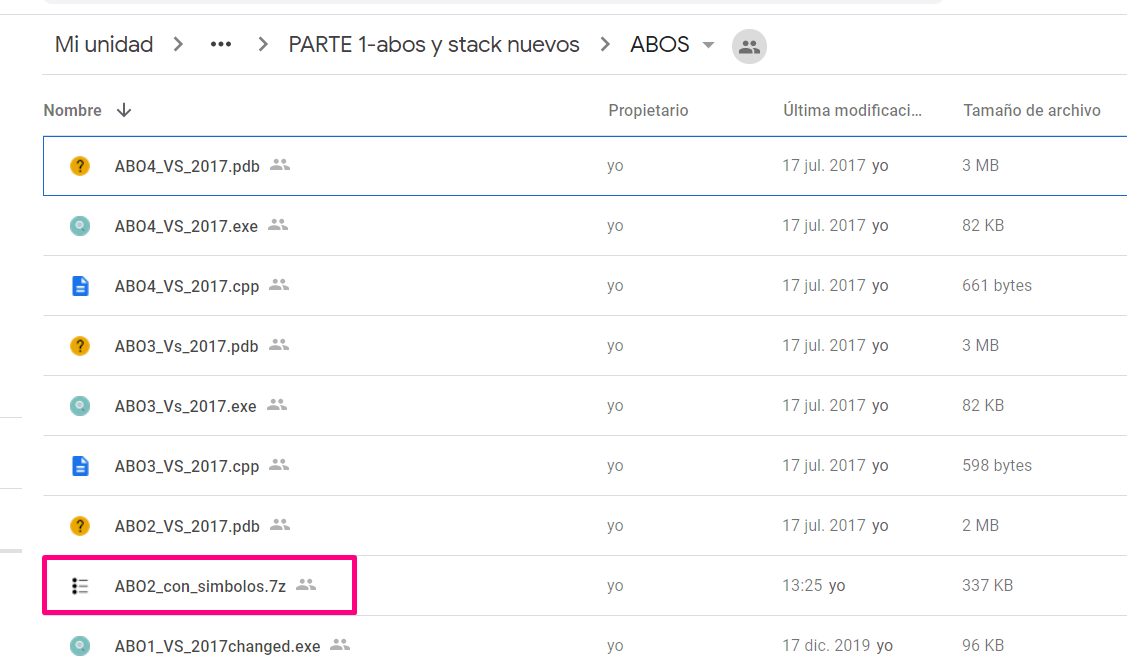
REVERSING & EXPLOITING USING FREE TOOLS (PART 7)

Well once we’ve seen and understood a shellcode and its resolver that it’s the most complex part, we will continue with the analysis and resolution of abo2 in GHIDRA.

Download ABO2 executable, I uploaded a new version to google drive, because the old one didn’t have the correct symbols, so now it works properly.



There, inside of the 7zip compressed file the executable (with .exe extension) and the symbols (with .pdb extension) and the source code (with .c extension).

<https://drive.google.com/file/d/1ka2_0V51vew_ZjZ0hprHxFmJPCCL8KO7/view?usp=drivesdk>



As the new ghidra version 9.1.1 was released I will download it to update it.

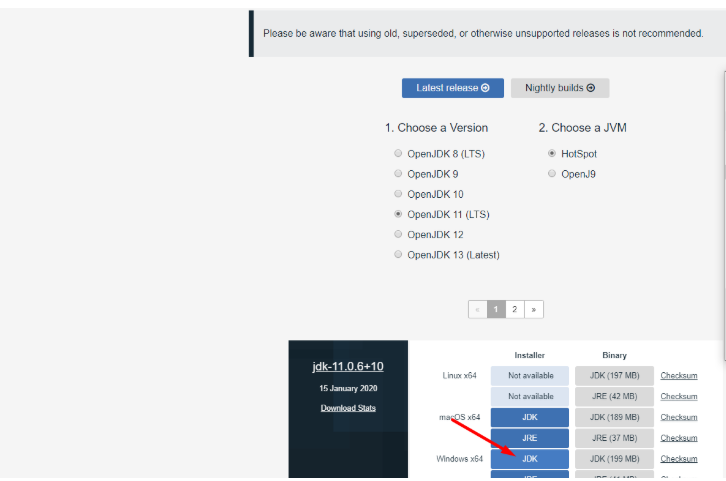
<https://ghidra-sre.org/ghidra_9.1.1_PUBLIC_20191218.zip>

Of course, if in the moment that you read this training there’s a newer version the link will change, so you can visit this url:

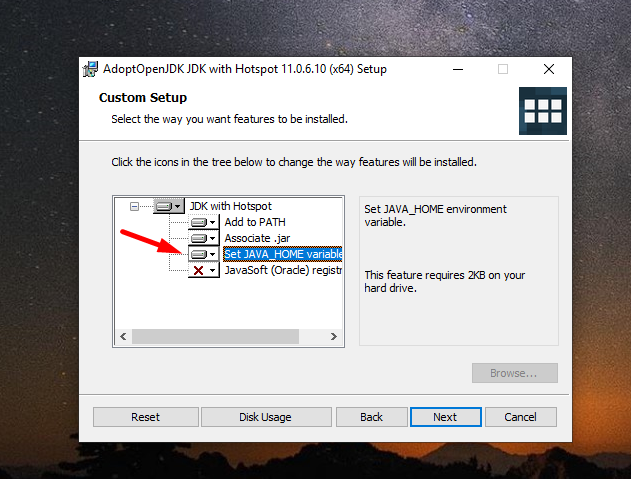
<https://ghidra-sre.org/>

The version 11 of the JDK can be downloaded from here:

<https://adoptopenjdk.net/archive.html?variant=openjdk11&jvmVariant=hotspot>



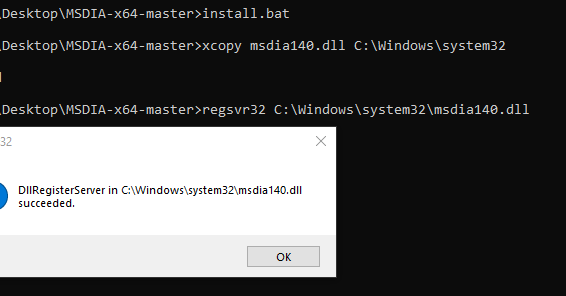
From the webpage you download the MSI file, install it, be sure you check the option to add binary to PATH environment variable.



If you didn’t have it, add the dia sdk.

<https://github.com/MalwareTech/MSDIA-x64>

Execute in a CMD as administrator the bat file.



Bat file only copies the file msdia140.dll from the compressed file we downloaded to system32.

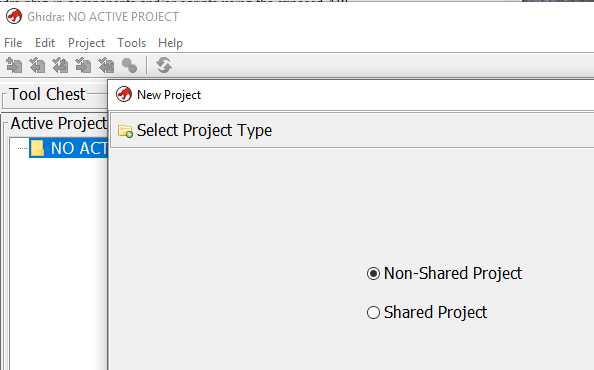
*xcopy msdia140.dll %systemroot%\system32*

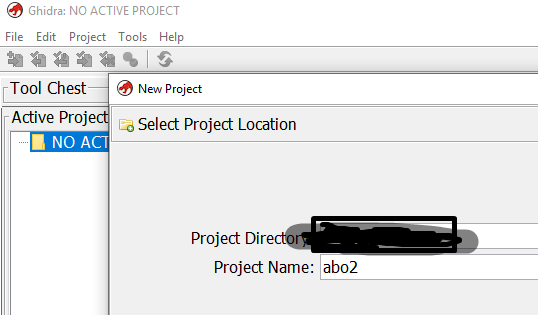
And then it registers it:

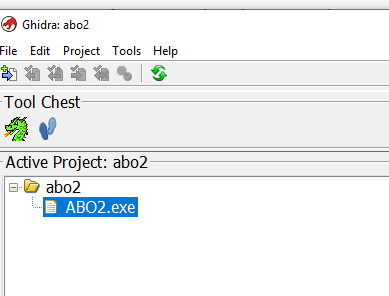
*regsvr32 %systemroot%\system32\msdia140.dll*

Also once the file is copied to system32 it can be also registered dragging the file msdia140.dll and droping it over the file regsvr32.exe in the same folder.

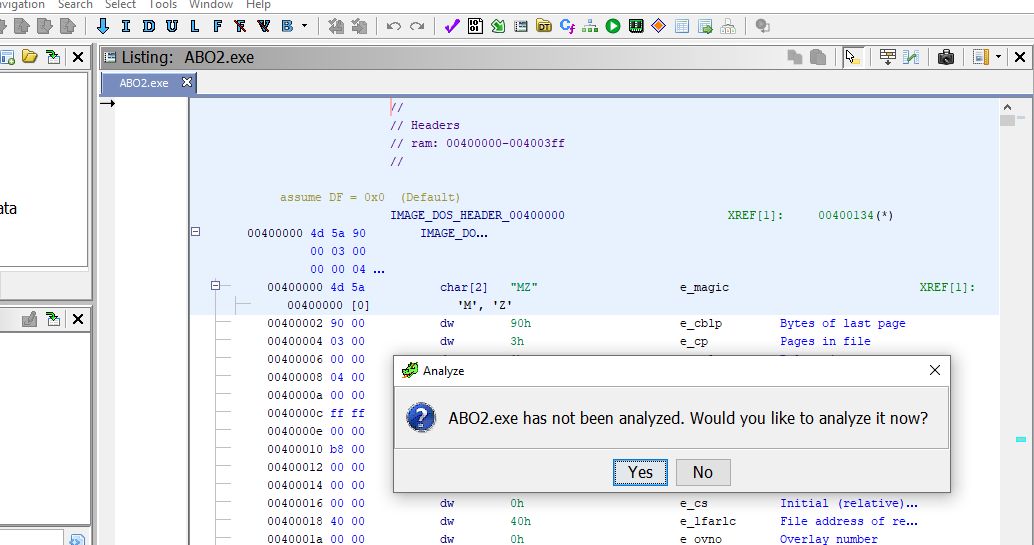
Well, now Ghidra runs without problems, I go to FILE-NEW PROJECT and I add new one.

