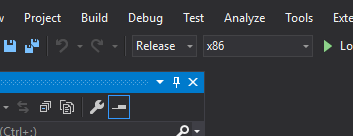
Reversing & Exploiting With Free Tools (Part 12)

Here we go with our first exercise compiled in 64 bits, we will take it carefully because we will have to see the things from a different/new perspective, but thankfully the base is the same, but it’s important to know the differences between 32 and 64 bits to success on reversing.

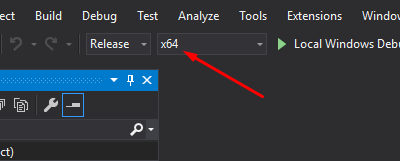
Starting with 64 bits

We’ve already marked some differences in previous parts.

Let’s going to clarify that when I talk about programs compiled in 32 bits what we call “of 32 bits”, I’m talking about those compiled in x86 wherever they run later (in an OS of 32 bits or in WoW64).



And when I talk about those of “64 bits” I’m talking about those compiled in x64.



There you have the Visual Studio options to compile in x86 or x64 (what we will call “32 bits” and “64 bits”)

We already know that if the process is for 64 bits, all its modules will be of 64 bits, and it will have DEP always enabled. I’ve tried to compile in 64 bits without DEP but at the end DEP is always enabled, also working I’ve never seen a process of 64 bits without DEP so it’s a truth based on experience.

Neither exploitation based on SEH is possible, SEH doesn’t exist on the stack as with 32 bits, a function table exist to manage the exceptions, but not on the stack, so this method doesn’t work, it’s exclusive of 32 bits.

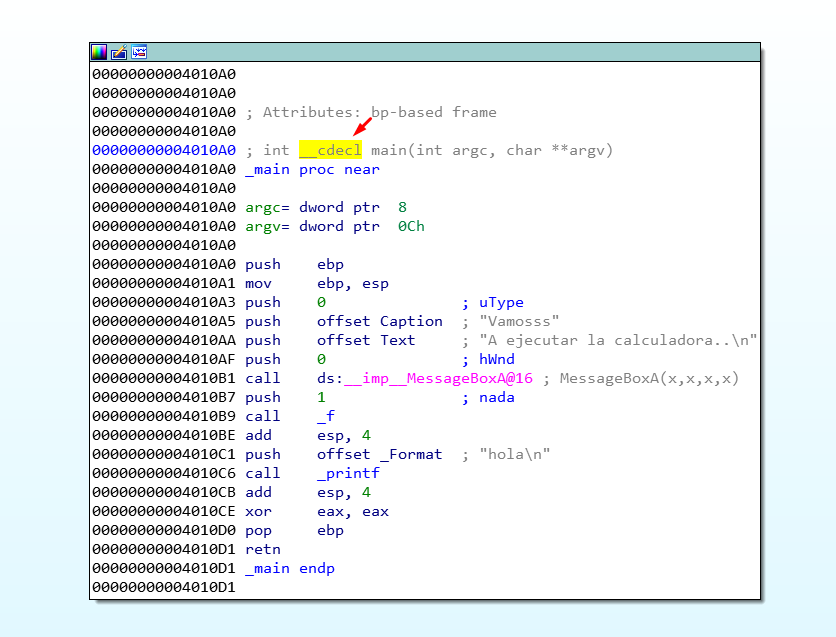
So we see that the work is more complex, we also have to costume to the way how arguments are given, while in 32 bits the stack is used, here it’s a little bit different.

CALLING CONVENTIONS in 32 BITS

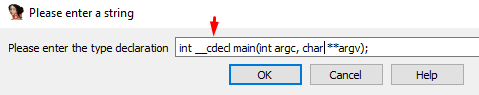
When we talk about calling conventions (CC from now) we are talking about different conventions used to call functions and give them their arguments, the most important thing for us of each CC is:

* How arguments are given to the function (through the stack, through the registers or a combination of both).
* How the job of preparing the stack before and restoring after a function call, between the caller and the callee.

If we see the examples of 32 bits from previous parts, we can see:



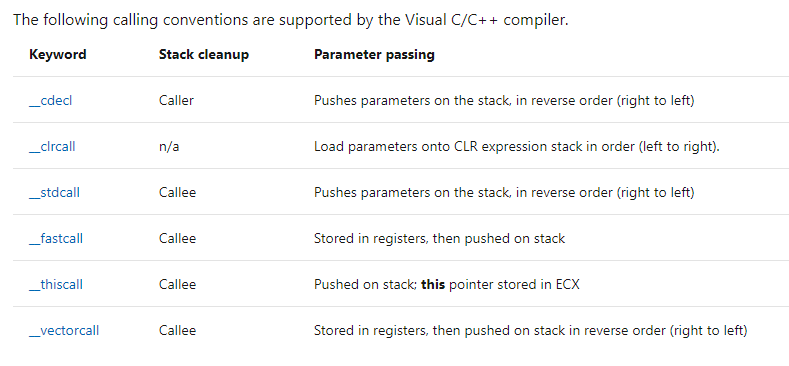
Doing right-click -SET TYPE in the function’s name.



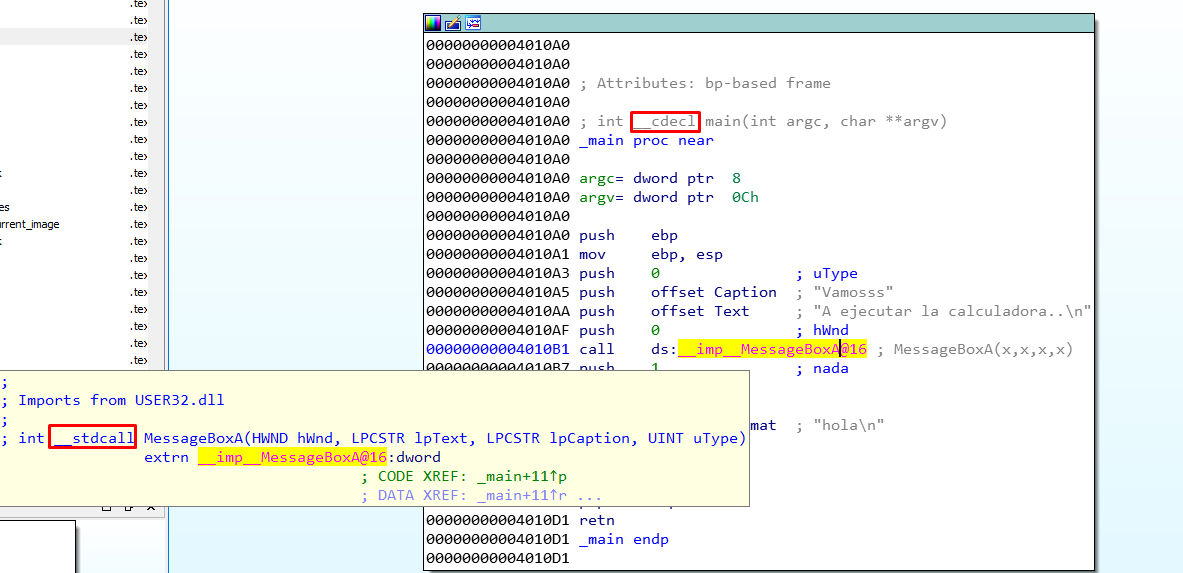
We see that before of function’s name there’s a word, in this case \_\_cdecl, that tells us the CC type that is used to manage the arguments, and so on.

Here a list of CCs we can commonly see:

<https://docs.microsoft.com/en-us/cpp/cpp/calling-conventions?view=vs-2019>

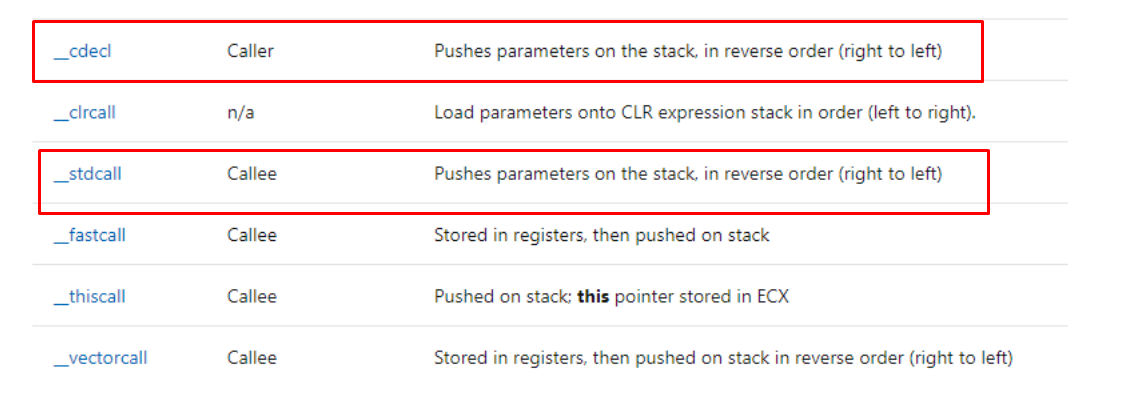


In our work with Windows in 32 bits, the more used CCs are \_\_cdecl and \_\_stdcall.



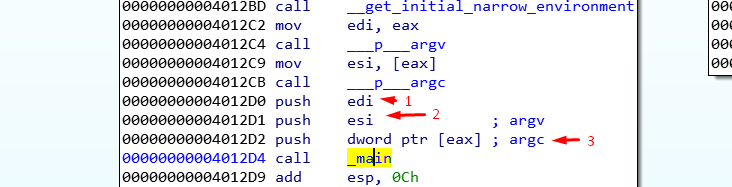
Moving the mouse over the MessageBoxA function we see that it uses the CC \_\_stdcall (because Windows DLL functions commonly use this calling convention).

As we can see in the next table, both CC use the stack, as we saw in the 32 bits exercises, to pass the arguments to a function, there says that both use REVERSE ORDER:



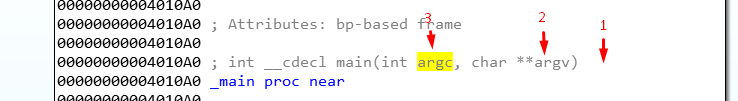
REVERSE ORDER

reverse order means the order how the arguments are pushed regarding the function declaration. Just to clarify this, PUSH is not always used to pass the arguments, but the idea is the same (sometimes mov [esp+x], value is used).



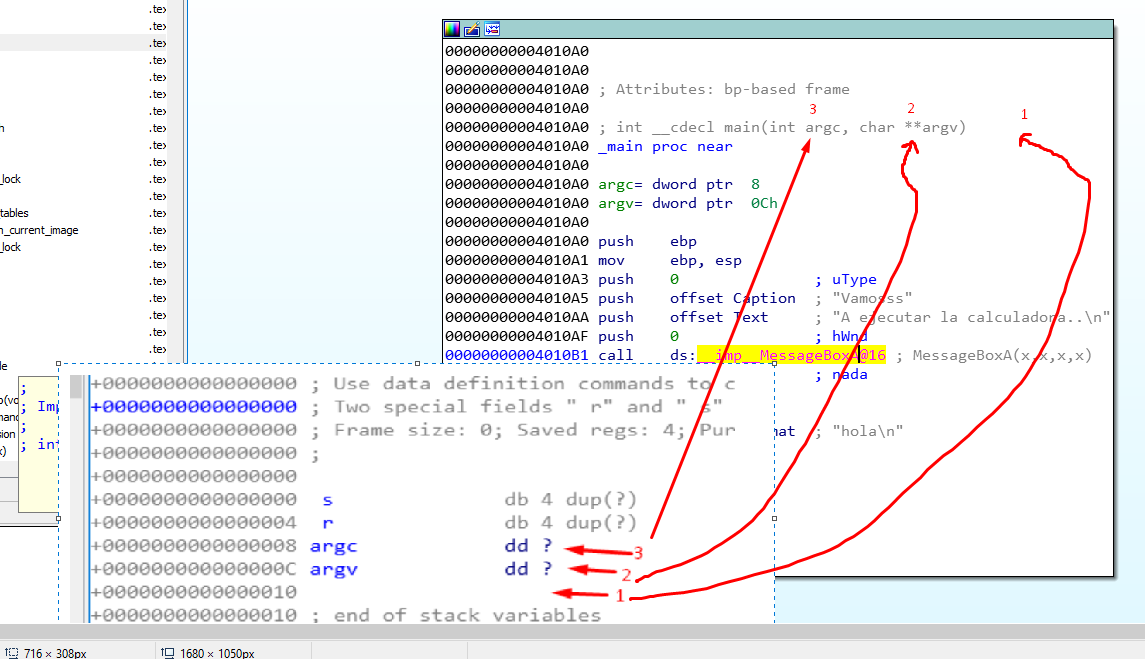
There we can see the call to main that we get pressing the X in the main function, we see that 3 arguments are given, first PUSH EDI that it is not used (envp), then PUSH ESI that is the second PUSH (argv), and finally PUSH DWORD PTR [EAX] the third PUSH (argc).

If we return to the function IDA only shows 2 arguments, because only those two are used and discard the non-used.



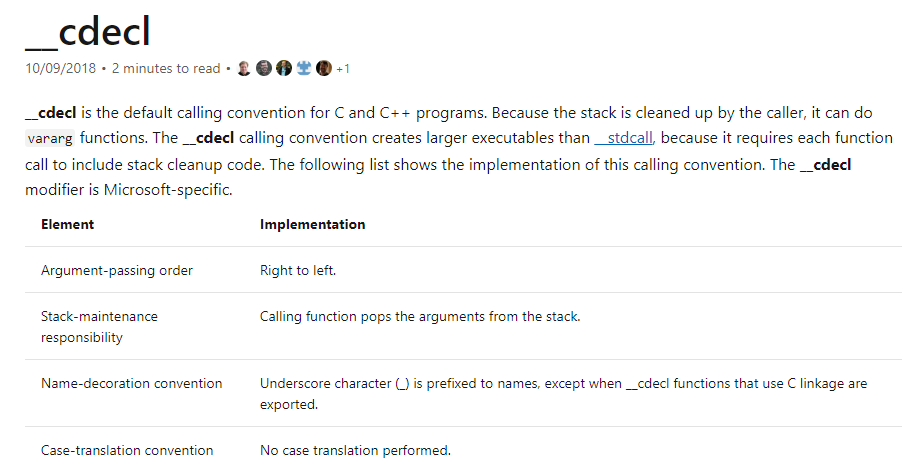
We can see that the first PUSH correspond to the last argument in the right of the function declaration, for that reason is REVERSE ORDER, because the arguments are pushed from right to the left in the function declaration.

