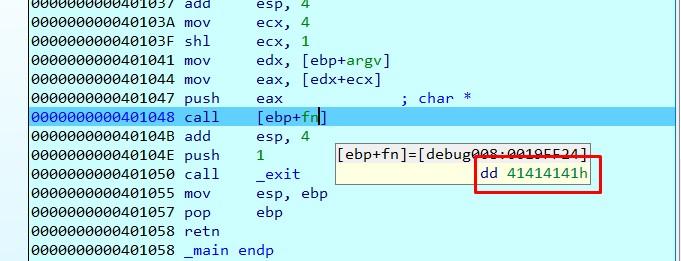


The point is that adding 8 to argv will move EAX the pointer to the second argument **bbbb**.



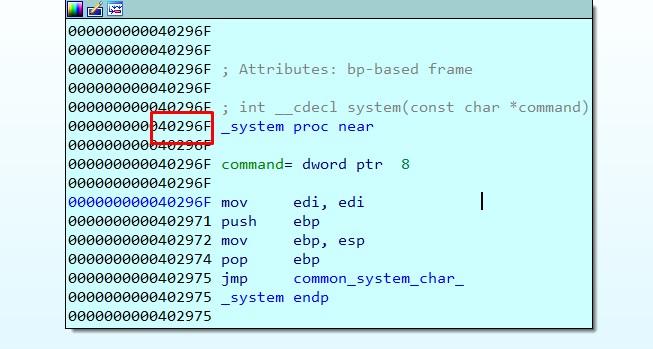
And that **bbbb** will be the argument of the function we'll jump to.

Whose address will be 0x41414141 if I pass it to DWORD.



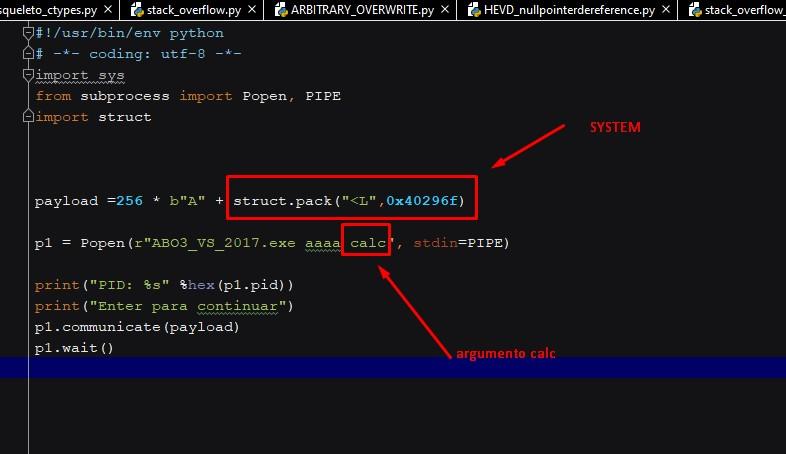
So, I jump where I want and control the argument, if instead of stepping on fn with

0x41414141, I overwrite it with the system address which is fixed and known because as we saw it is embedded in the executable.

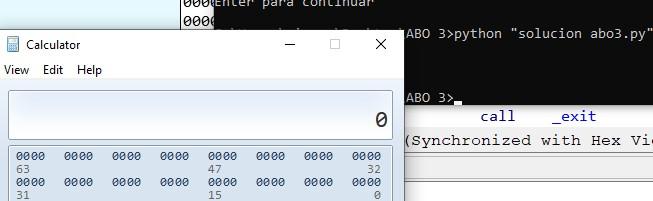


# SCRIPT SOLUTION

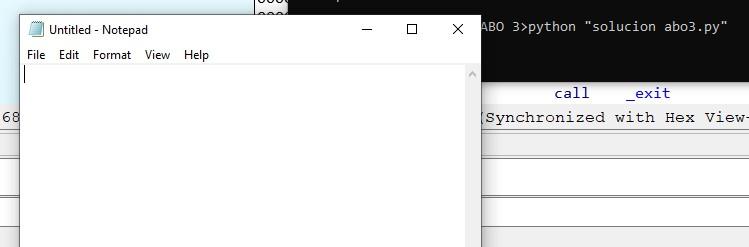
And instead of the **bbbb** argument I put the **calc** string to run the calculator.



If I execute the script the calculator will be executed

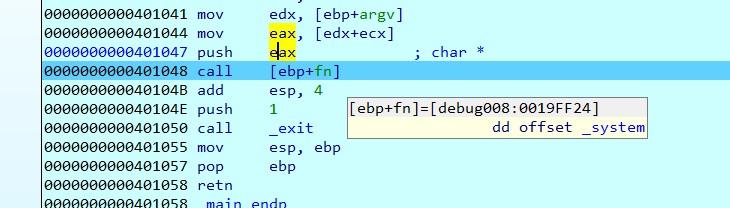


I can also run another executable file, for example notepad.

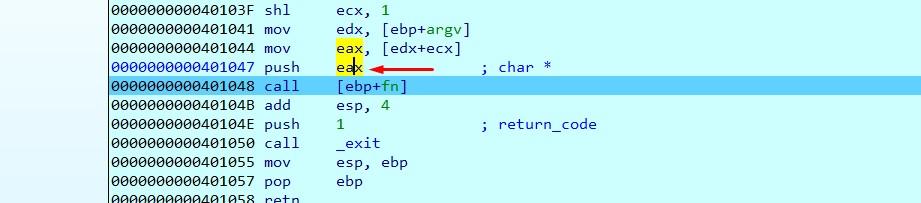


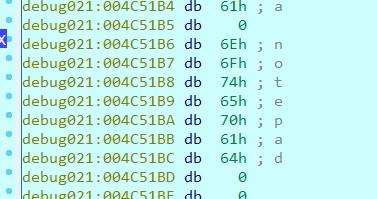
If I attach the debugger and stop at the jump to run.

Once I pass it to DWORD I see that it runs system



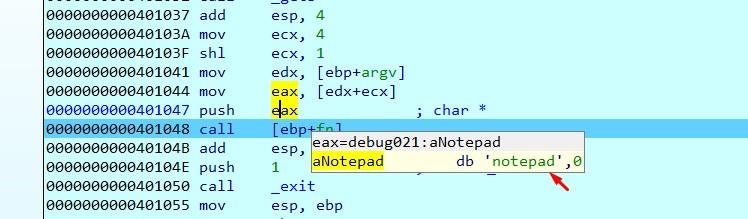
And if I double-click there on EAX





I can turn it into a string with the A key.





Well, we have solved the ABO3 in IDA FREE and the next part we will solve the ABO4 in radare.

See you in the 9th part.

Ricardo Narvaja

1/31/2020