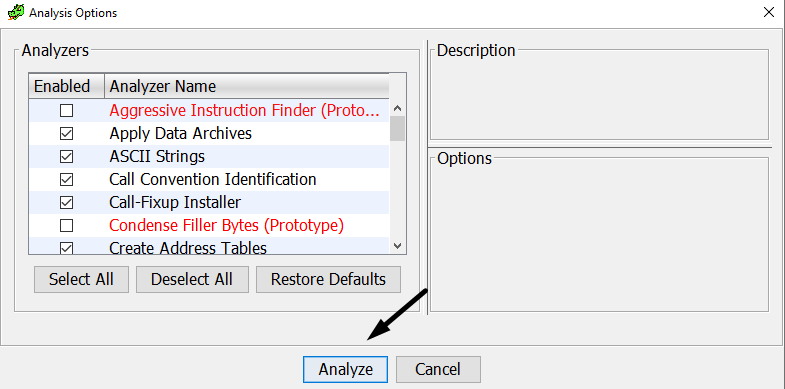




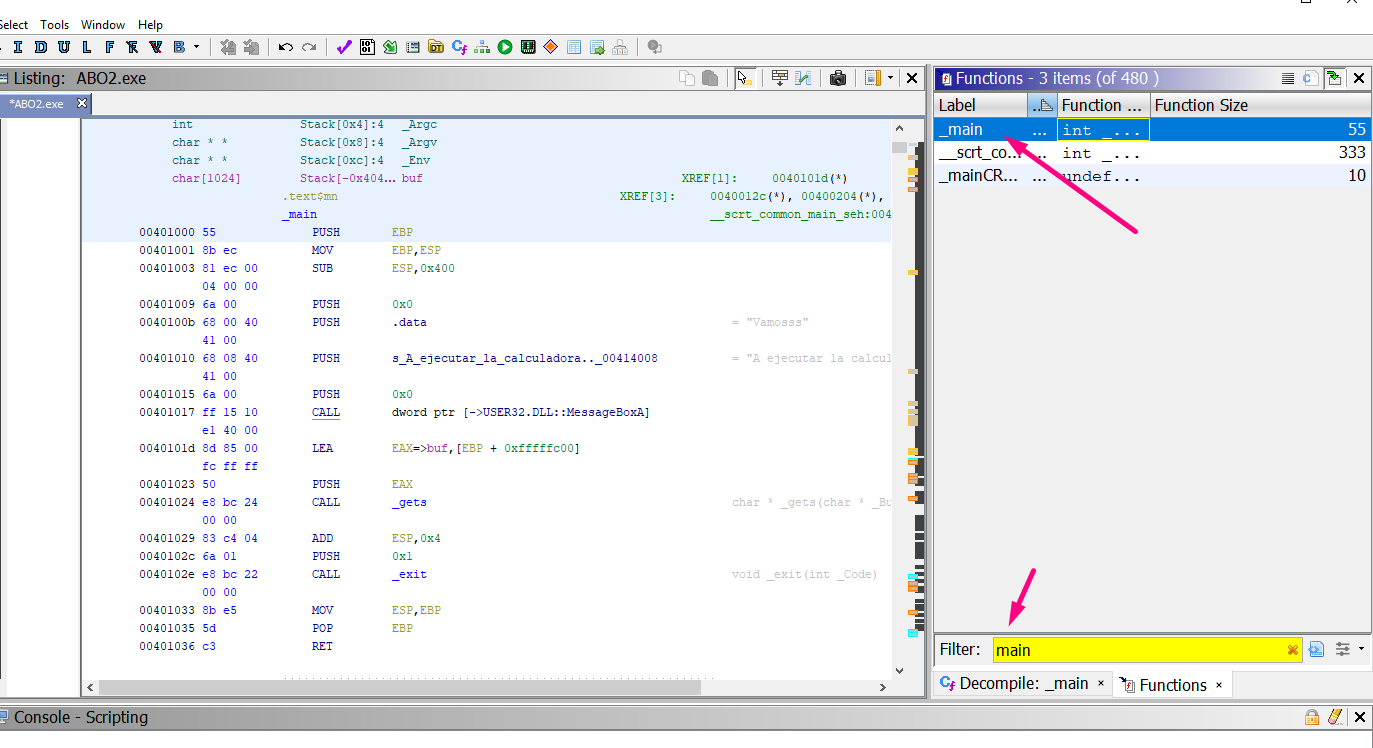


I double-click in the exe name.

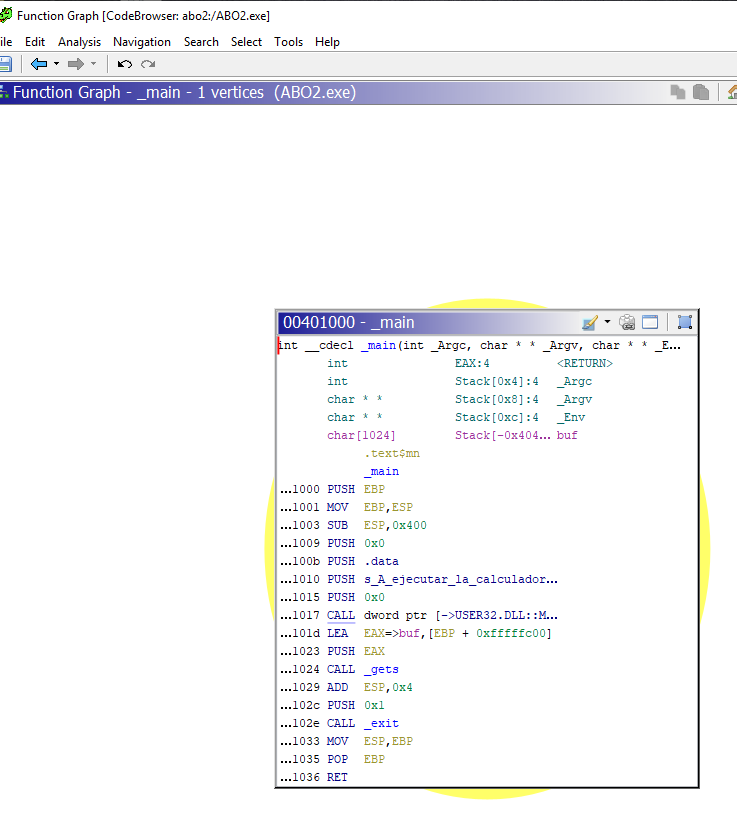




And going to the menu WINDOW-FUNCTIONS, I have a filter to type the function I want to search, in this case I will search main, in the function list I double-click in the main that appears.

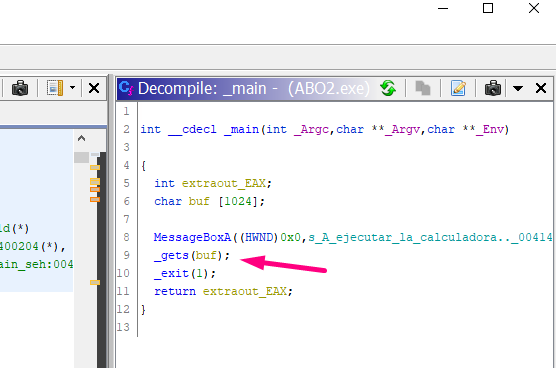


If we go to WINDOW-FUNCTION GRAPH, as it’s one block only there’s not too much to show.



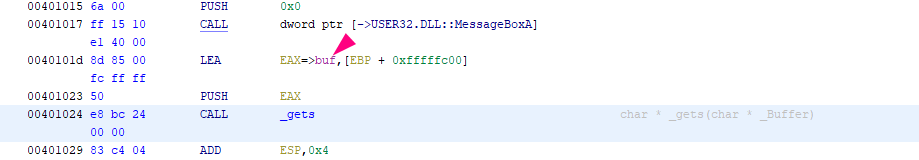
Here it looks similar to ABO1, but once we analyze it we will see that this is not true.

We can decompile with WINDOW-DECOMPILE or CTRL+E.

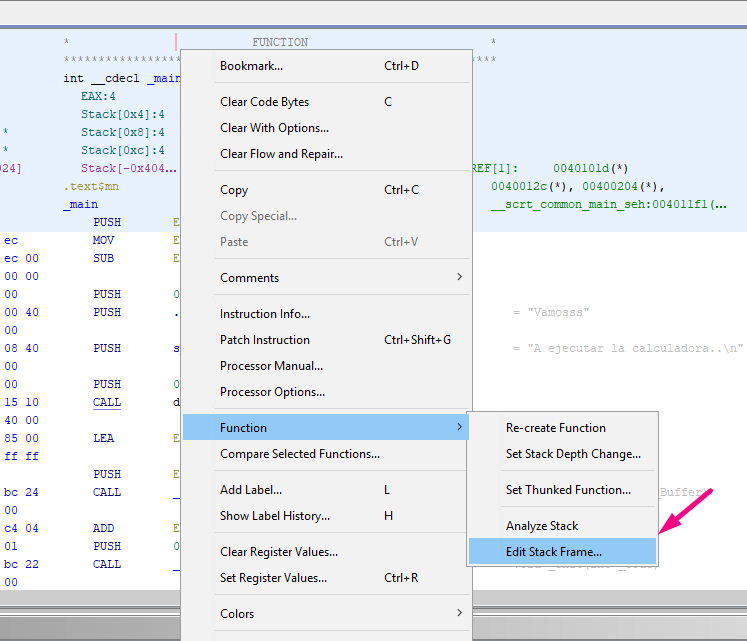


We see that there’s a buffer of 1024 bytes.

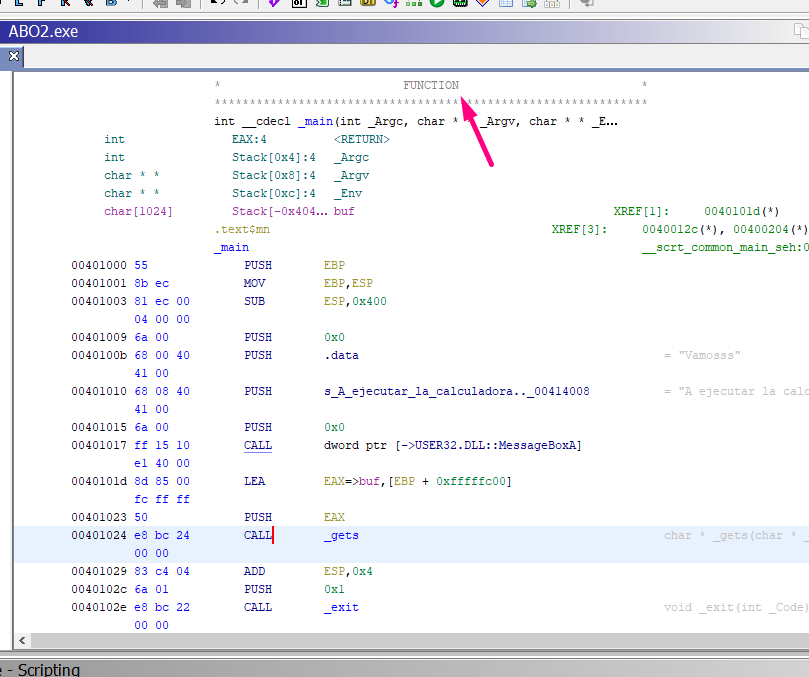
The address of buf is given to EAX, and then is pushed, so this will be the argument of gets function.



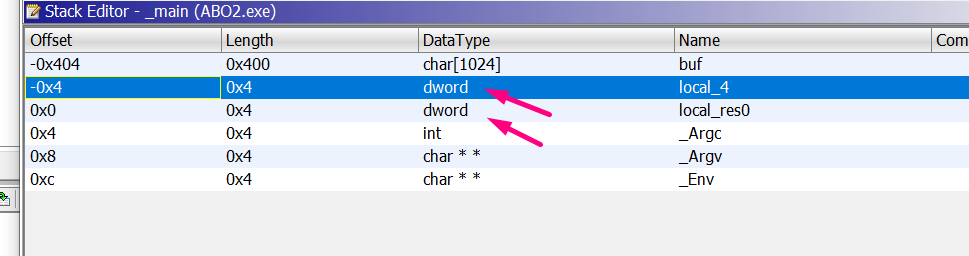
Of course that the buffer called buf will be filled, and can start overflowing with 1024 bytes.



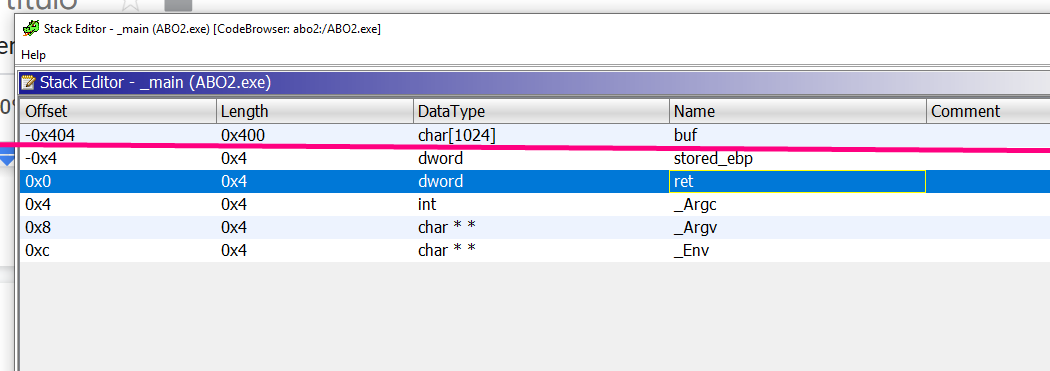
This menu appears right-clicking in the word FUNCTION.



Pressing the key B we can change data types, where stored ebp and return address should be stored for both to be DWORDS.

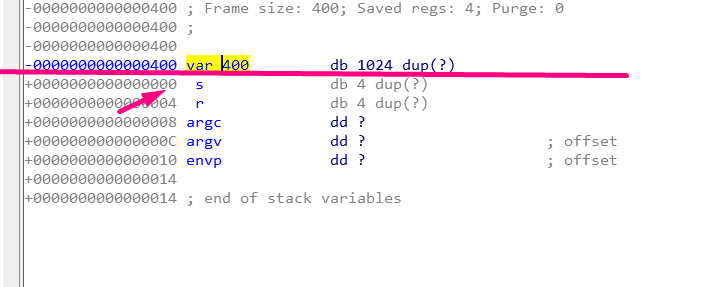


Remember, GHIDRA has a particular way to show stack variables.