This is because in this way the first PUSH is below on the stack, the second PUSH above of this one, and the third one (first argument) on top of the stack, after this the return address is pushed.



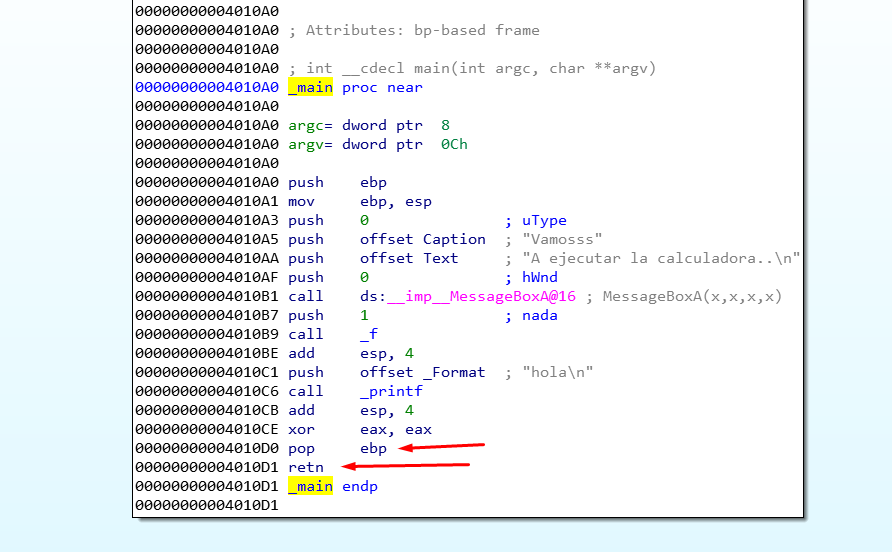
CALLING CONVENTION \_\_CDECL

Here we see its characteristics



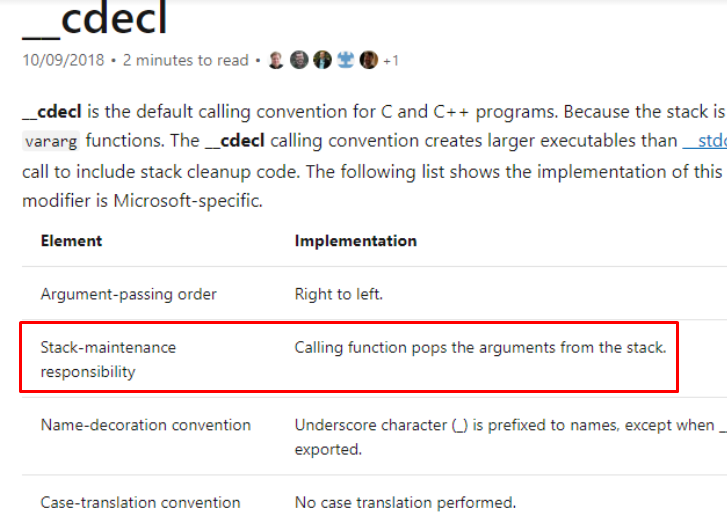
Two first points are what we will care about, arguments are passed through the stack from right to left (REVERSE ORDER) and the work of cleaning the stack from the given arguments corresponds to the caller.

How can we see this?

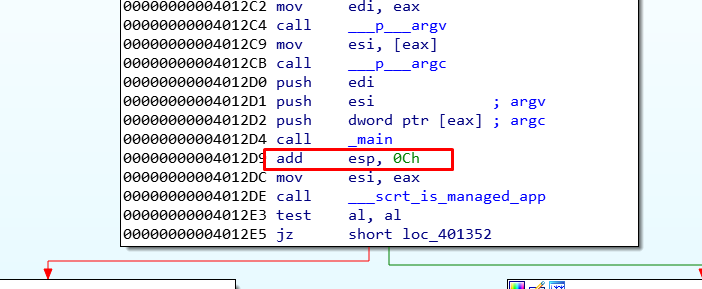


We can see that the function received three arguments on the stack, if the work of cleaning the stack at the end corresponds to the callee in this case main, it should be before of RET function and apart from the POP EBP that restores the STORED EBP, three POPs more to clean the arguments that were PUSHED on the stack before calling the function, so the stack would be cleaned poping as much as arguments as were pushed.

We can see that in \_\_CDECL we don’t have that, the callee doesn’t balance the stack, and is the caller task.



There it is, the caller function must do it, so we’ll see what happens when main finishes and returns to its caller function.

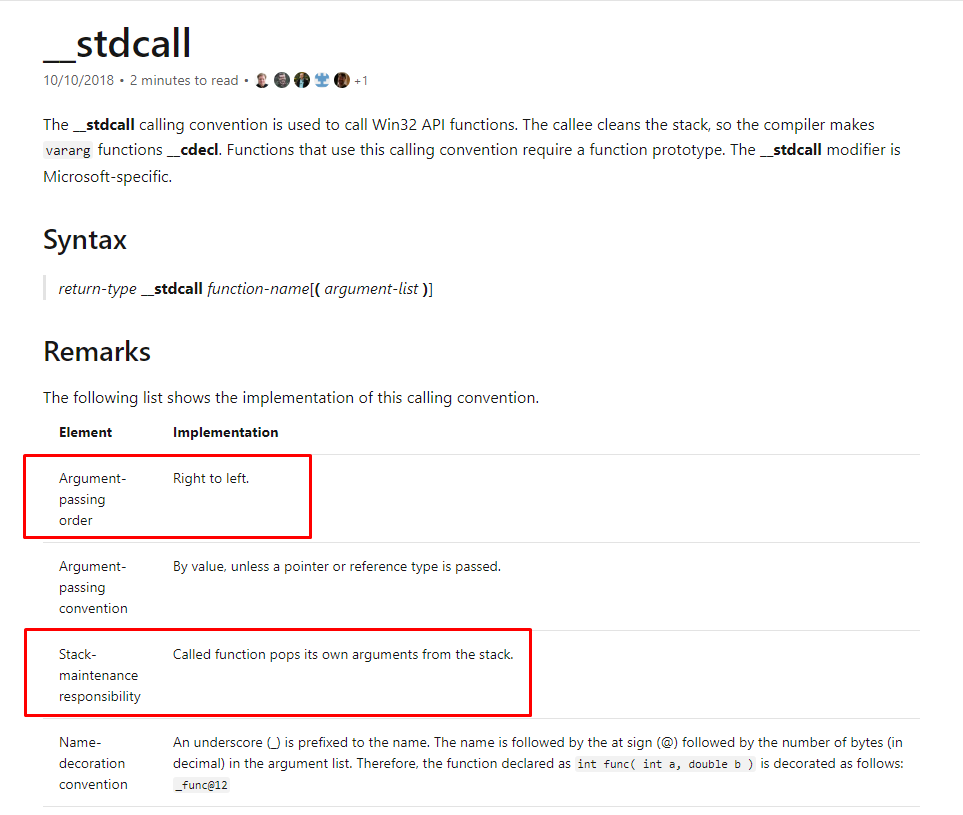


There it is the function caller of main, with that ADD ESP, 0xC it moves the stack in the same way than three POPs but without moving any value.

So the characteristics of this CC that is used in 32 bits is the REVERSE ORDER of the parameters on the stack and the CALLER FUNCTION cleans the stack.

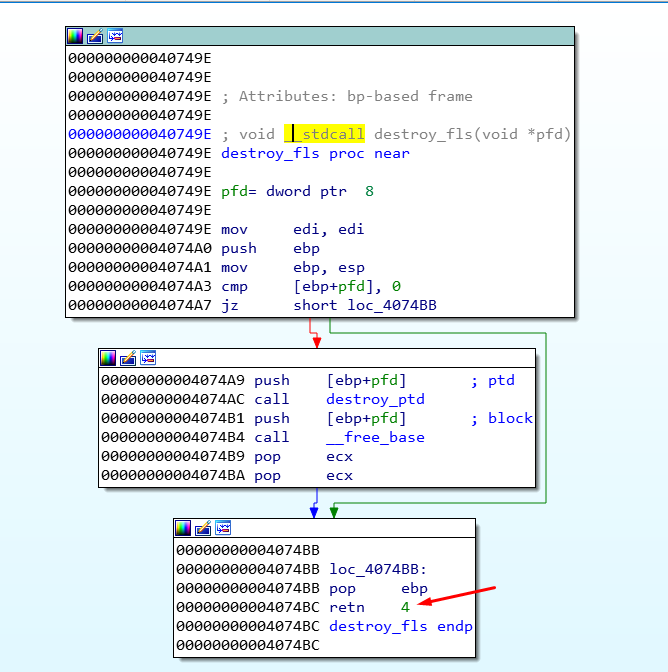
CALLING CONVENTION \_\_STDCALL

<https://docs.microsoft.com/en-us/cpp/cpp/stdcall?view=vs-2019>



We see that the arguments are also given in REVERSE ORDER, the difference is that the CALLEE cleans the stack instead of the CALLER.

Here we have an example in the same executable:



We can see that the function has only one argument, only one PUSH was done to give it the argument.

At the end of the function if \_\_cdecl were applied there would be a POP EBP-RET and the CALLER would clean the stack with an ADD ESP, XXX.

In this case we see that the CALLEE cleans the stack, so before of RET, either a POP is added or ADD ESP, 4 or as it happens, in this case, RETN 4 that it will return and then clean 4 bytes from the stack as a POP.

RETN 4 = RETN + ADD ESP, 4

If the function has 2 arguments:

RETN 8 = RETN + ADD ESP,8

If the function has 3:

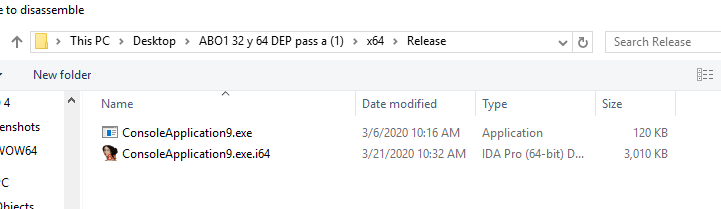
RETN 0C = RETN + ADD ESP, 0C

In general RETN X is used but we could have functions that before RETN has various POPs or an ADD ESP, XXX to return to the CALLER with the stack clean.

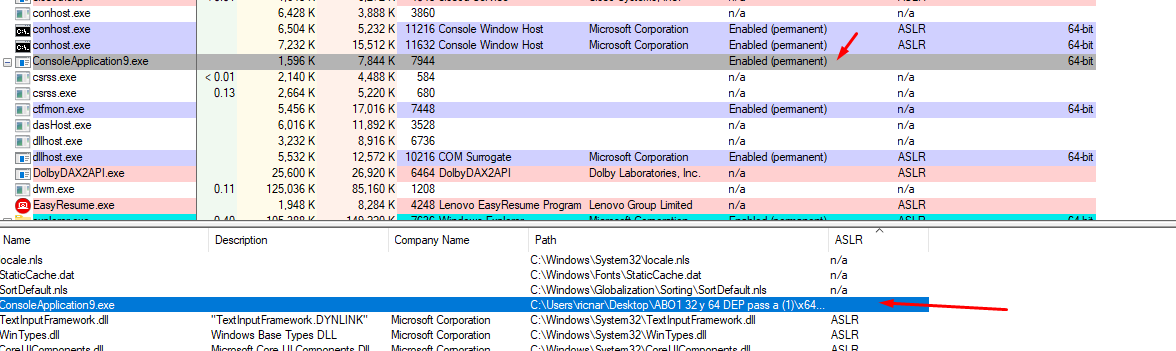
CALLING CONVENTIONS IN 64 BITS

Finally we reached the point we want the CC in 64 bits, let’s going to open the exercise of 64 bits in IDA FREE.