While all functions based on EBP, that register is taken as reference so horizon should be in the read line and be 0, here because it takes as reference the ESP value at the beginning of the function, it’s moved 4 bytes from what we are used to work with IDA or others disassemblers, in GHIDRA the STORED EBP is in -4.

Here we can see it in IDA correctly, **s** that is STORED EBP is in 0 because there is the horizon (remember that with PUSH EBP, STORED EBP is saved in top of the stack, then the function’s EBP is set to the current value of ESP in that moment, in the picture is the red line in IDA, that is what we call the HORIZON because is taken as reference).



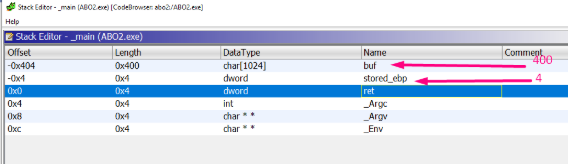
Any way, we can see that we have to send 0x404 bytes, and then the value to modify the return address.

In GHIDRA the buf’s offset is -0x404, and stored ebp is -4, so if we subtract one with the other, we can get that buf size is 0x400.

We add 4 bytes from stored ebp and then it would be the return address, so a possible payload would be:

**payload = b"A" \* 0x404 + struct.pack("<L",0x41414141)**

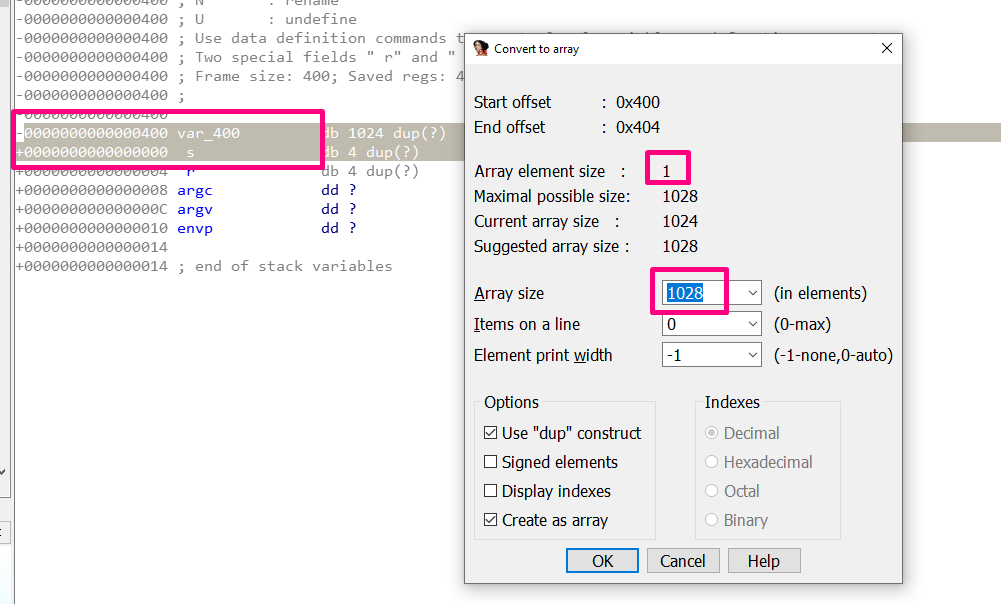
If we want to modify the return address with 0x41414141.



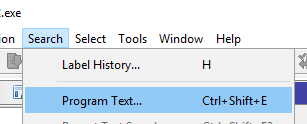
In IDA

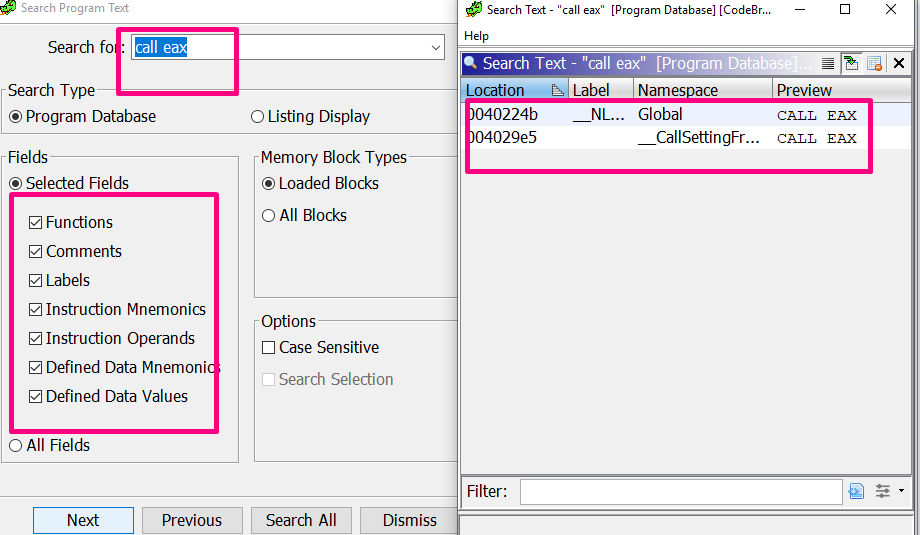
I select the area to fill, and then I RIGHT CLICK - ARRAY, and it shows us the size element of array 1 and length 1028.

So 1\*1028 = 1028 that it is 0x404 bytes in hexadecimal, then it’s the return address.

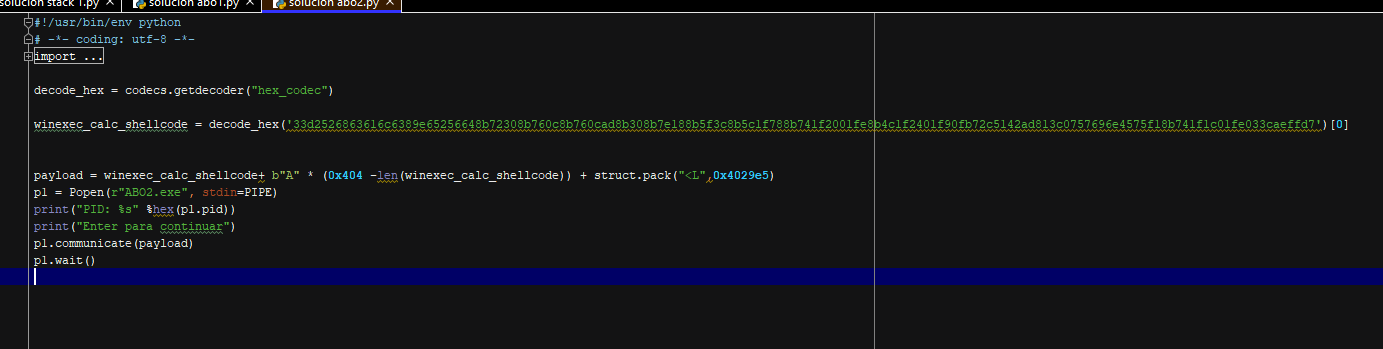


So apparently it would be similar to ABO1, we only need to find the CALL EAX to jump to execute the shellcode.





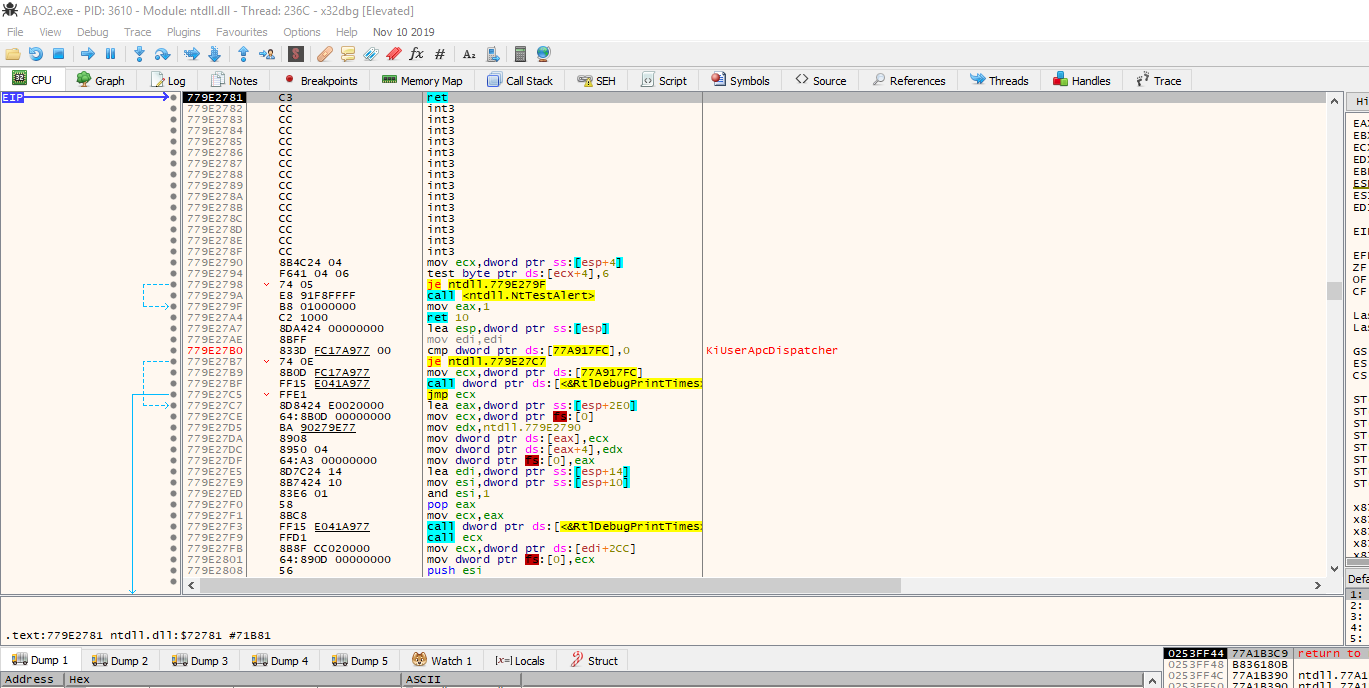
There I found a couple of CALL EAX search as text.

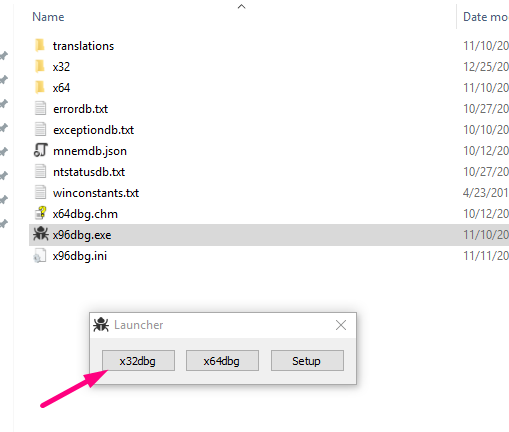


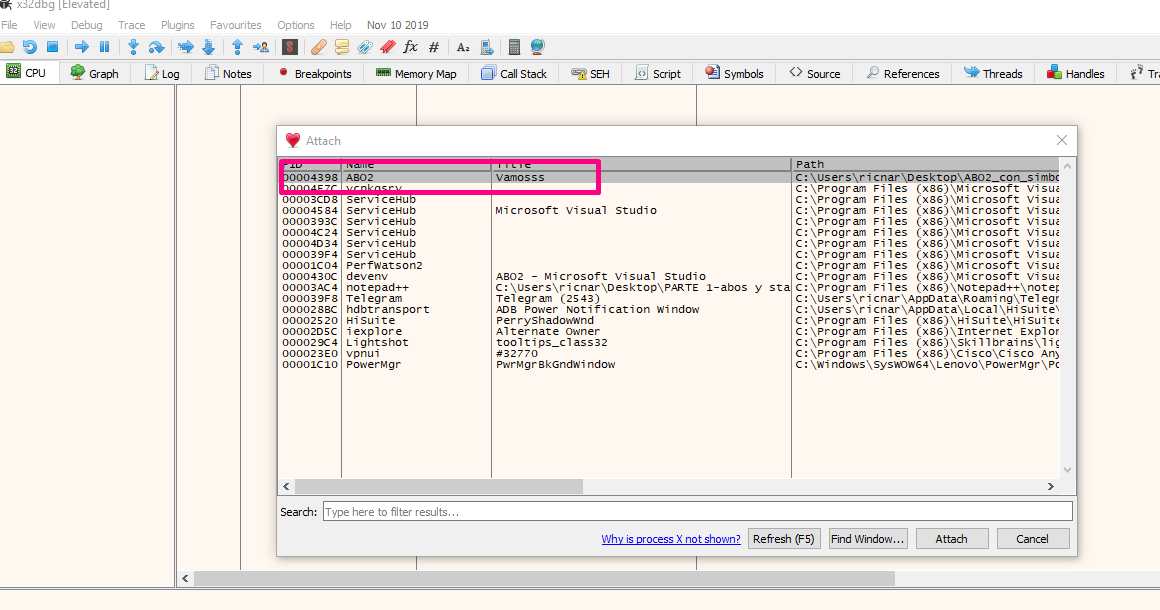
If I execute from a CMD I see that it doesn’t pop a calc, it only closes.

Let’s debug with x64dbg, so execute the .py from a console.

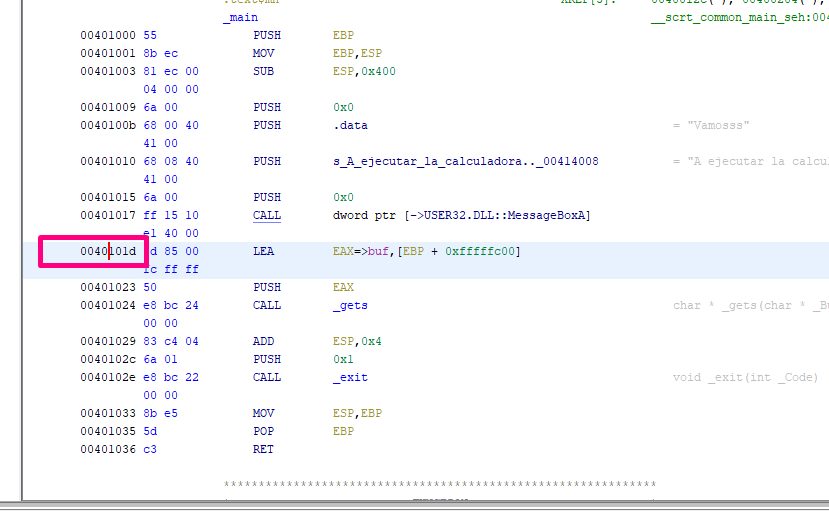
When the MessageBoxA appears, attach with x64dbg for 32 bits.



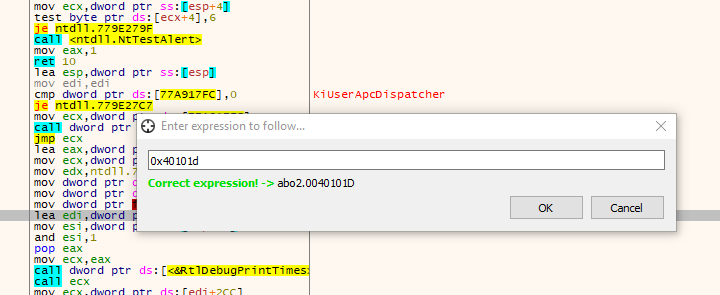




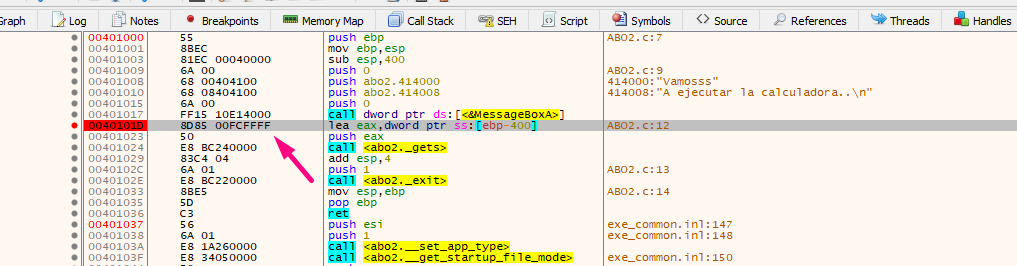
I search in GHIDRA the address to set a breakpoint after the call to MessageBoxA, it’s the address 0x40101d.

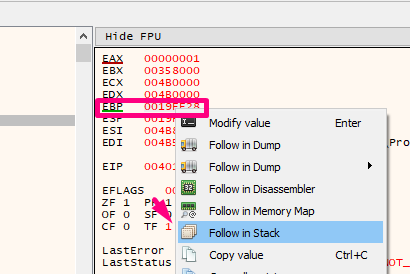


I search in x64dbg with right click - GOTO EXPRESSION

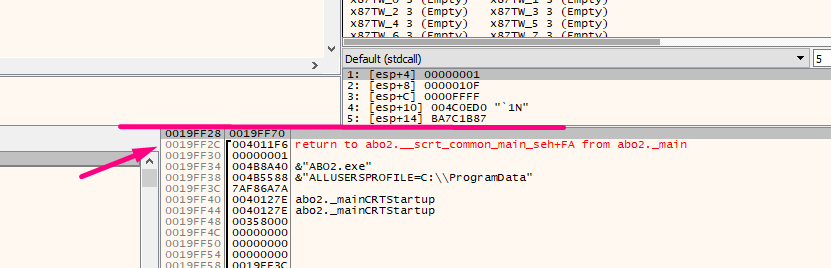


Set a breakpoint there, and then accept the MessageBoxA, so it will stop there:



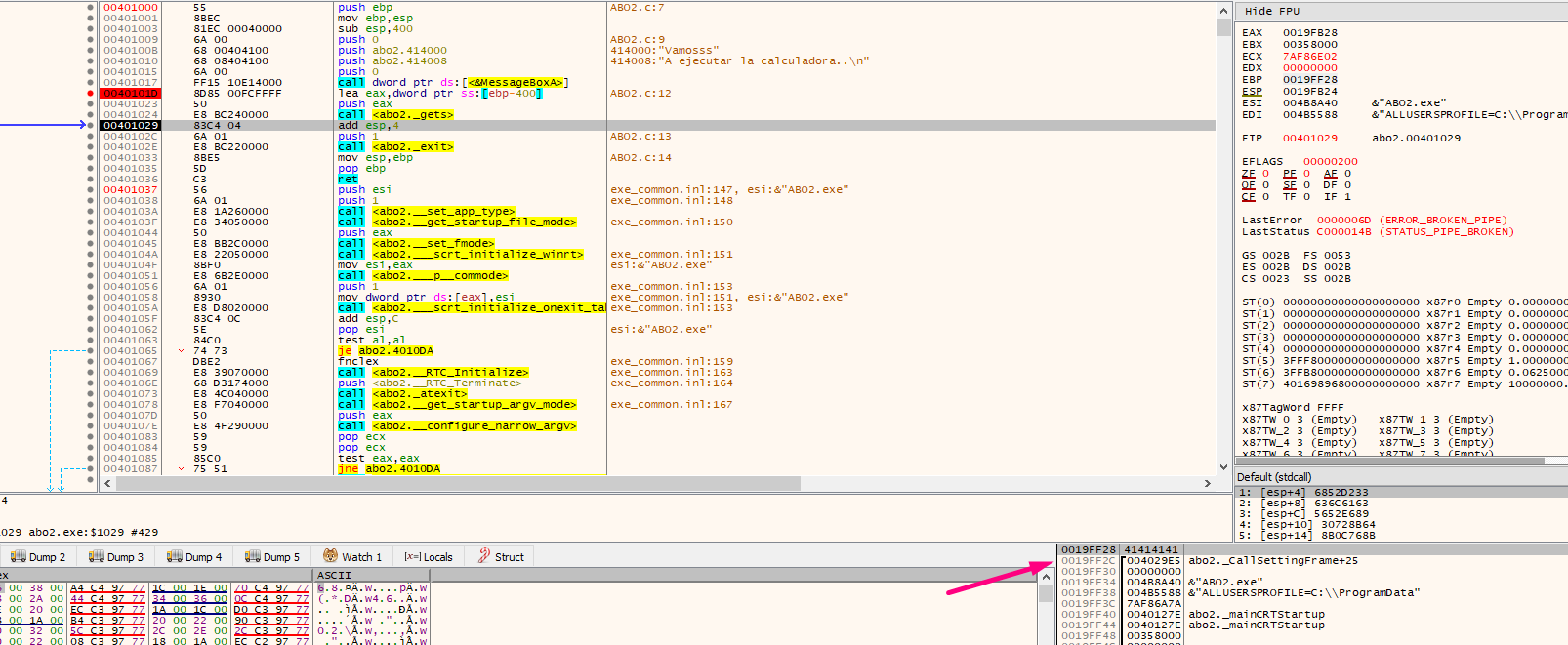


As we know that EBP is the horizon, I can right-click in EBP-FOLLOW IN STACK.

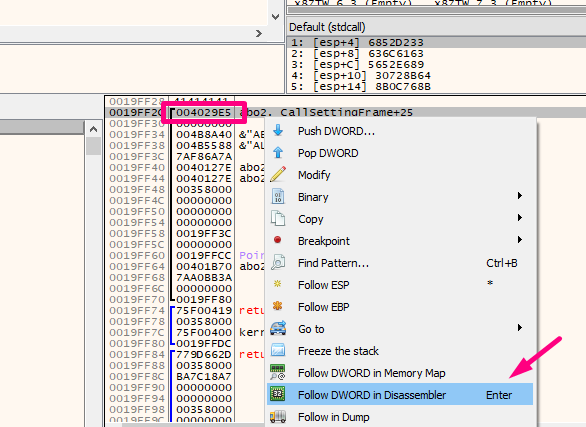


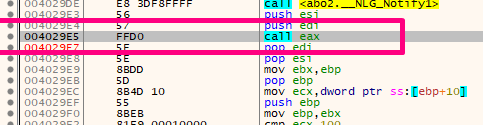
There it will be the horizon, in the line, right down it will be the STORED EBP, and below the RETURN ADDRESS, where it should return after executing main, in that case as gets was not called, there it is the original RETURN ADDRESS that it is 0x4011f6.

If I trace with f8 and I go through gets(), I will be able to see what happen with RETURN ADDRESS.



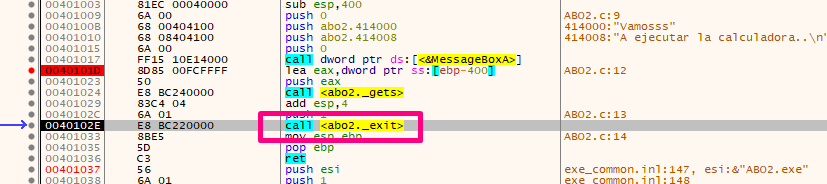
After stepping over with f8, I set again the horizon with EBP-FOLLOW IN STACK, and there I see that RETURN ADDRESS was correctly modified with the value 0x4029e5 that pointed to CALL EAX.





Well, let’s gonna see which is the program, continue tracing until we get the RETN of the function.

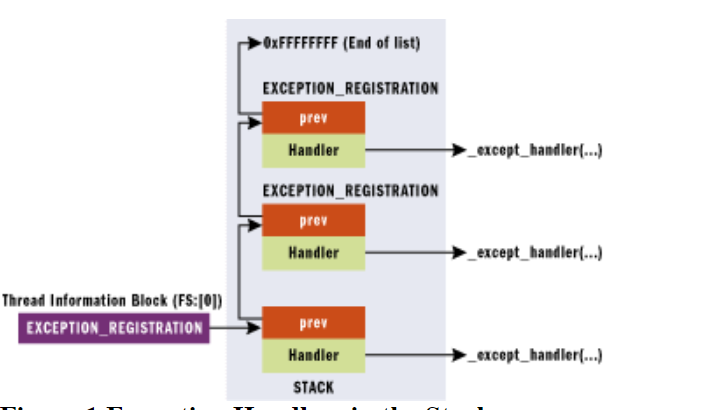
We see that before of getting the RETN instruction, there’s a CALL EXIT that makes the program finish.



So even if we modified the RETURN ADDRESS we can’t get the RETN instruction to modify the control flow to the CALL EAX because it closes before, so this way is not valid to exploit the program.