We can access the exercise here: https://drive.google.com/open?id=1nmPR6q5SVmS5dsJ6y9oXLUsJzyC2xJGG

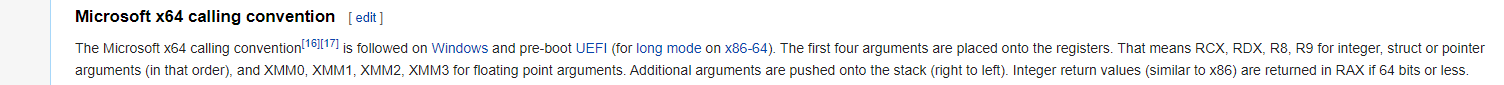


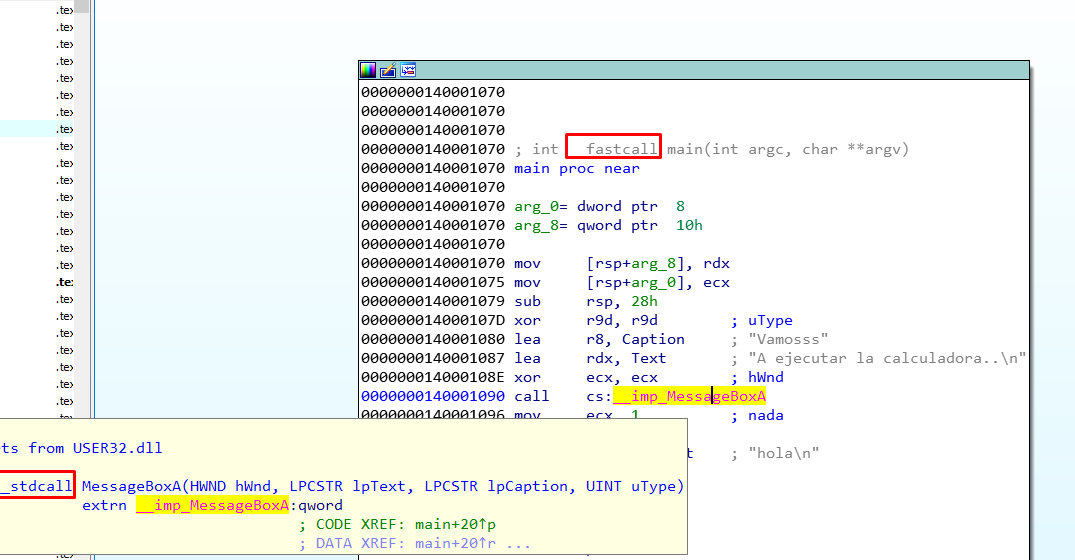
If we execute it, we can see something like the next picture:



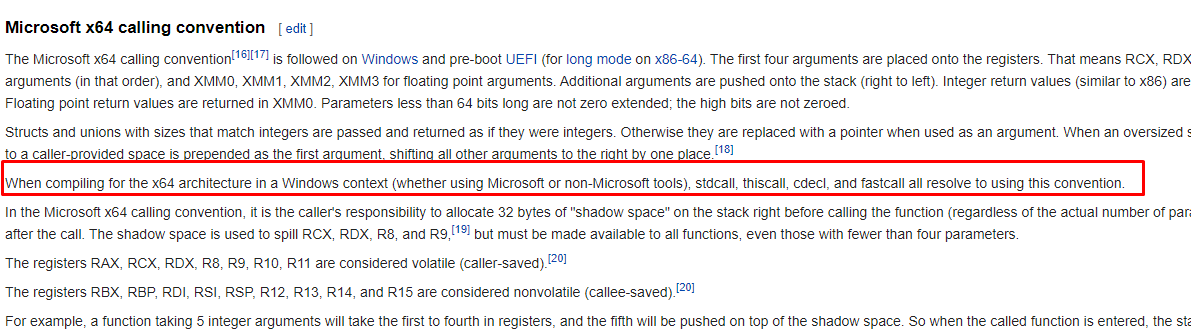
MICROSOFT x64 CALLING CONVENTION

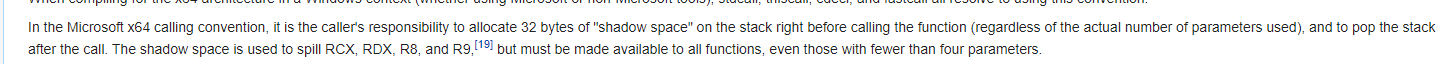
If we open the exercise in IDA we can see different CC in the functions, but independently of what it says, in Windows x64 only MICROSOFT x64 CALLING CONVENTION is used in any case.





We see that in some functions says \_\_fastcall and \_\_stdcall, but Windows uses its own CC, as we can see the first 4 arguments are given through the registers RCX, RDX, R8 and R9 in this order, and if more arguments are given, the stack is used.





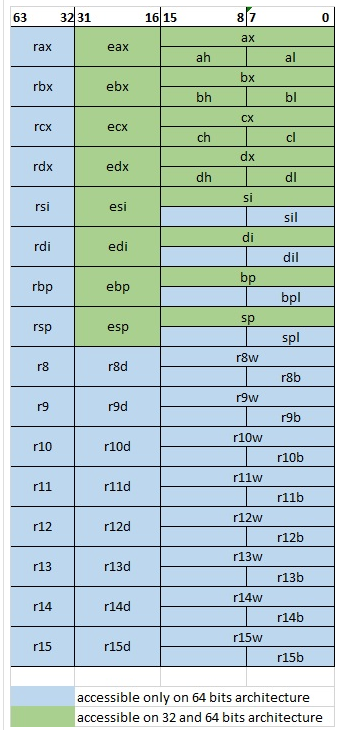
So here the CALLER, even if it doesn’t use it, it must allocate 32 bytes on the stack, this is known as SHADOW SPACE of before calling a function, and if it’s necessary it must clean the stack, we’ll see this in the practical part.

So this SHADOW SPACE must exist in the caller, even if it calls functions of 1, 2 or more arguments, it will be present, and it will be used for the CALLEEs to save the arguments from the register if they need the registers.

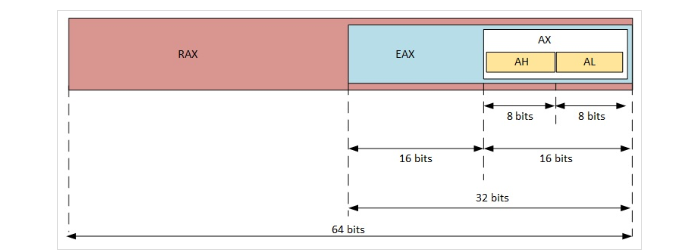
If a function must receive more than 4 arguments, those must be pushed on the stack below of the SHADOW SPACE.

REGISTERS IN 64 BITS

For those that don’t know the registers in 64 bits here is the complete table:

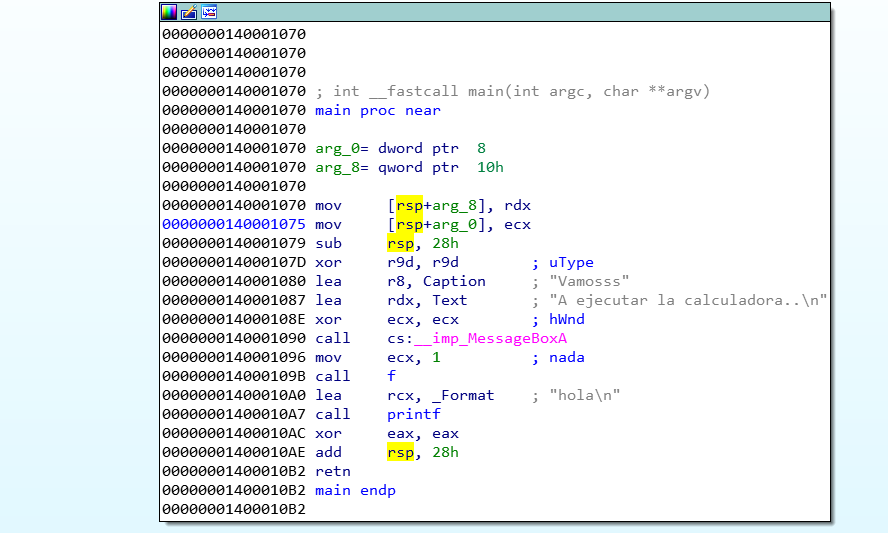


Green registers are those accessible on 32 and 64 bits, and the blue ones are those that only exist on 64 bits.



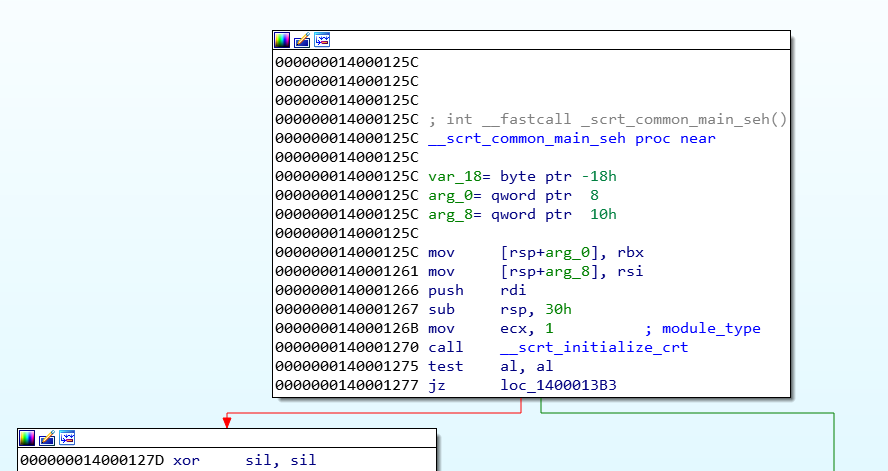
For example for RAX of 64 bits, the low part is EAX of 32 bits, and AX of 16 bits composed by AH and AL of 8 bits.

REVERSING IN 64 BITS

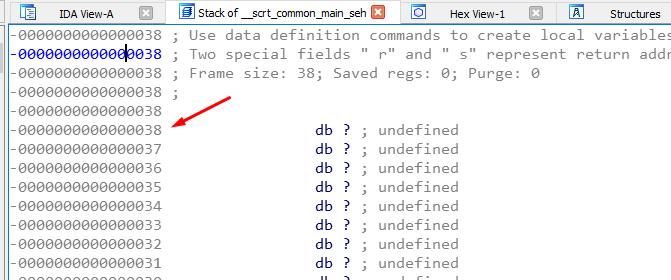


As a first view, we realize that here almost always functions are RSP BASED, and variables and arguments are referenced as RSP+XXX.

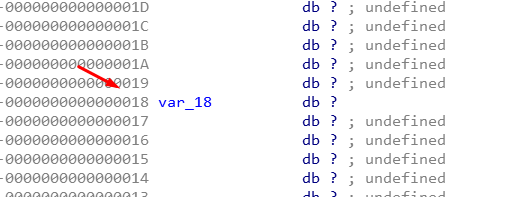
Let’s going to take a look at the main’s CALLER.



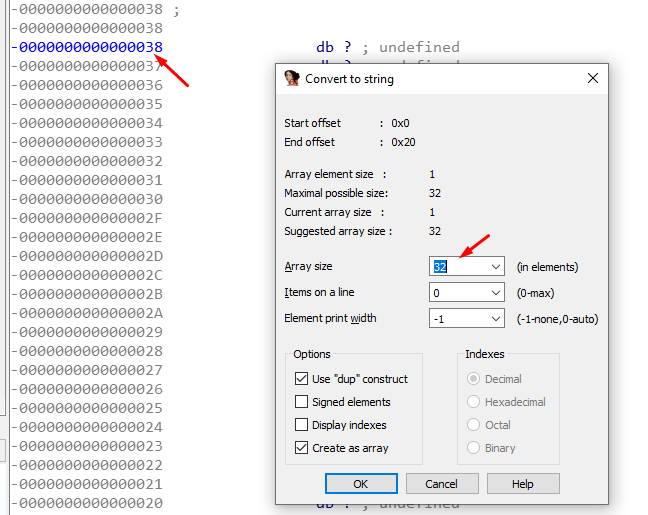
We see that it is a function with only one variable, if we go to the static representation of the stack we see that IDA shows us that it goes to -0x38 while the only variable is down.



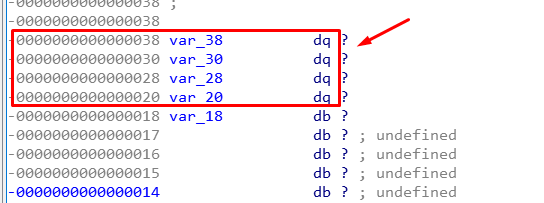
The variable is in -0x18



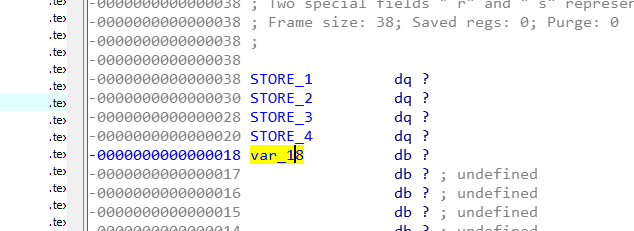
If in -0x38 we press A



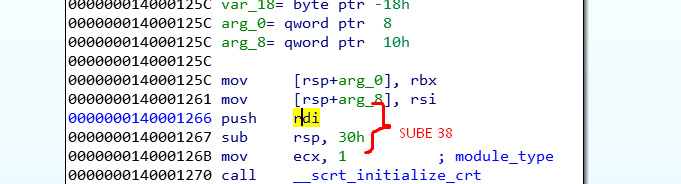
We see that just below of the variable we have 32 bytes, as I said before this is the SHADOW SPACE or an allocated space for when a function is called, as for example main.



Pressing the key D to change the types, we create 4 QWORD variables, that will be part of SHADOW SPACE.

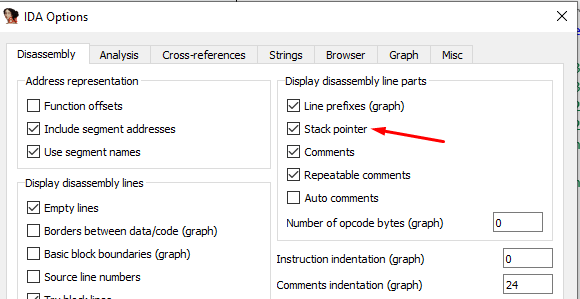


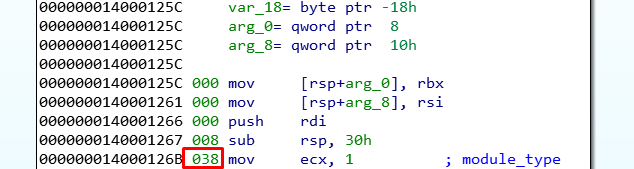
Remember that this space with this 4 variables will not be used by the caller, but is allocated for the callee.



Here we have push rdi, and sub rsp, 30h. This will allocate 38 bytes on the stack, so RSP will be above of the SHADOW SPACE.

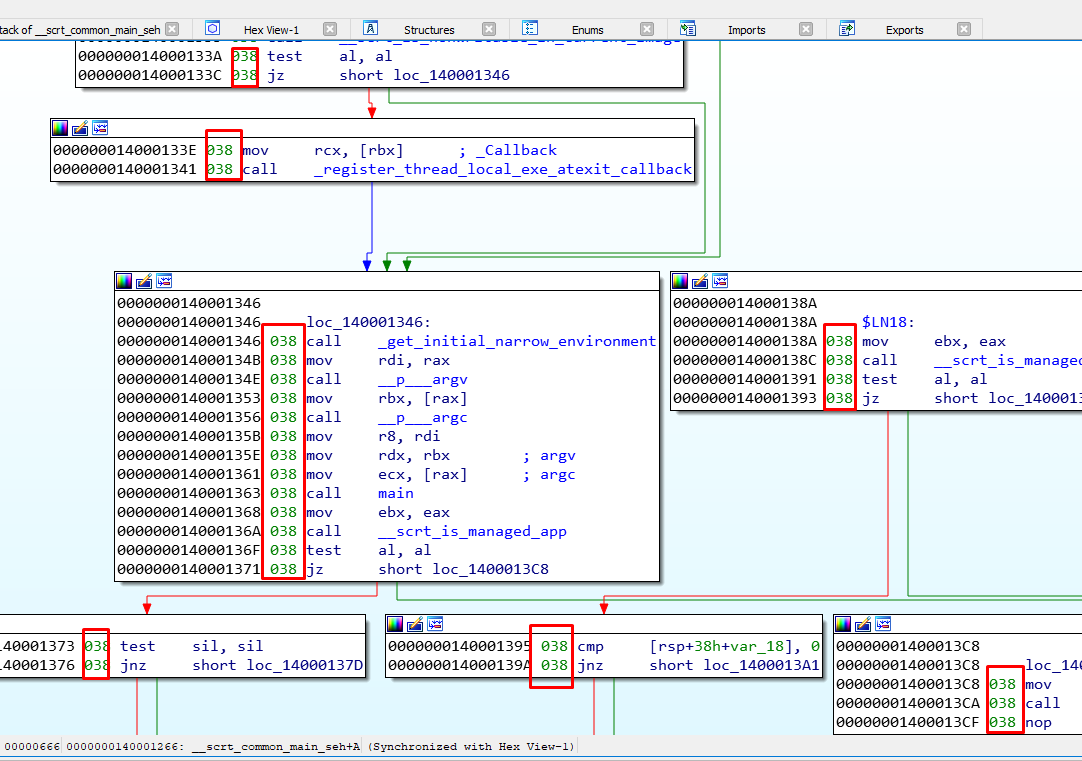
If you don’t believe me, in IDA you can see the static stack variations.

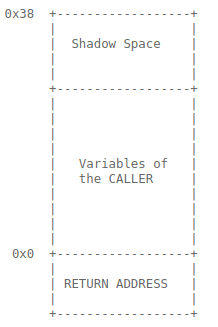




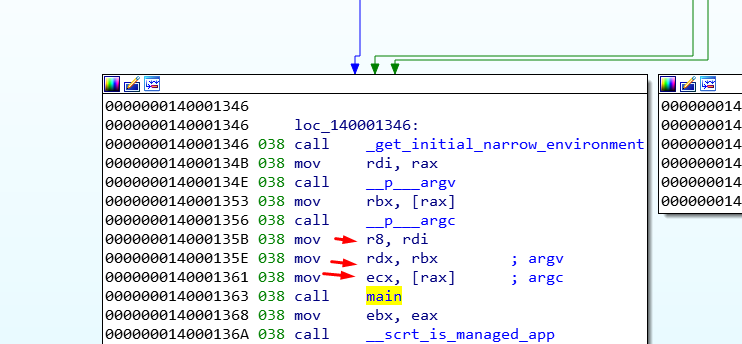
We see that after PUSH RSP it will be -8 and after the SUB RSP,0x30 the register RSP will be -0x38 compared with its initial value.

From that moment RSP will be constant and used as reference.





So RSP now is in -0x38 and with a space that will not be used, but when the callees use it and store them registers, they will not modify variables of the caller because those variables are right below.

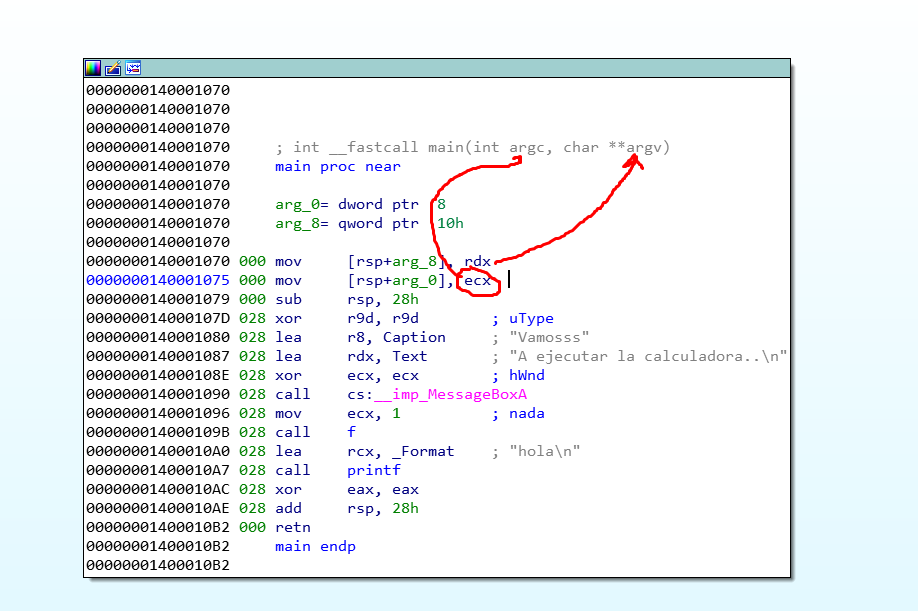


Here as there’s no PUSHES it doesn’t matter the reverse order, just in case more than 4 arguments are given.

It moves to RCX (in this case EDX because it’s a DWORD of 4 bytes) the first argument (argc), it moves to RDX the second argument that it is a QWORD of 8 bytes (argv, a pointer) and then it moves to R8 the third argument, also a QWORD (envp, a pointer).

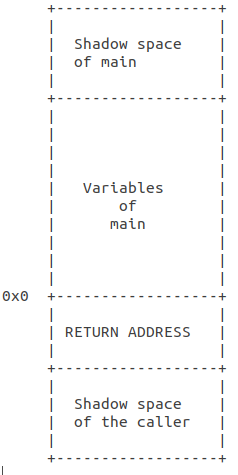
Let’s move to the main.

There we see the two arguments and we see that main stores them into the SHADOW SPACE of the caller.



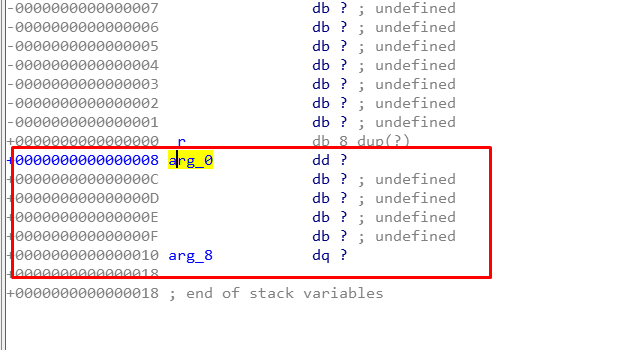
We know that the 0x0 in main corresponds to 0x38 in the caller, plus 8 bytes of the return address stored when main was called.

So the stack in main, would be something like:

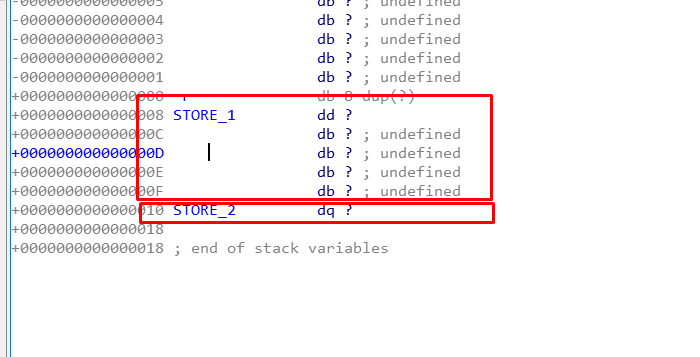


So at the beginning of main, we are in 0x0, below is the RETURN ADDRESS and below the SHADOW SPACE OF THE CALLER, and above of the RETURN ADDRESS we will have main’s variables plus its own SHADOW SPACE to call a function as the function f.

If we see the static representation of the main’s stack.



Just above of the RETURN ADDRESS is the SHADOW SPACE first argument was a DWORD and the other two were QWORDS, the third argument is not used so it’s not shown here.

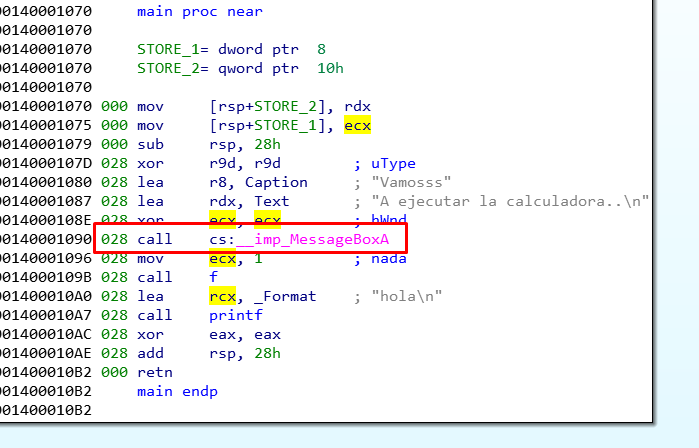


So we can rename it as STORE\_1 and STORE\_2 allocated by the caller and used in the callee.

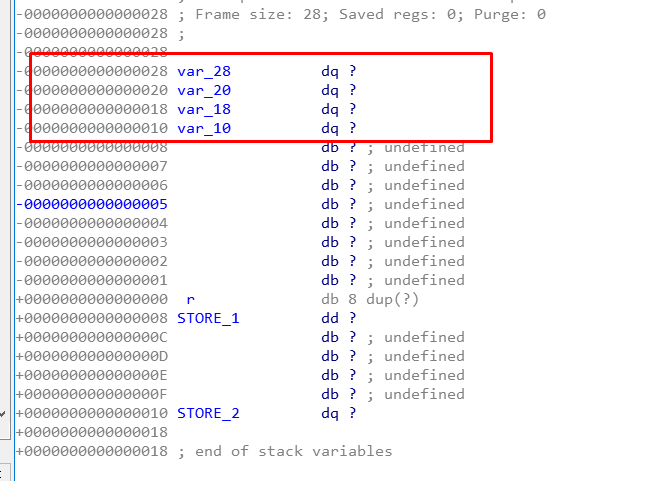
As STORE\_1 is a DWORD 4 bytes are left empty in the middle, then STORE\_2 is a QWORD and the third one that is not used.

Well, we are getting an idea of how to work with this, maybe in functions with less than 4 arguments this is not really important but sometimes in functions with more than 4 arguments, it can become harder, so it’s better not to have any doubt.

We see that with this system of SHADOW SPACE, there’s no need to balance the stack because there aren’t PUSH instructions to give the arguments, and RSP becomes constant.



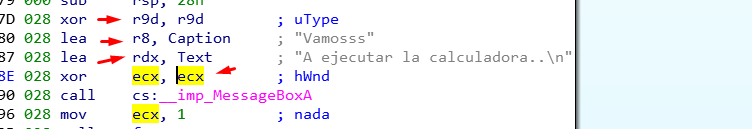
If we see the static representation of the stack, we can mark its own SHADOW SPACE.



It doesn’t have variables, only arguments that are in the registers so it stored space for 4 QWORDS its own SHADOW SPACE, so we can rename it.



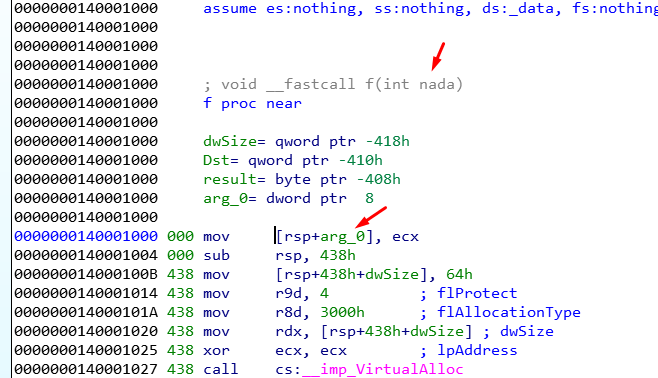




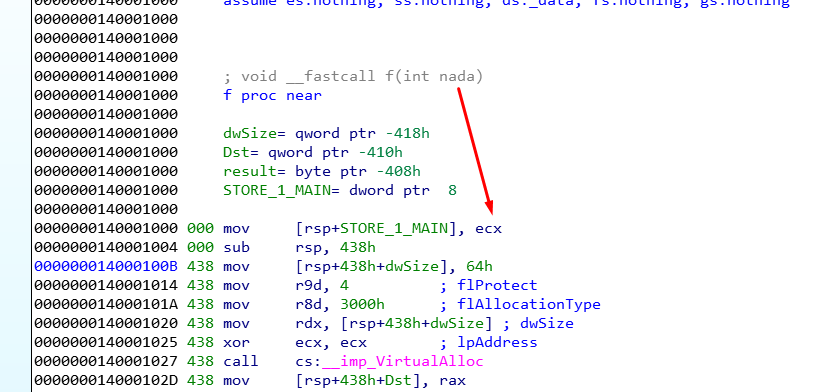
Now we can see the 4 arguments given to MessageBoxA in ECX, RDX, R8 and R9.

All the functions called from main use the same SHADOW STACK, as no two functions are called at the same time, there will not be problems.

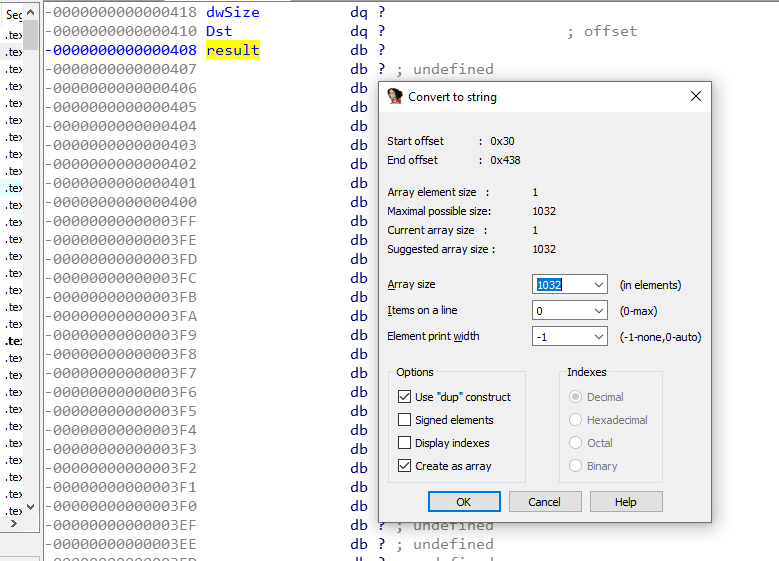
Then we get to function f, it’s a function of just one argument given through ECX and stored in the main’ SHADOW STACK.