# EXPLOITING SEH IN 32 BITS.

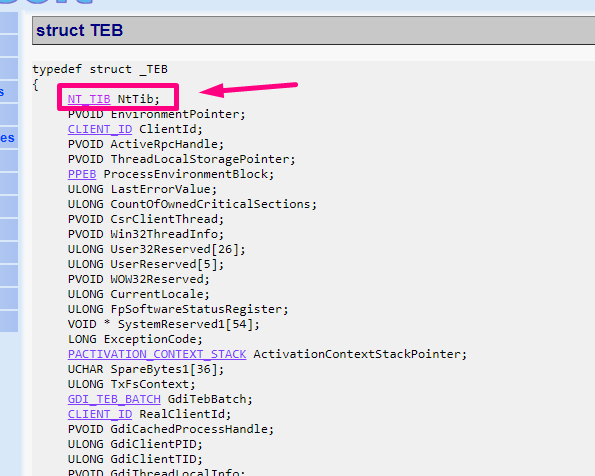
In 32 bits exist the possibility, that it is exploiting it modifying the exception handler or SEH that it’s stored on the stack.



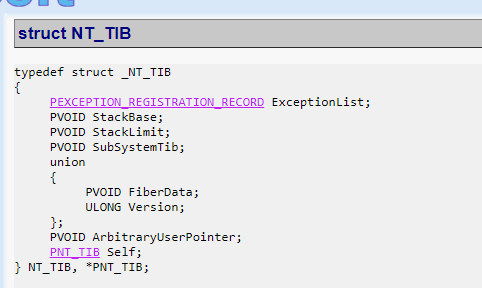
# TIB

If we remember the TIB structure in 32 bits from the part 6, we see that FS:[0] is the stack address where it would be the first of the chained structures.

I always say indistinctly TIB or TEB because both are same value, TIB is the first field of TEB structure.



There we see the TEB structure and its first field, that is the TIB structure, so the memory address is the same:

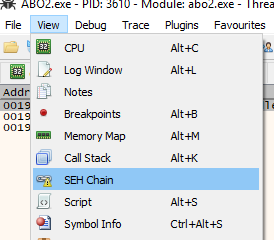


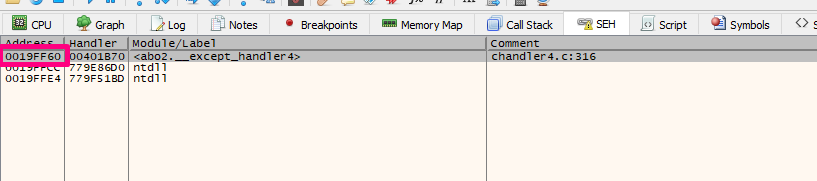
In the first field is the ExceptionList that it is a pointer on the stack where it starts the EXCEPTION REGISTRATION RECORDS chain.

# SEH CHAIN

If we go to VIEW - SEH CHAIN in x64dbg.

We have the EXCEPTION REGISTRATION RECORDS structures on the stack.

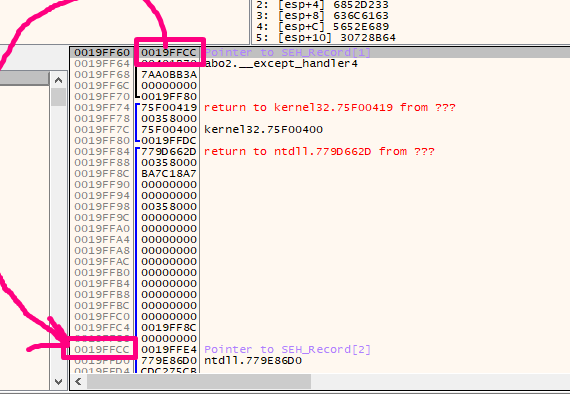




We can see there the address 0x19ff60 that it’s the value I can see in fs:[0] as we said the first field of the TIB.



So in the TIB structure the first field is 0x19ff60 that it’s a pointer to the first EXCEPTION REGISTRATION RECORD structure on the stack.

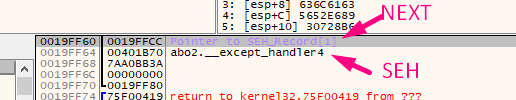


The nuts and bolts here is that the EXCEPTION REGISTRATION RECORD structures are 8 bytes, are made of two pointers, the first called NEXT that points to the next exception structure, and the second that it is the value of the SEH (the function handler).

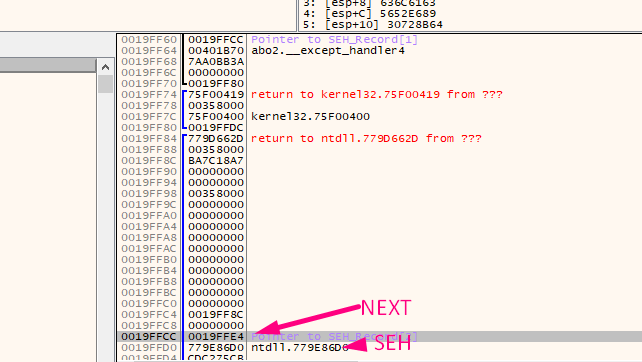
# NEXT Y SEH

There we can see it the first DWORD is the NEXT that points to the next exception structure, and the second DWORD is the SEH or HANDLER.

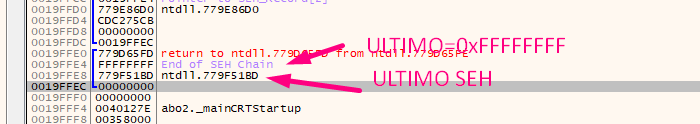
The last one points to the function that will be called when an exception happens in the program.



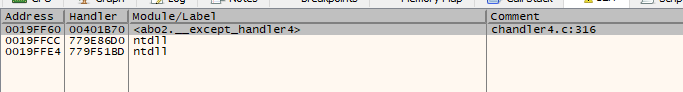
As we saw NEXT points to the next EXCEPTION REGISTRATION RECORD structure that it is in 0x19FFCC in my machine.



As we could see each of this structures point to the next EXCEPTION REGISTRATION RECORD structure, the next one will be in 0x19FFE4.



The last NEXT of the chain is in 0xFFFFFFFF and below the last SEH.

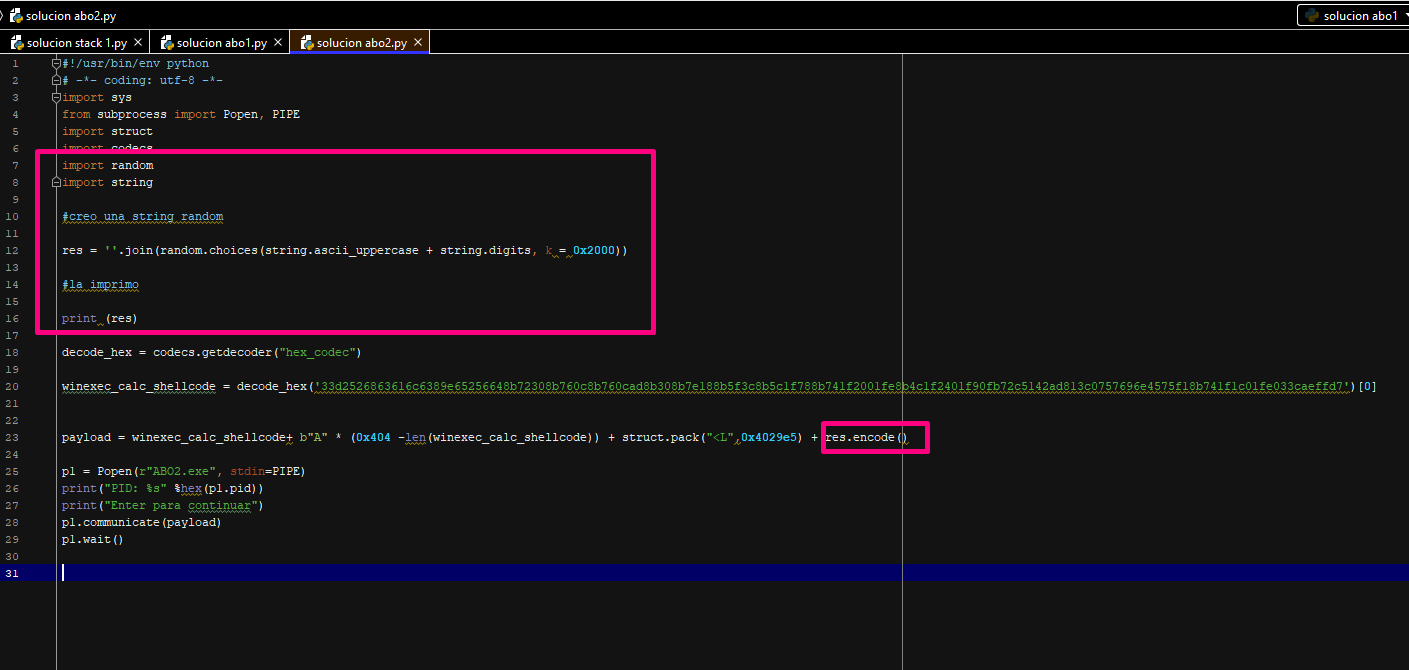


We can see that the list showed by x64dbg of each EXCEPTION REGISTRATION RECORD structure is correct, in the first column are the addresses of each structure, from the first one pointed by FS:[0], to the last one that has NEXT=0xFFFFFFFF, and the second column that are the handlers or SEH.

We already saw the the NEXT field of each structure points to the next EXCEPTION REGISTRATION RECORD structure, but for what it is the SEH or HANDLER, and how can we use it if we can modify it on the stack?

What it would happen if we can for example modify some SEH on the stack?

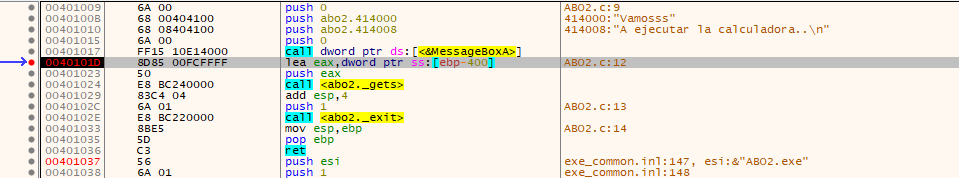
Just with that nothing would happen, we should try to make the program crash so if it tries to manage the exception it jumps to our address of our controlled SEH. If I fill the stack, and I continue writing until it finish its section, trying to overflow it, until the program crashes, that would produce an exception, and if we have any SEH modified with a value that we control, in this way catching the exception and trying to continue the program will jump wherever we want.!!!!!!!!!!!!!!!!!



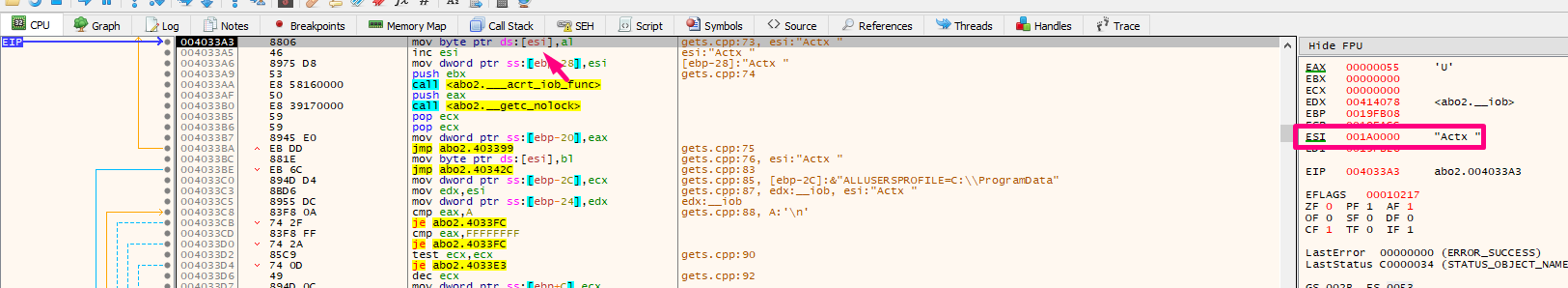
I will try with this.

I include a random byte string, big enough (0x2000) to overflow the stack and crashing the program, to add it I create it as string and then I change it to bytes (remember this is Python3) I this I do it with res.encode().

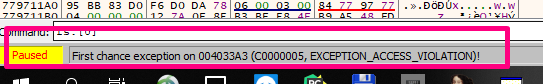
I run the script and I attach x64dbg as before until we go out of MessageBoxA.