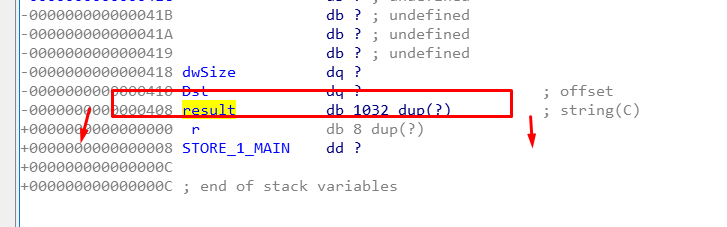


Rename it:



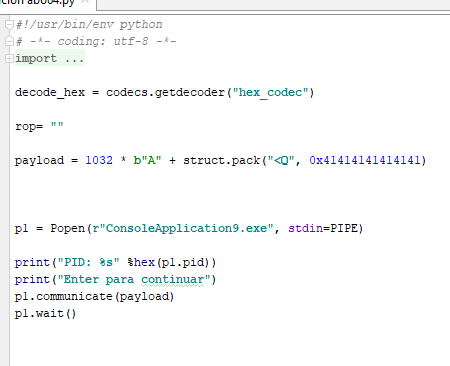
Modify **result** to become a buffer that IDA says it has 1032 bytes.



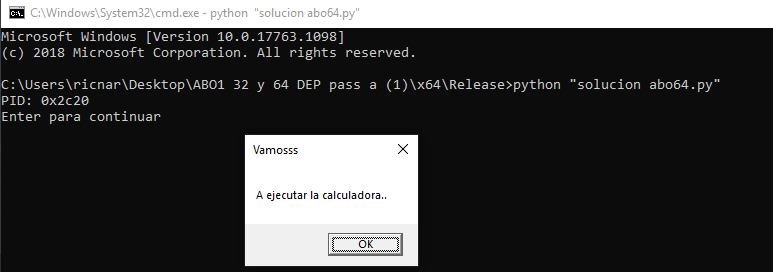


That means that when calling gets() with the argument of the address from buffer **result**, it will have to fill it up to 1032 bytes and then 8 bytes more to modify the RETURN ADDRESS.

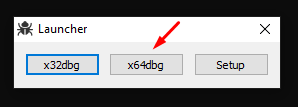
With this we would modify RETURN ADDRESS.

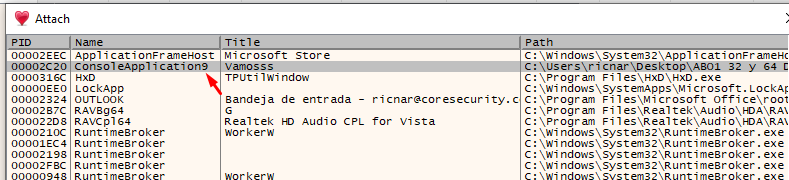


Try executing the script in the same folder as the exercise.

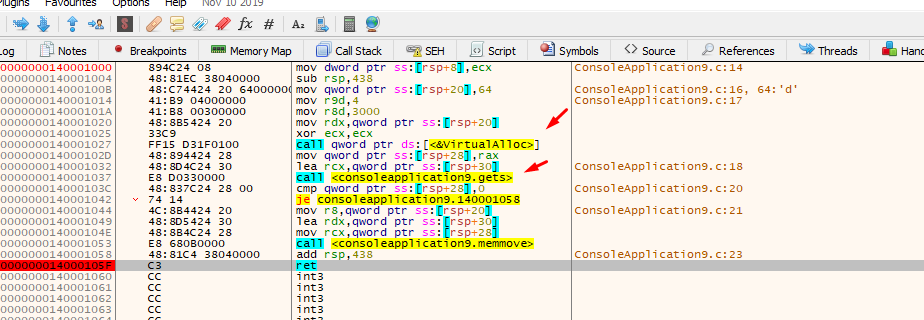


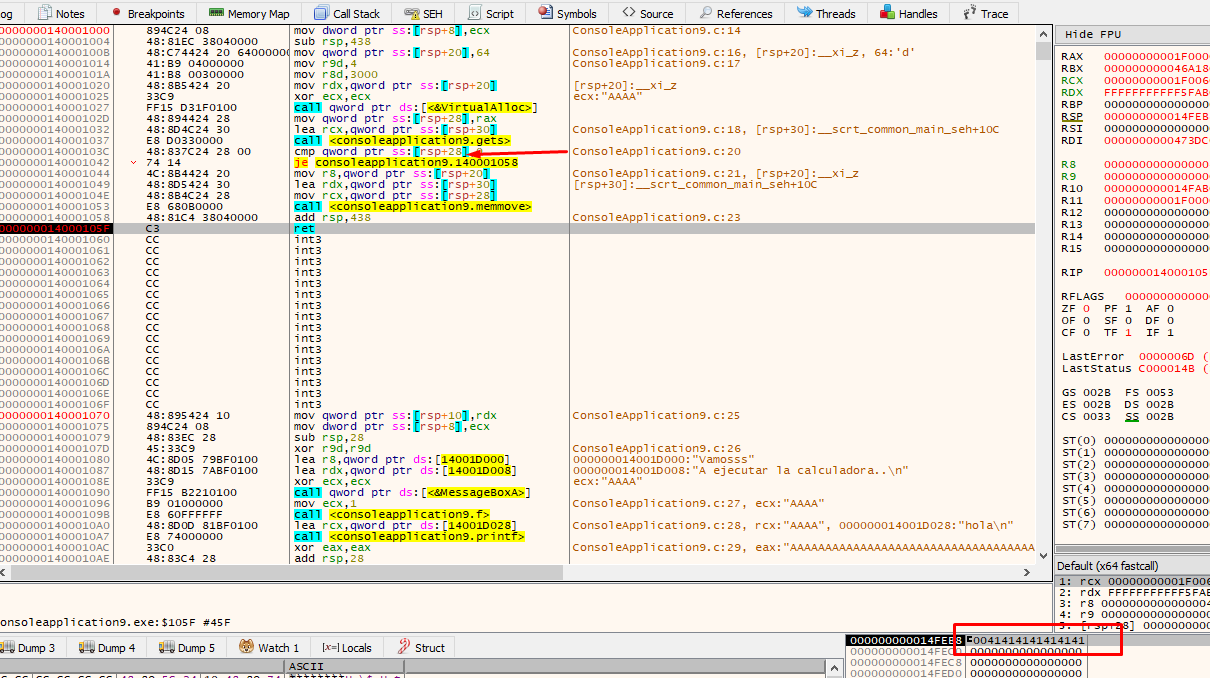
I attach the 64-bit version of x64dbg, running it as a windows administrator.



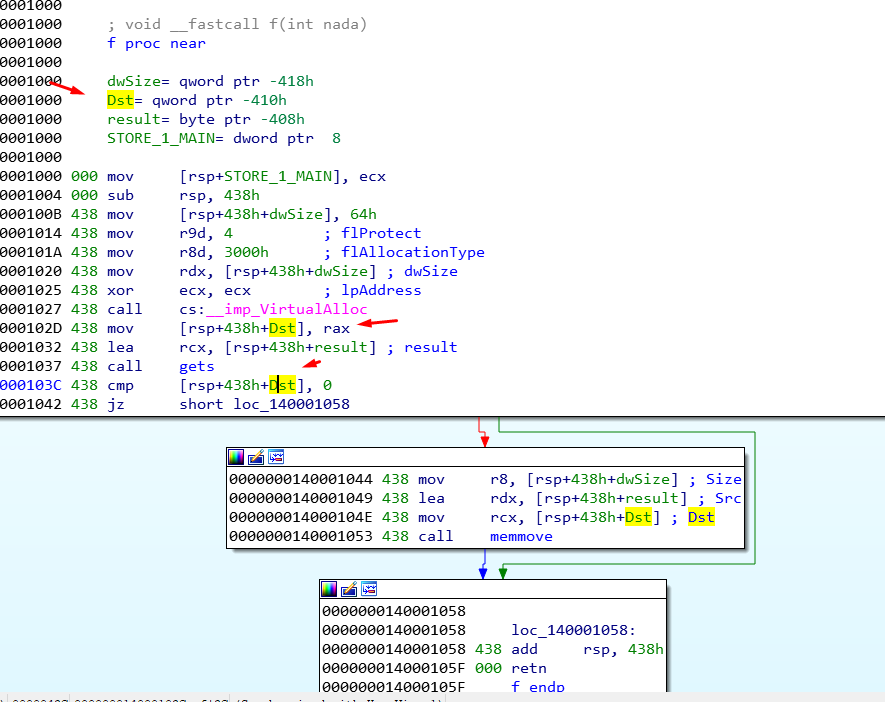


I search the RET from the module, and I set a BREAKPOINT, I accept the MessageBox and stops there:





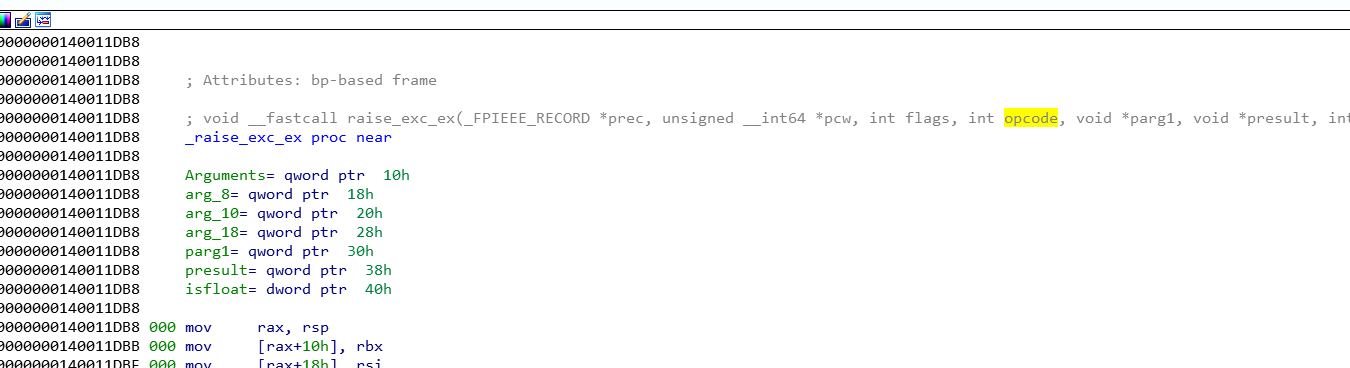
We see that in this case Dst is not zero because the variable is above of the **result** buffer, so I can’t modify it.



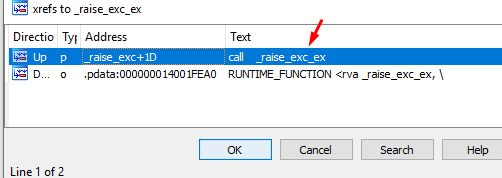
Also it will not break because **memmove** copies that buffer on the heap that was created with VirtualAlloc, the data that I sent it has the correct size so there will be no problem and we will modify the ret.

Of course, here we will have to rop to give execution permission to the stack, or to the section where the program copied the data we gave.

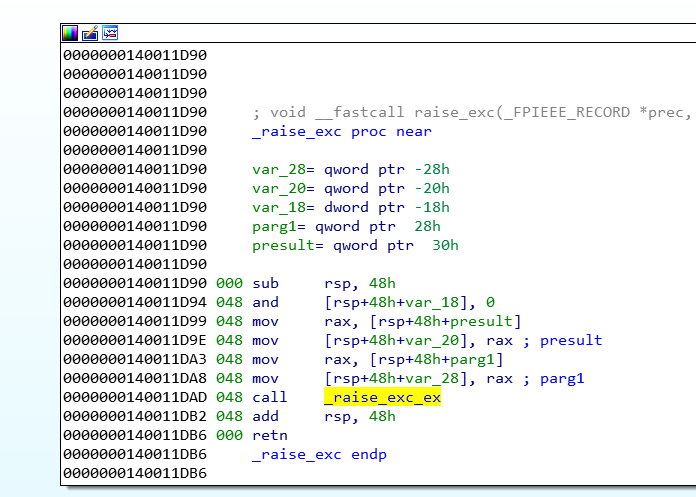
Before starting the rop I’m going to try to search in the same executable a function with more than 4 arguments, to see all the possibilities of the calling convention.



Here we have one with many arguments, let’s look for a caller pressing X.



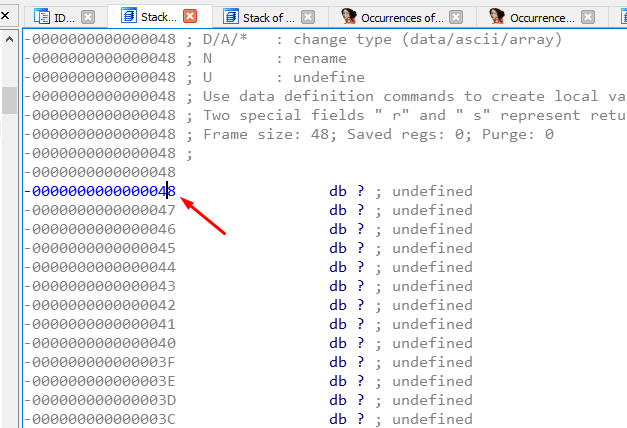
There we see.



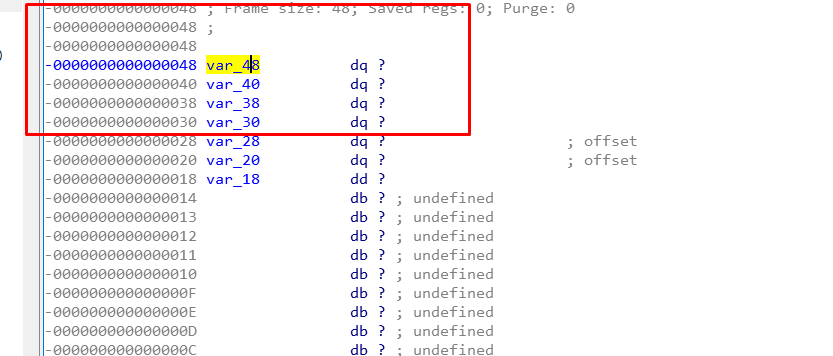
We can see how it becomes harder with more than 4 arguments.