When it stops in the breakpoint I trace with F8, when I try to stepping over gets it crashes because it copies more bytes than the stack allows, and as after that there’s not writable section it crashes.

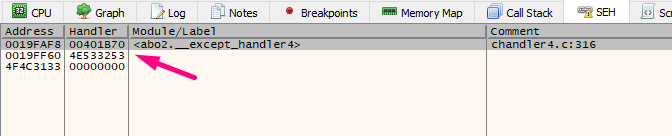


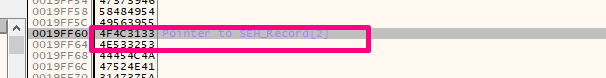
Right below It shows the exception:



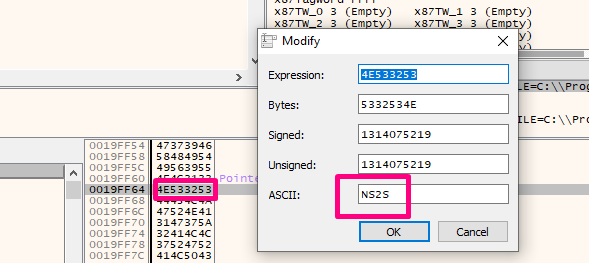
Let’s gonna see the exception list VIEW - SEH CHAIN

We see that the exception in the middle have been modified, If I go to the stack to see 0x19ff60:



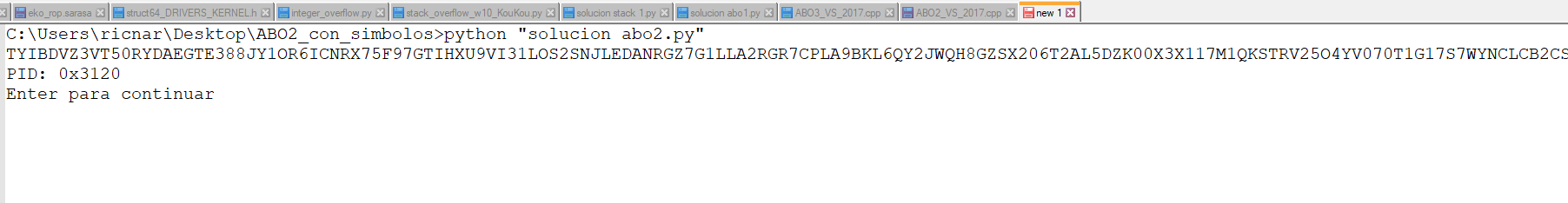


The NEXT of this structure is also modified, in my case with 0x4f4c3133 and the SEH with 0x4e533253.

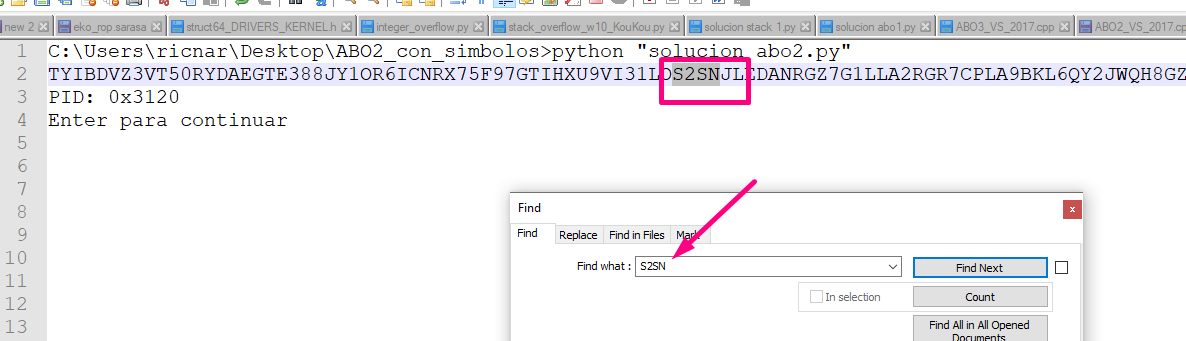


There I see the ASCII characters of those bytes that modified SEH.

Copy and paste it into a NOTEPAD with the output from the console.

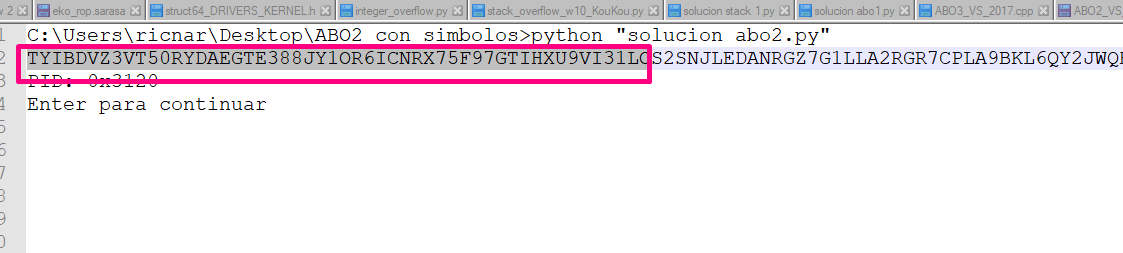


I search those ASCII values in the string that printed to see where are:

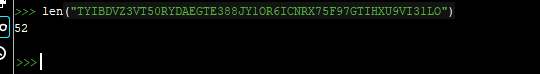


Because of little-endian those bytes will be reversed, because the stack show us as DWORDS.

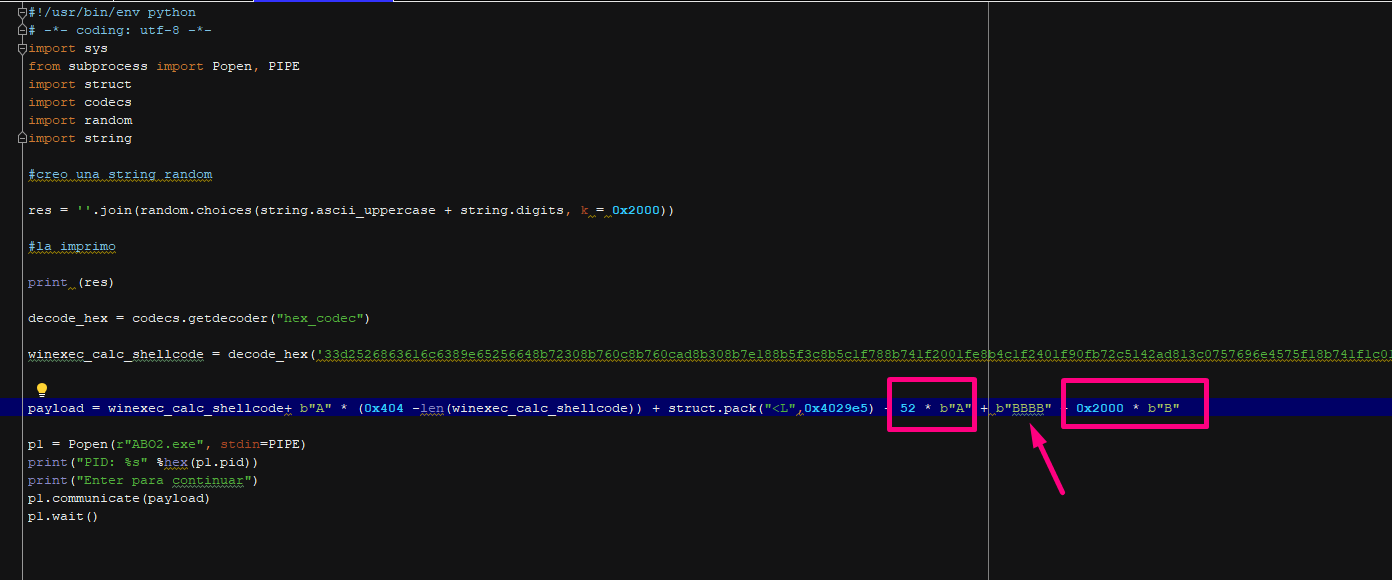
I copy the string until the values that modified the SEH:



And I copy it in PYCHARM to see the len:

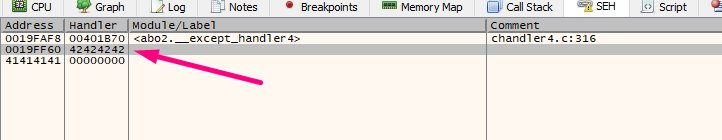


So after the value 52 from the beginning of the random bytes it is the SEH, the’s gonna try this:

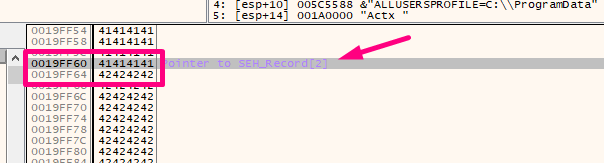


Replace the random bytes for 52 \* “A” and then it comes the SEH that I will modify with “BBBB” and then I fill the stack until the crash.

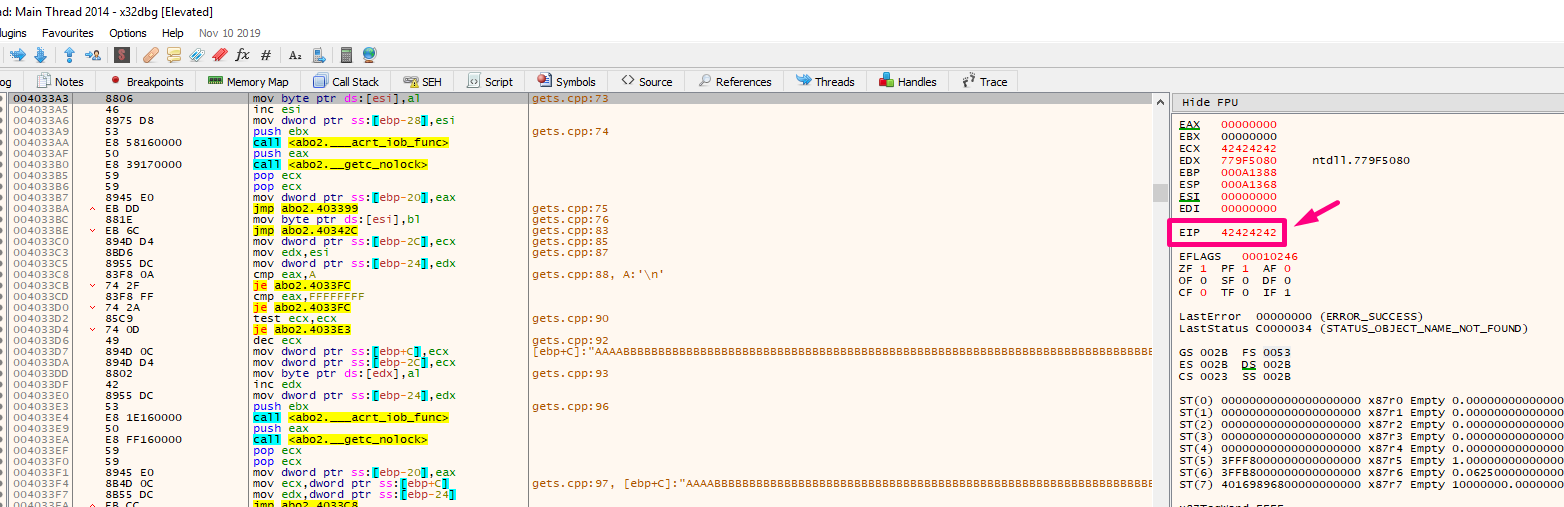
I execute it again and now I see how the SEH was modified after crashing:



There it is the structure, NEXT modified with 0x41414141 and SEH with 0x42424242

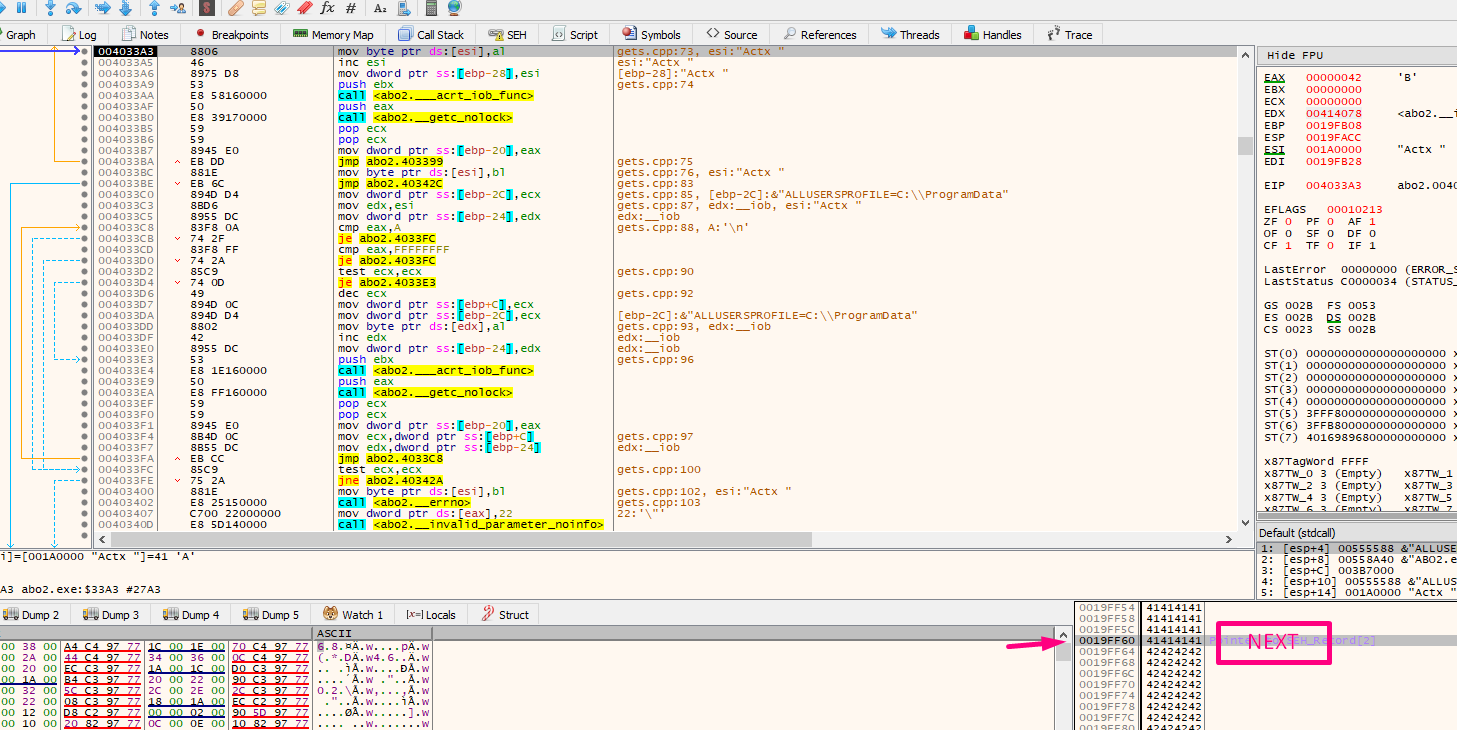


If I press SHIFT + F9 to manage the exception it would jump to 0x42424242, but there, there’s no code, try it:



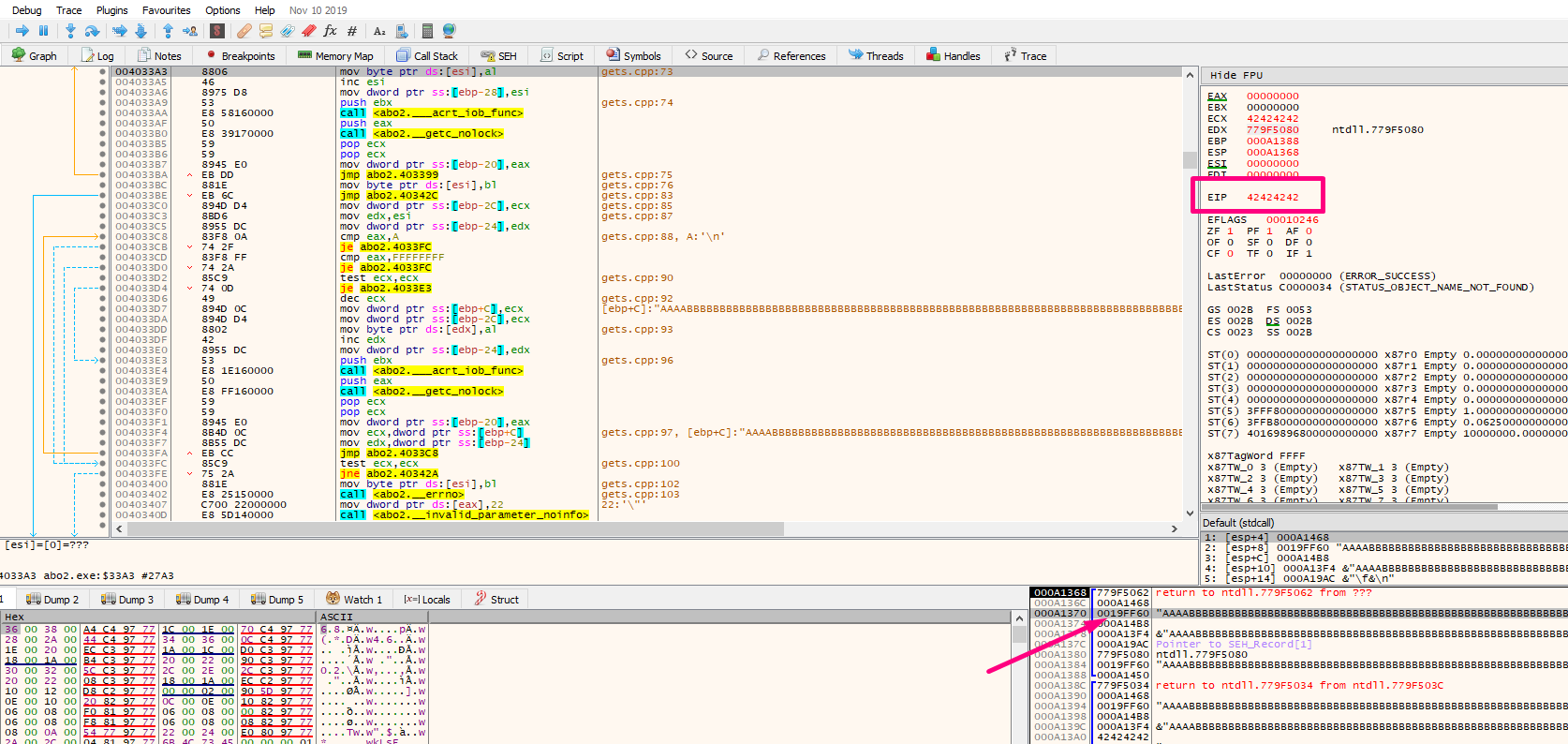
It jumps to 0x42424242 but as there’s no code it shows nothing.

If we see, before of jumping to the exception.



The NEXT in my case is 0x19ff60 and after managing the exception so it’s not obviously a direct jump, it jumps to the system, and after many checks it jumps to the SEH.

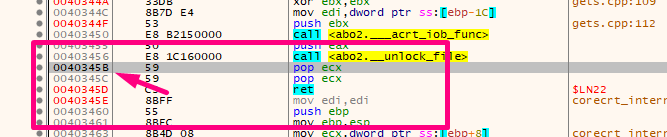
And once that jumped, we see that in the third position of the stack leaves ALWAYS the NEXT (because of the design of the exception handler of 32 bits).



The easier approximation it would be to leave in the SEH, a pointer to a code with two POPS and one RET so once it jumps there, with the two POPS it takes off from the stack the two first DWORDS and the RETN allows me to jump to NEXT that it is in the third position.

Let’s gonna search the POP POP RET in the code.

Searching a little bit we can see a POP POP RET in 0x40345b.



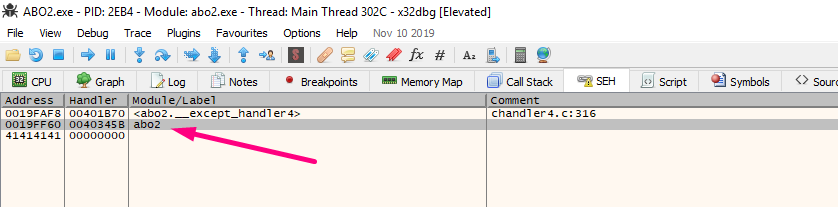
So replace the “BBBB” that modified the SEH with 0x40345b.



Let’s gonna see what happens.

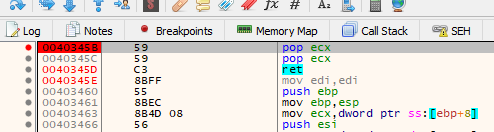
Attach with x64dbg.

And when it crashes let’s see the SEH:

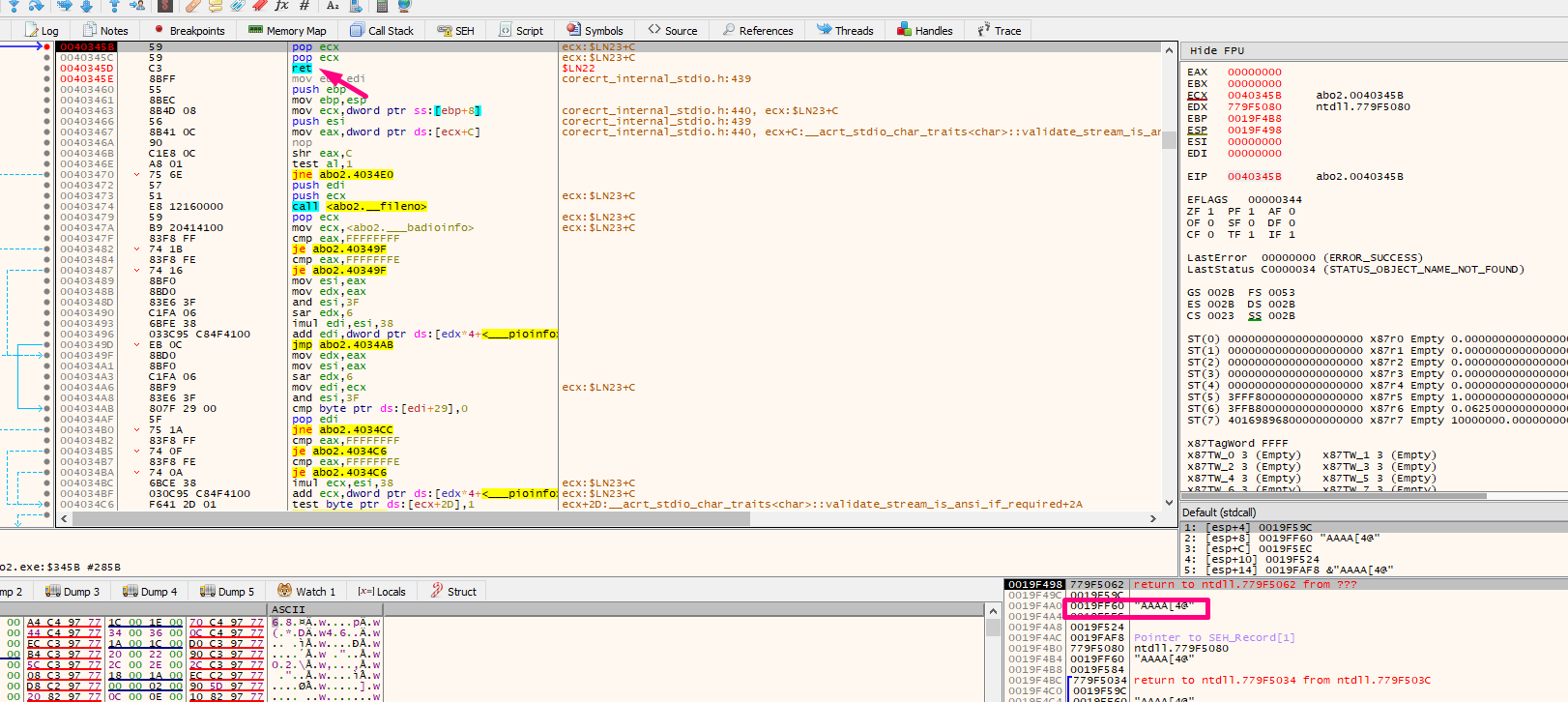


It’s modified with 0x40345b

Let’s set a breakpoint there before jumping.

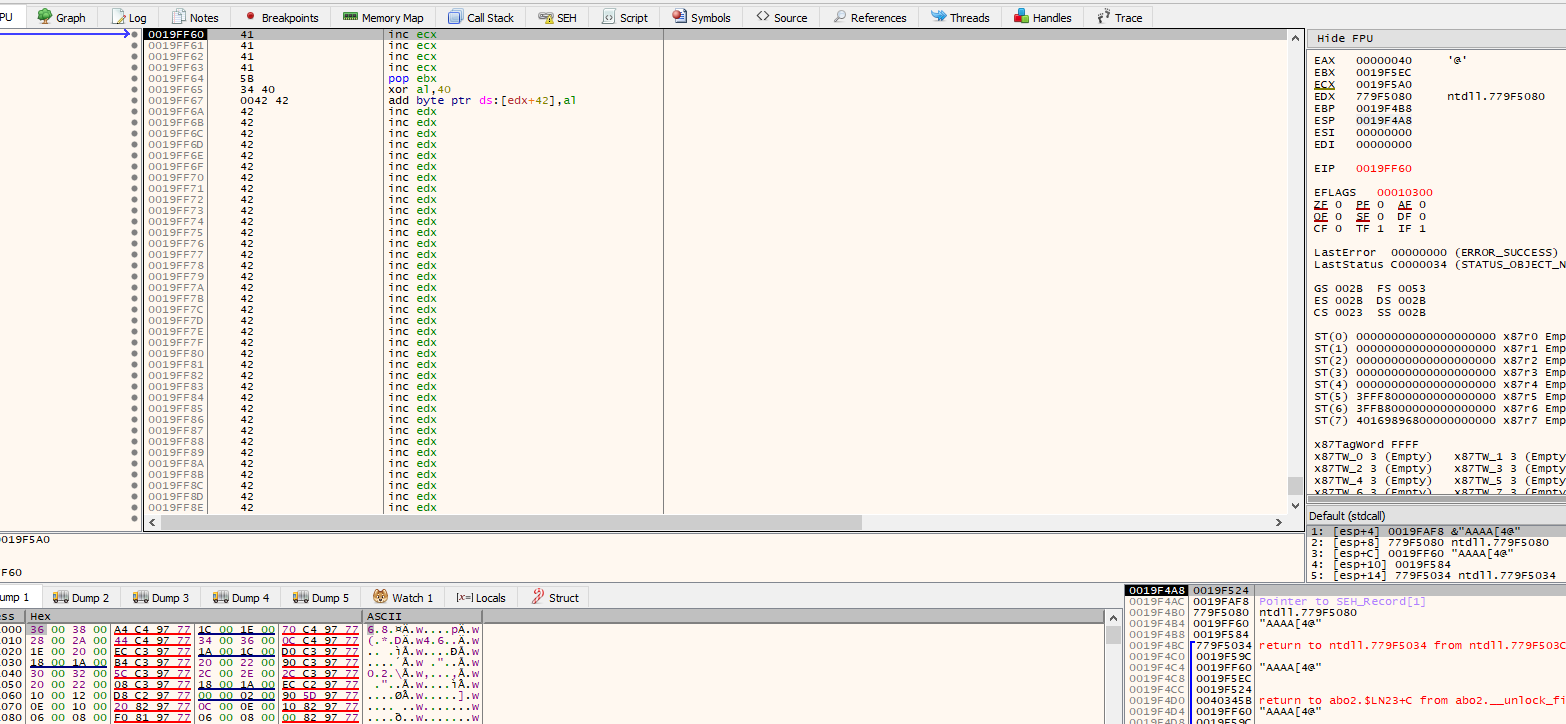


Now SHIFT + F9 to manage the exception and jump to 0x40345b.



We see that it arrived to the first POP that will take off first value from the stack, second POP will take off the second and then it will get the RETN.

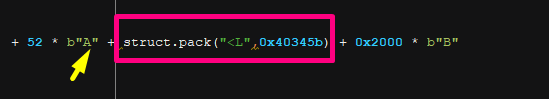
So it will jump to the NEXT I press F7.



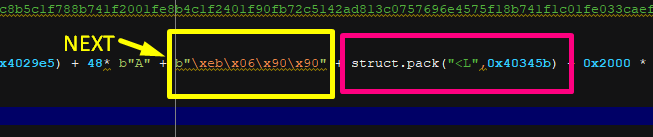
We’re now executing code (There’s not DEP in the binary), and as the 41414141 that were in NEXT are my value, I can control them and changing them, It’s necessary because right below are the SEH bytes.

Remember we are in the stack, the **41 41 41 41** are the NEXT and after that are the **5b 34 40 00** that it is the reversed SEH (**0x0040345b**) and I have to jump over there if not, it can crash because are not valid instructions, right below of that I can put whatever code we want.

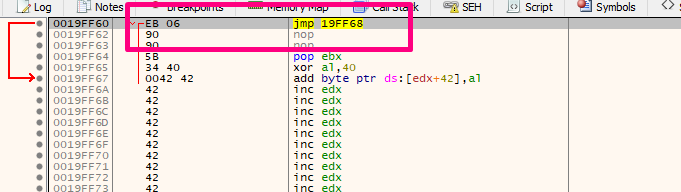
So let’s gonna modify the NEXT for **“\xeb\x06\x90\x90”**  that it is a direct jump.



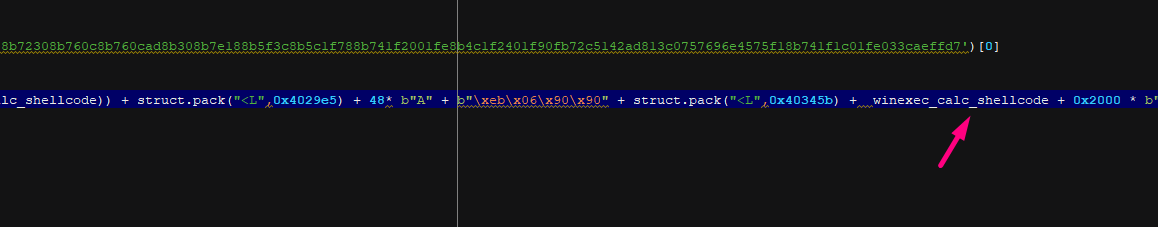
Now we can see in PINK the SEH and in yellow the NEXT just right before, it’s inside of the 52 ‘A’s so I have to write only 48 ‘A’s and the 4 bytes to maintain the 52 bytes from before:



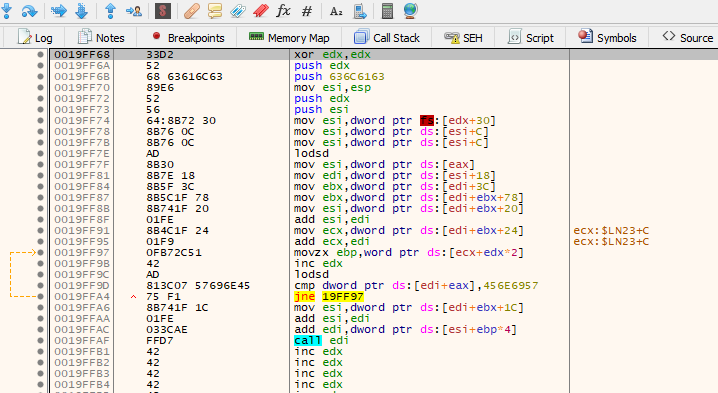
Let’s see now what happen:



We can see that **EB 06 90 90** is a jump to avoid the SEH and get the ‘B’ (0x42) that are right below where we will be able to write our shellcode:

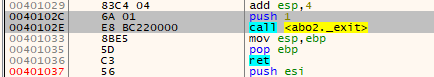


This will work but we missed something yet:

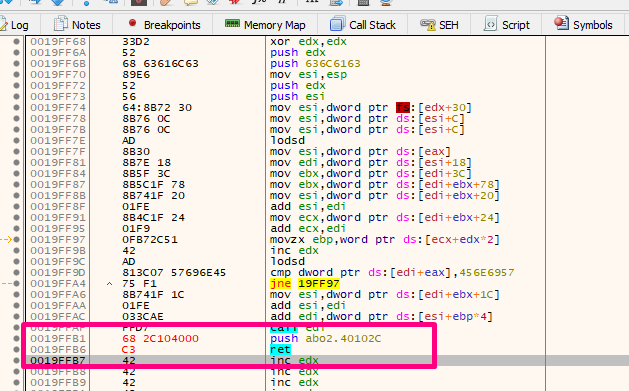


It executes the shellcode and will pop the calc, the problem is that once it finishes, it will crash, it will jump again to the SEH and it will execute the shellcode again, and it will do that always, it will pop thousands of calcs.

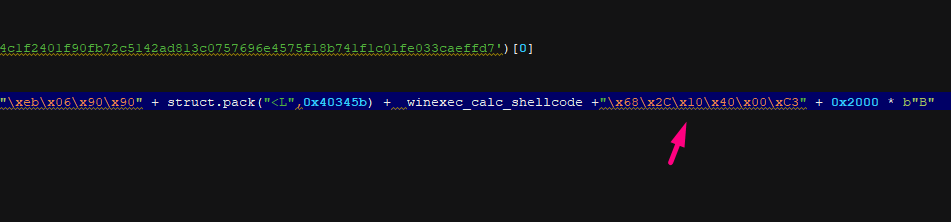
So we have to set a jump to exit after the shellcode.



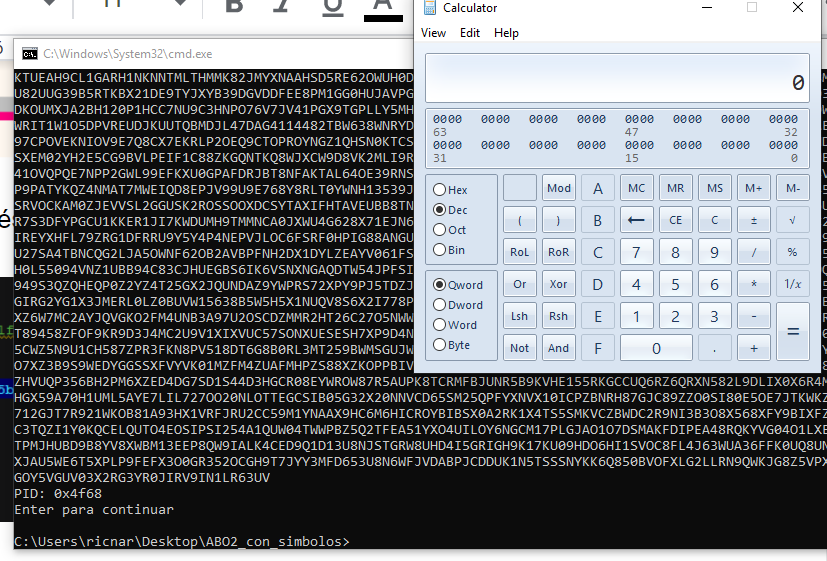
We can see that there’s a call to exit in the program:



If we add those bytes after the shellcode it will jump to exit.



And we execute it.



This technique has some restrictions, it can only be executed in modules that are compiled with SAFE SEH OFF.

Here there’s a POWERSHELL script that checks the state of SAFE SEH

<https://github.com/NetSPI/PESecurity>

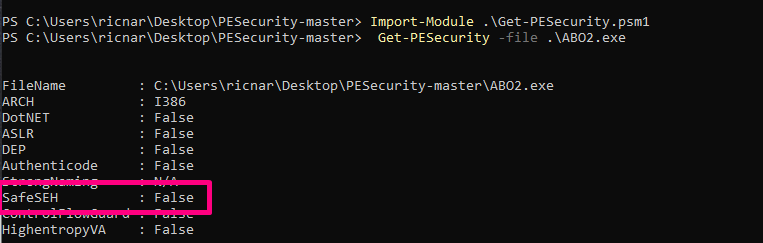
So if I open a powershell console.

And going to the folder with the script:

**Import-Module .\Get-PESecurity.psm1**

**Get-PESecurity -file .\ABO2.exe**

If ABO2.exe is in another folder set complete PATH.



So we can jump to this module, in other case we will have to search another module without SAFE SEH to jump.

With this we already have ABO2, and we saw SEH and how to exploit it in 32 bits, this method doesn’t work in 64 bits because the exception handler is not in the stack.

See you in part 8

Ricardo Narvaja

Translated by Fare9

29/03/2020