Think in those programs without symbols where IDA doesn’t help and you’ll see it’s good to clarify and do properly reverse engineering.

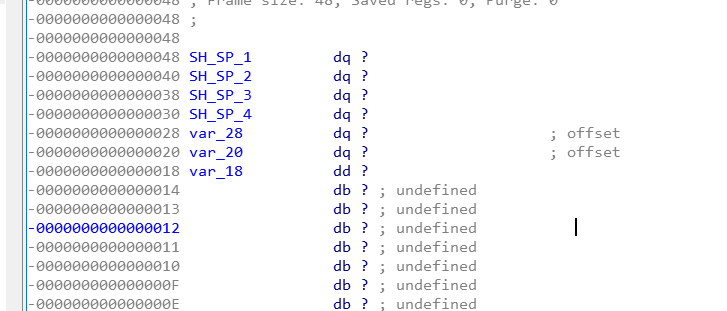
At the beginning of the function mark the SHADOW SPACE in the static representation of the stack.



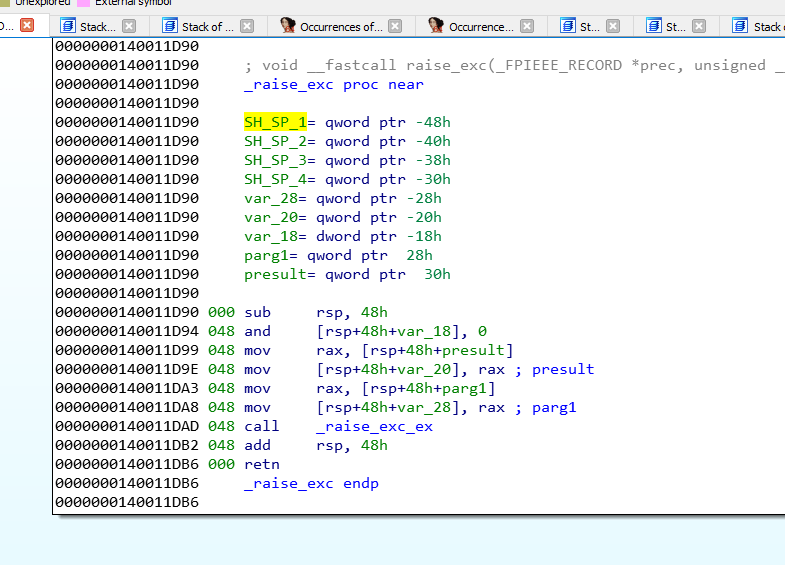
We will have something like this:



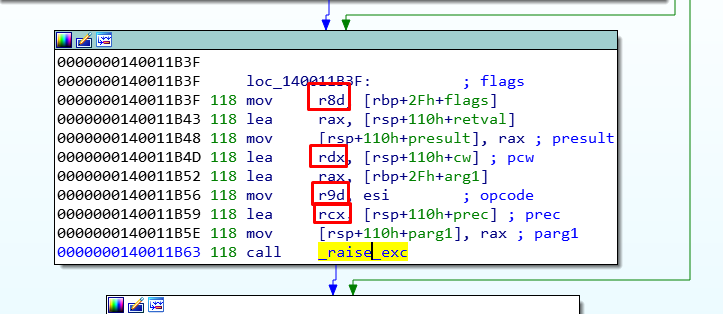
There it is the SHADOW SPACE, rename:

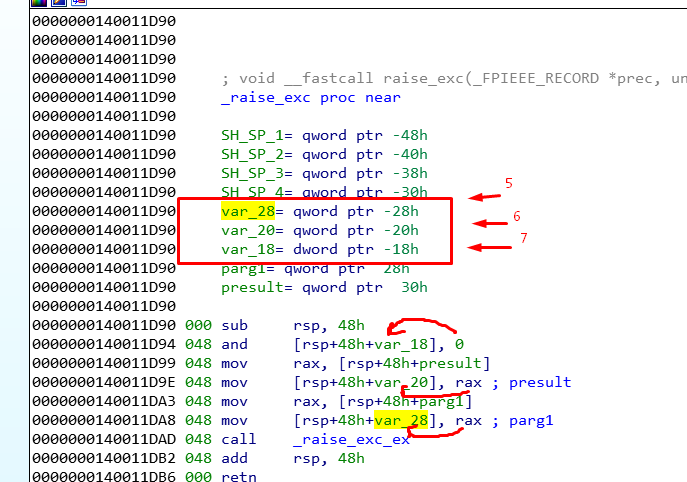


We can see that in this case, the first 4 arguments are not stored, just pass through the function:

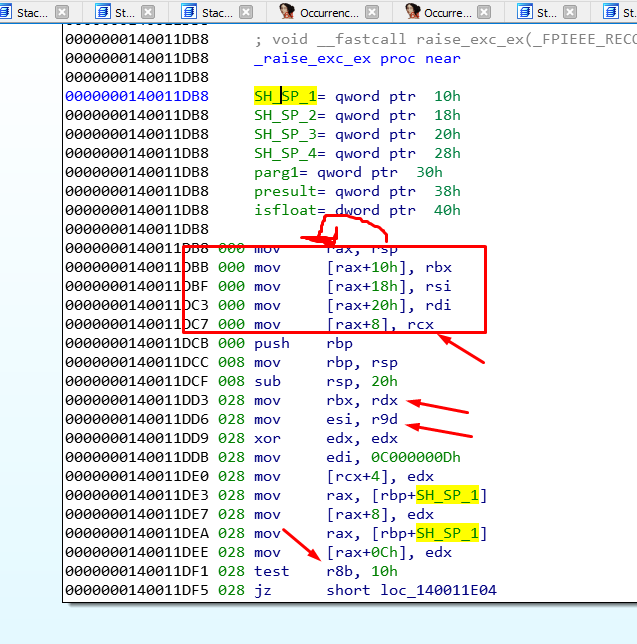


So there’s no mention to RCX RDX, R8 and R9, if we go to the caller of this function, we can see that are stored there:





4 first arguments are given through registers, and the other 3 are given down of the SHADOW SPACE.

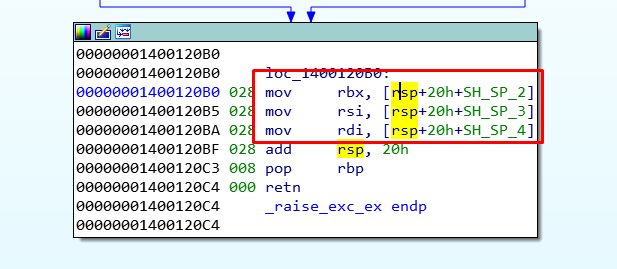


There we see the SHADOW SPACE of the caller and right down the 3 other arguments that are given through the stack, of course the first 4 are given through registers.

In this function we see that the function uses the SHADOW SPACE in a different way, just one of the registers is stored there, but the other 3 QWORDS are used to store other registers (RBX, RSI, and RDI) that are not used for arguments, but it wants to preserve them.

The three registers RDX, R9, and R8 are used directly.

We see that at the end of the function the three registers are recovered from the SHADOW STACK:

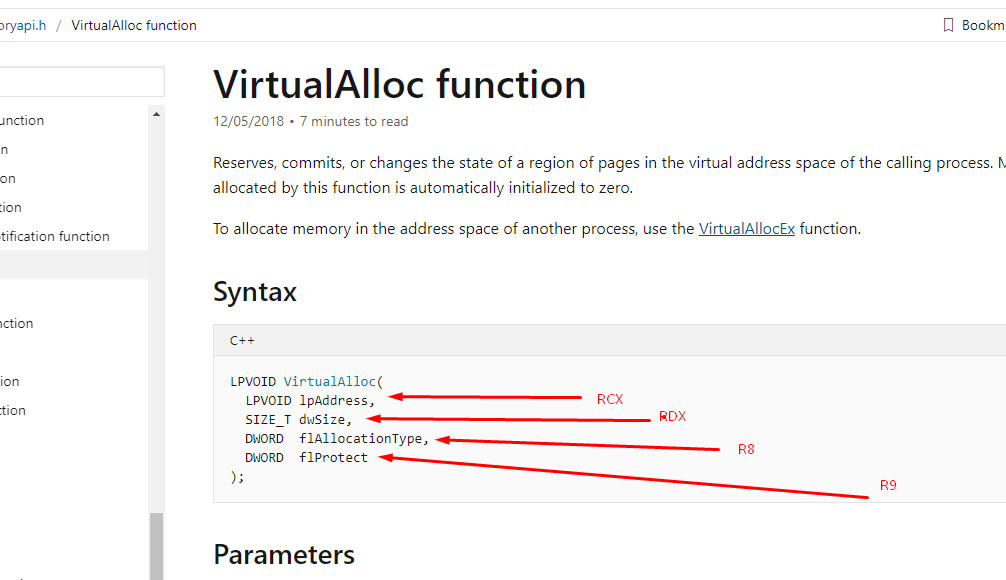


As we can see there are functions that use the SHADOW SPACE to store arguments, and others that use it to store REGISTERS to PRESERVE.

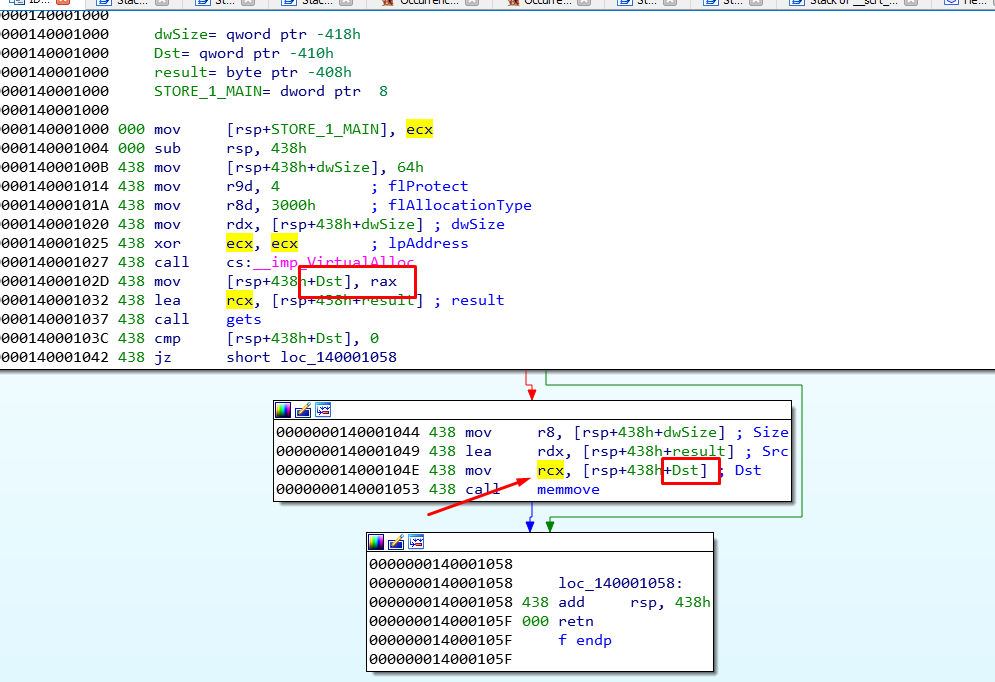
RESOLVING THE EXERCISE FOR 64 BITS.

First part of the ROP.

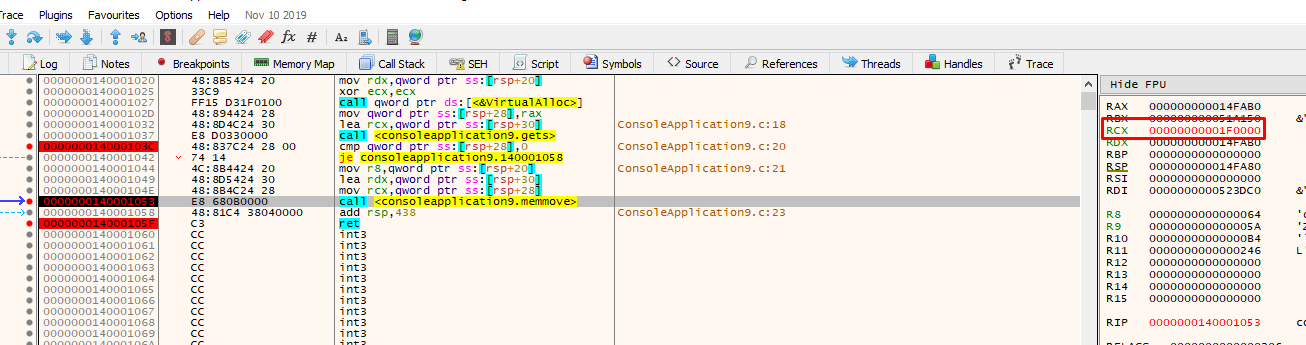
We will make it easy, remember that in RCX it goes the first argument of VirtualAlloc.



We see than in RCX the value of Dst was stored that it is where my data was saved in the heap, this value grows each time that goes through the memmove copying and returning the value that points to where it copied.



Let’s gonna see it in x64dbg, in my system points before copying to 0x1f0000 and if I step out with f8 the memmove.

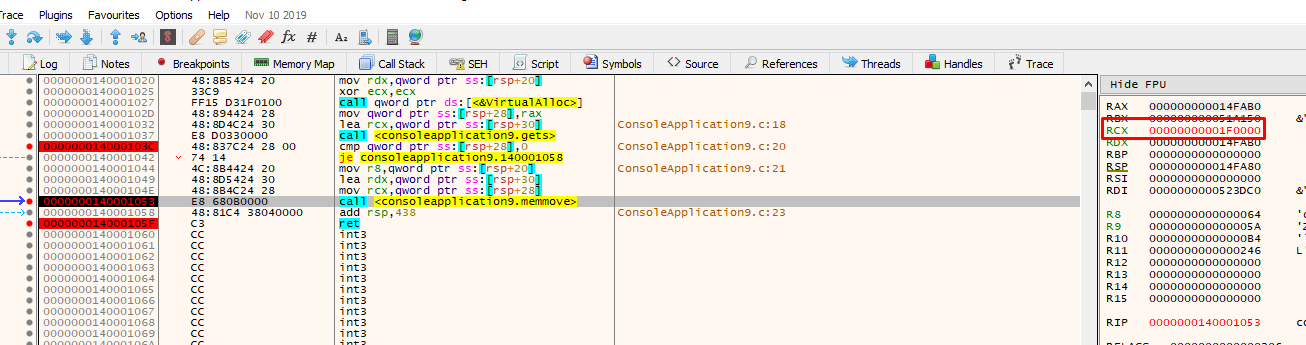


RCX points to the last DWORD that copied.

So we have the argument of the address to unprotect in RCX, many people can say that I need more than 4 bytes, but don’t forget that it will unprotect the whole section of 0x1000 (page size) and will start to unprotect from 0x1f0000 in my machine, so there will not be any problem we have the hardest part.

In RDX I have to set 1, that is the size, let’s going to use RP++ to search gadgets we have access.

I copy the new executable to the RP++ folder.

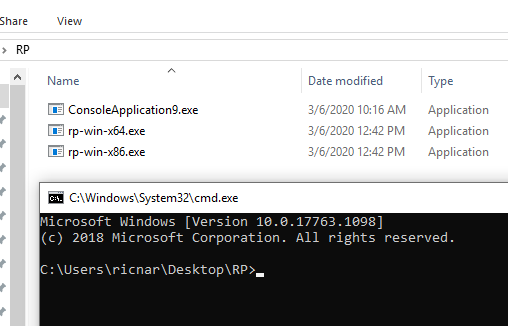


RCX queda apuntando al último DWORD que copio.

Así que ya tengo el argumento de la dirección a desproteger en RCX. muchos pueden decir que necesito más que 4 bytes, pero no se olviden que se desprotegera toda la sección de 0x01000 y empezará a desproteger en 0x1f0000 en mi máquina, así que no habrá problema ya tenemos lo más difícil.

En RDX debo poner un 1 que es el size, usemos el RP++ para ver los gadgets que tenemos disponibles.

Copio el nuevo ejecutable a la carpeta de RP++.



***rp-win-x64****.exe --file=ConsoleApplication9.exe --raw=****x64*** *--rop=4 > pepe.txt*

Let’s going to write some useful gadgets, after that we’ll see how to use them.

.Line 3144: 0x0000def5: pop rax ; ret ; (1 found)

.0x000086d6: pop rdx ; sub al, ch ; ret ; (1 found)

.Line 2485: 0x00001100: mov r8, qword [rdx] ; mov ecx, dword [rdx+0x08] ; mov qword [rax], r8 ; mov dword [rax+0x08], ecx ; ret ; (1 found)

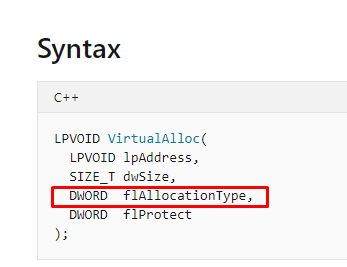
.0x00011c40: cmovne r9, rcx ; mov rax, r9 ; ret ; (1 found)

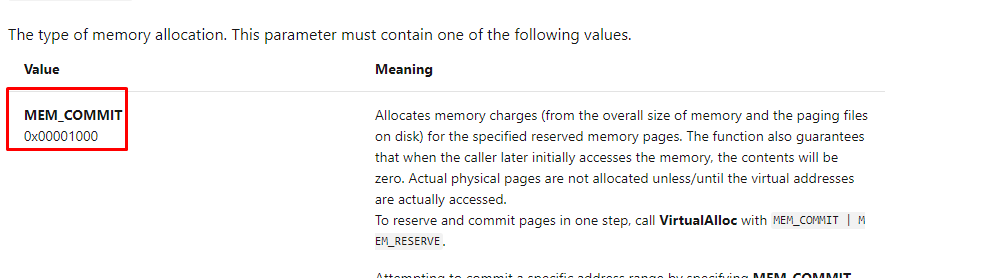
.0x00011cfd: cmove r9, rdx ; mov rax, r9 ; ret ; (1 found)

, 0x00001052: movzx r8d, byte [rdx+0x02] ; mov word [rax], cx ; mov byte [rax+0x02], r8L ; ret ; (1 found)

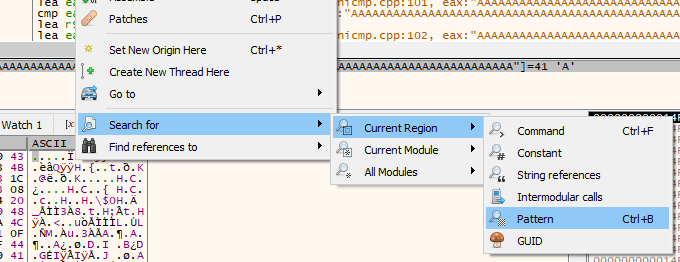
**.0x000010a2: movzx r8d, word [rdx+0x04] ; mov dword [rax], ecx ; mov word [rax+0x04], r8w ; ret ; (1 found)**

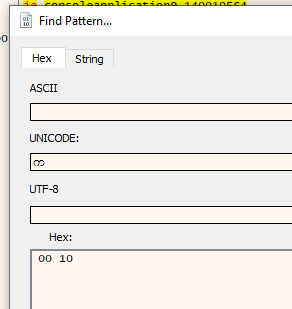
We see that the most difficult to set is r8, and there are two possibilities, we will choose the last one, because it doesn’t modify RCX, just reads a word because we need to set in r8 a 0x1000.

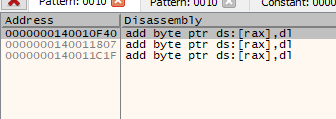


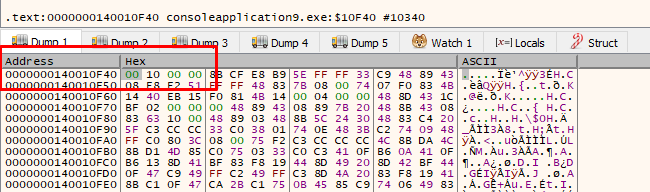


RAX points to the beginning of my data, so it’s writable and readable, we just have to set RDX to somewhere for R8 to read 0x1000, so we search for a 0x1000 in the executable.









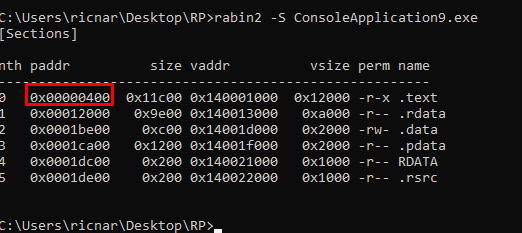
We have to write that address in RDX and substrate 4 because the GADGET adds 4 to the address in RDX.

**.0x000010a2: movzx r8d, word [rdx+0x04] ; mov dword [rax], ecx ; mov word [rax+0x04], r8w ; ret ; (1 found)**

We need a gadget more to set RDX.

0x000086d6: pop rdx ; sub al, ch ; ret ; (1 found)

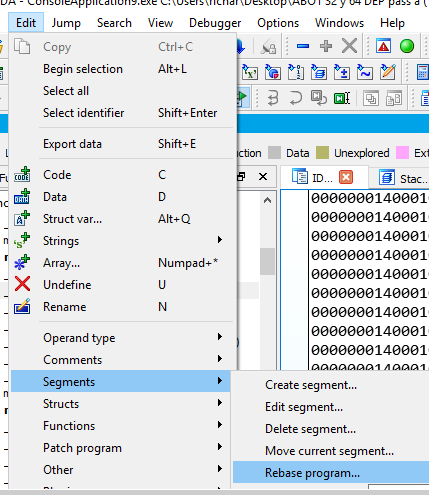
With this we could prepare the rop part to call VirtualAlloc.

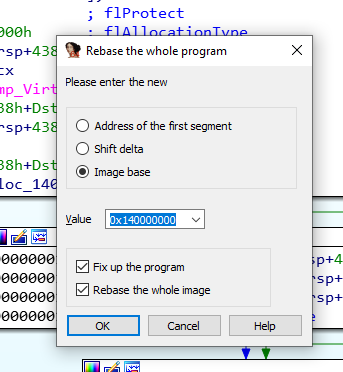


With rabin2 I see that the code section starts in 0x400 on disk, so I substrate to the value that RP++ gave me the 0x400, and then I add the image base plus 0x1000 to get the virtual address.

For example:

0x000086d6: pop rdx ; sub al, ch ; ret ; (1 found)

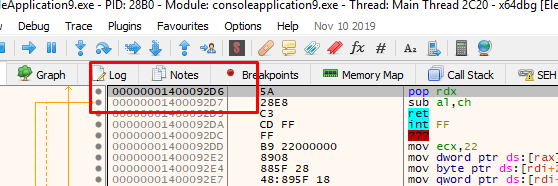




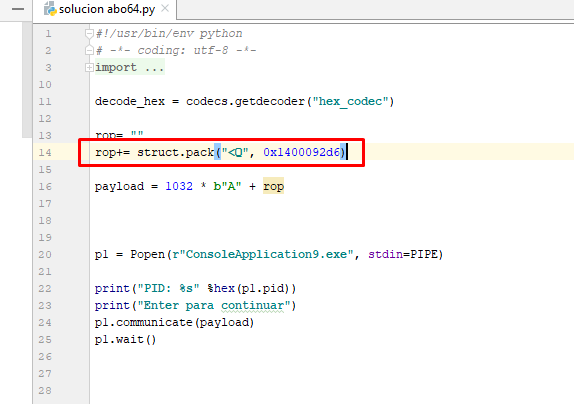
Imagebase = 0x140000000

hex(0x86d6- 0x400 + 0x140000000 +0x1000)

'0x1400092d6'



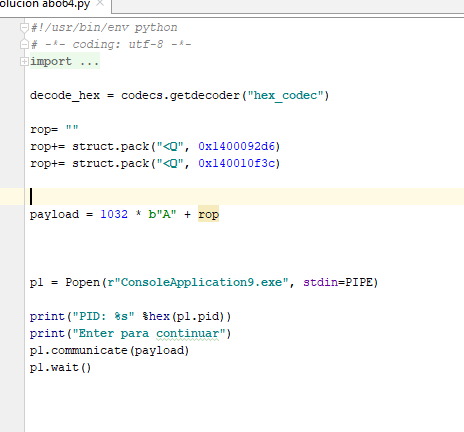
This would be the first gadget of the ROP.



We have to substrate to the value 4 for the RDX address, because of the gadget for R8:

hex(0x0000000140010F40-4)

'0x140010f3c'



Now the GADGET that moves to r8 the value 0x1000.

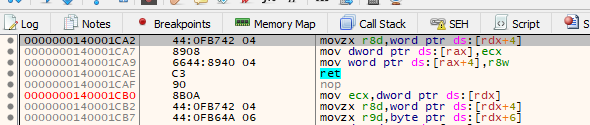
**.0x000010a2: movzx r8d, word [rdx+0x04] ; mov dword [rax], ecx ; mov word [rax+0x04], r8w ; ret ; (1 found)**

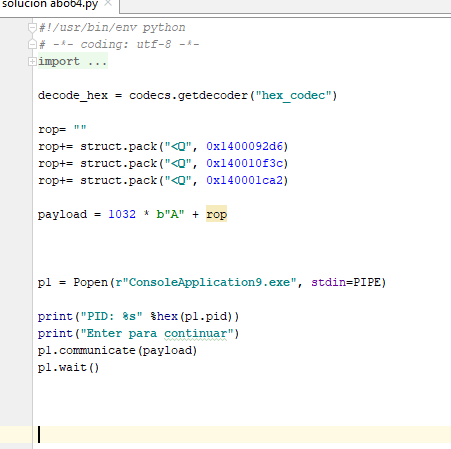
its virtual address is:

hex(0x**10a2-** 0x400 + 0x140000000 +0x1000)

hex(0x10a2 - 0x400 + 0x140000000 + 0x1000)

'0x140001ca2'

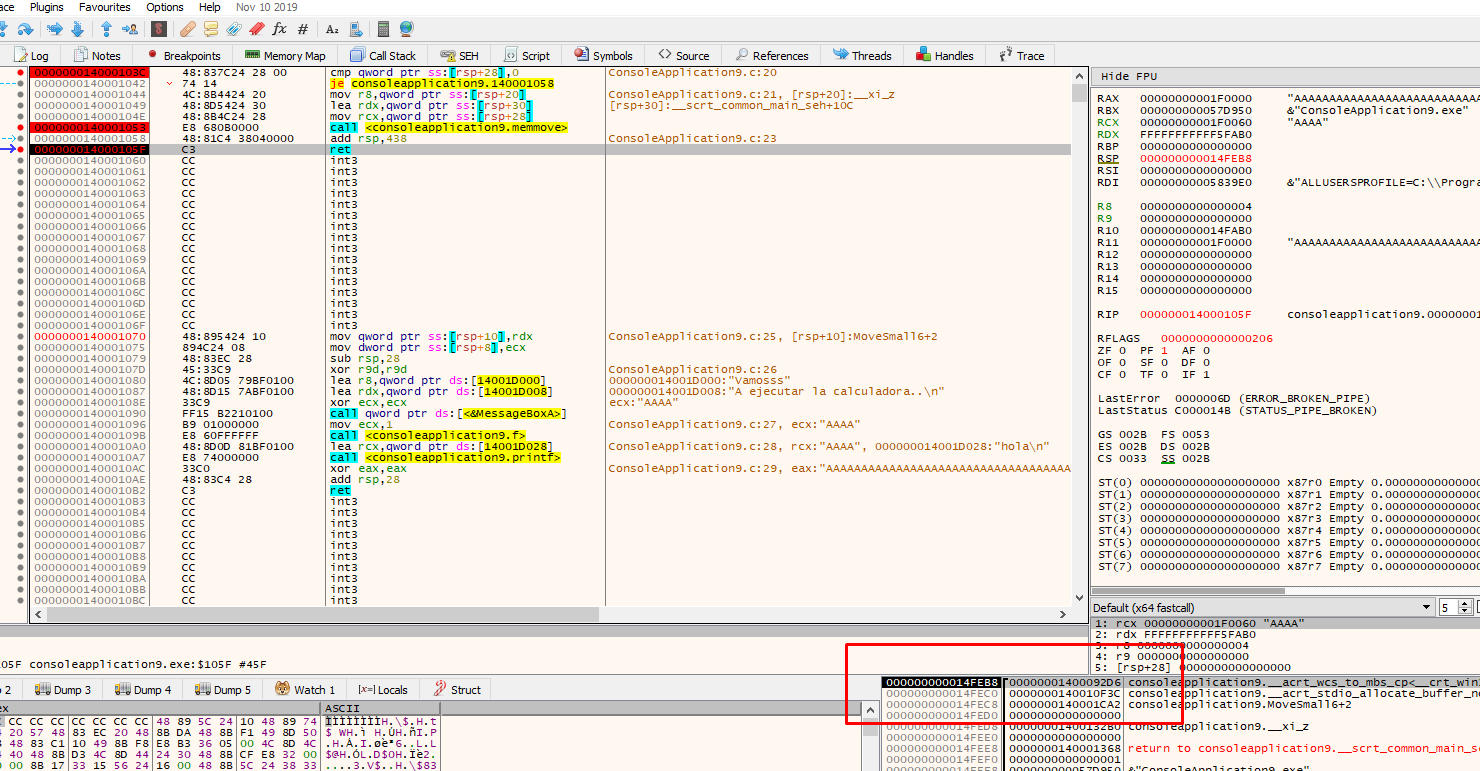




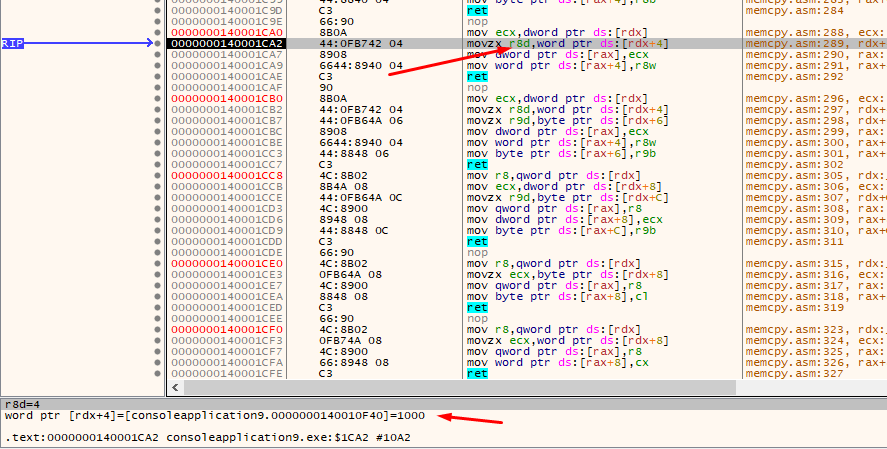
We can check if now we go well and we move 0x1000 to r8.

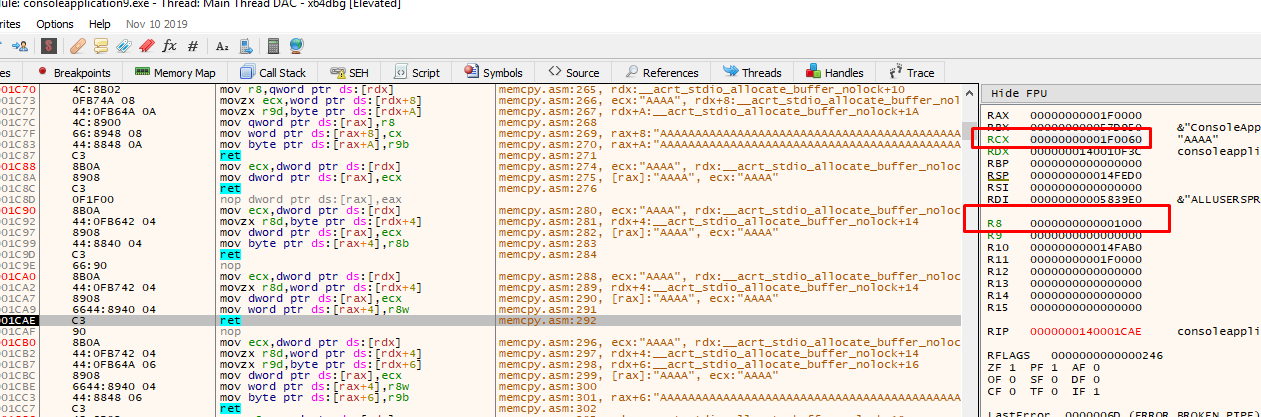


Now let’s see:



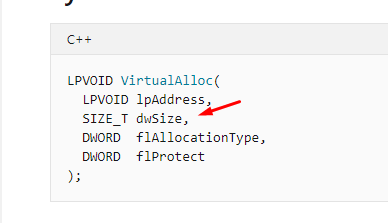
Just continue tracing:





We already have RCX and R8, we just need RDX and R9.

RDX would have 1 that is the size, this is easy.



We just need to set R9 with 0x40, that it is the flProtect.

.0x00011c40: cmovne r9, rcx ; mov rax, r9 ; ret ; (1 found)

**0x00011cfd: cmove r9, rdx ; mov rax, r9 ; ret ; (1 found)**

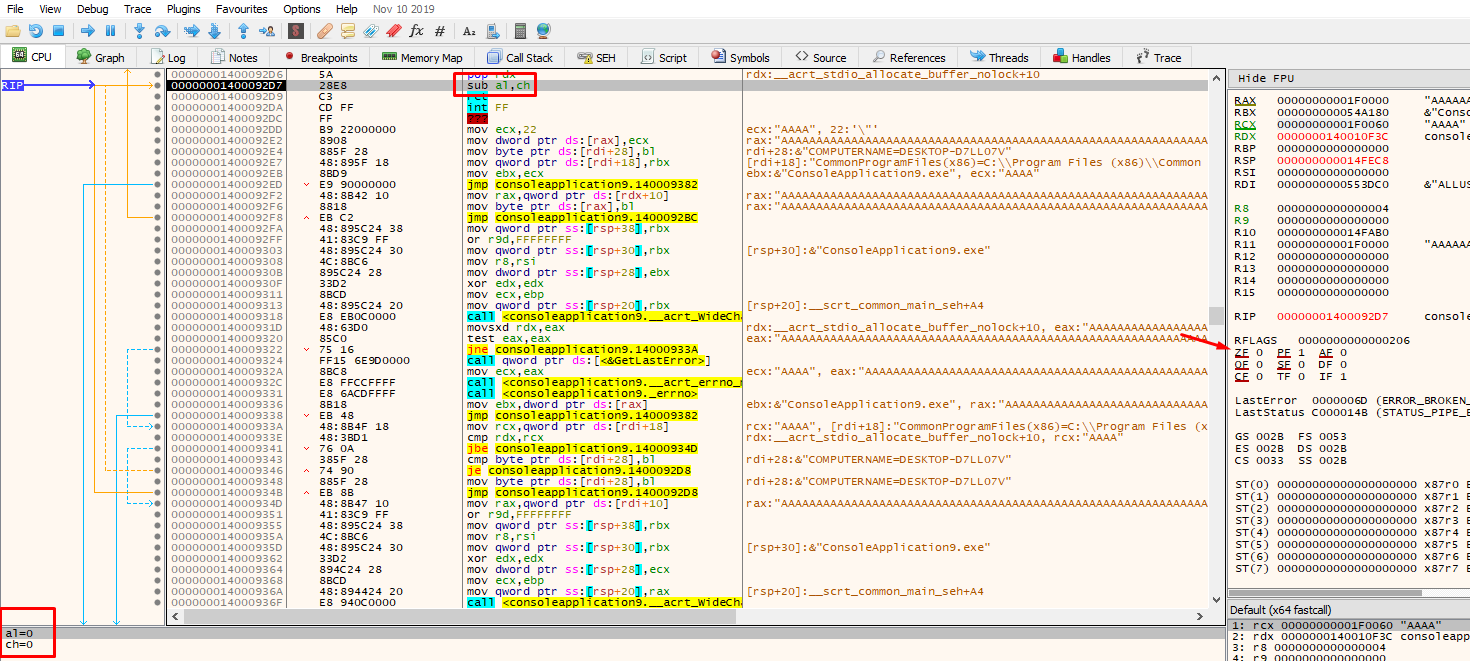
We have two, the first one breaks RCX so discarded, let’s see what happen with the second one:

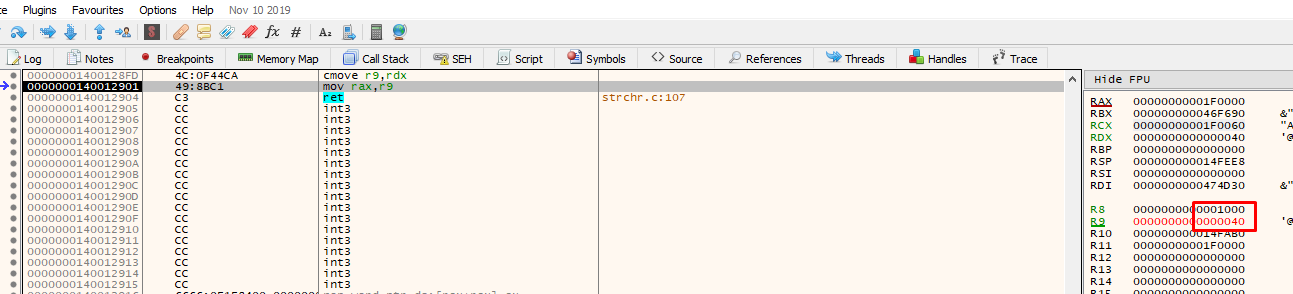
hex(0x11cfd - 0x400 + 0x140000000 + 0x1000)

'0x1400128fd'

This moves RDX to R9 only if flag Z is set, from one of the previous gadgets there was a junk substrate, and as both members are zero, the result is zero, so Zero flag is set if the operation gives zero. So everything is working properly as Messi dribbling a whole team:

<https://thumbs.gfycat.com/AmpleWelcomeAnura-size_restricted.gif>





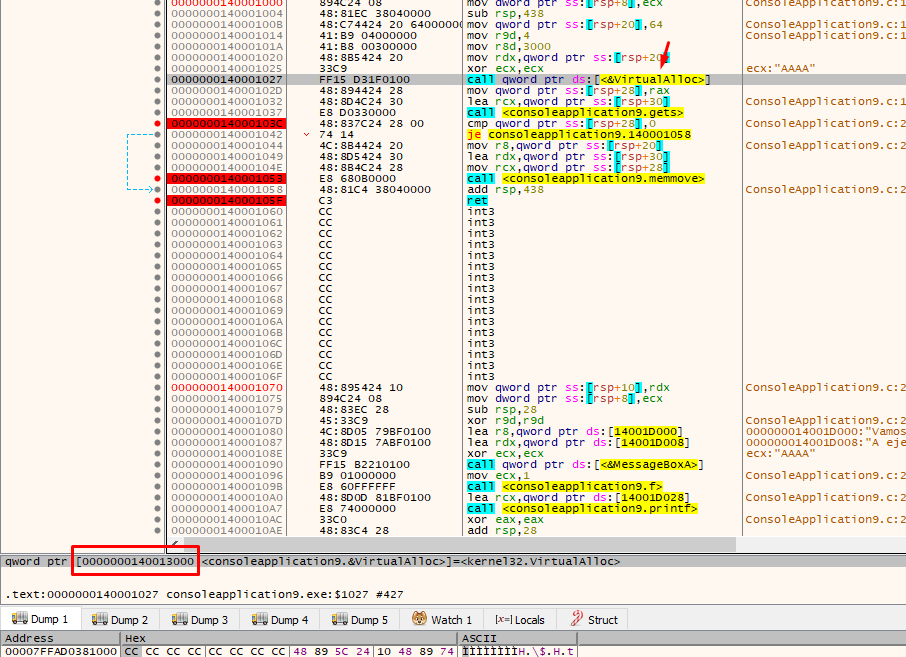
We repeat the gadget POP RDX, this time we give the value 1 for the last register we need:

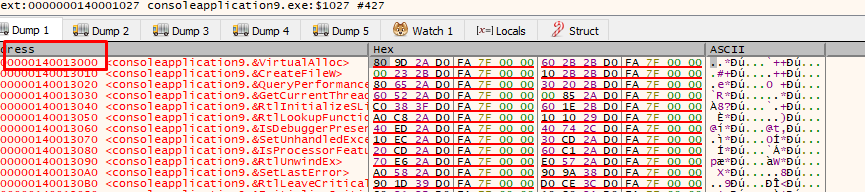


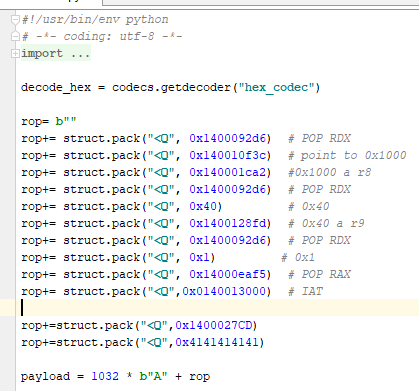
We already have all the registers set, we just need to call VirtualAlloc.

.Line 3144: 0x0000def5: pop rax ; ret ; (1 found)

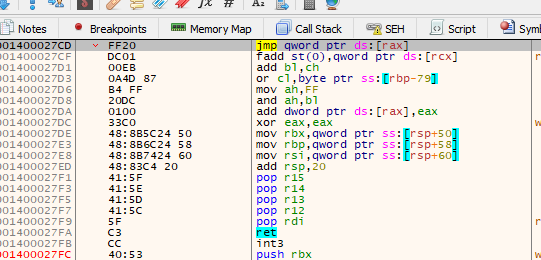
This is used to set the value of the IAT of VirtualAlloc in RAX that it is in 0x00000001400013000

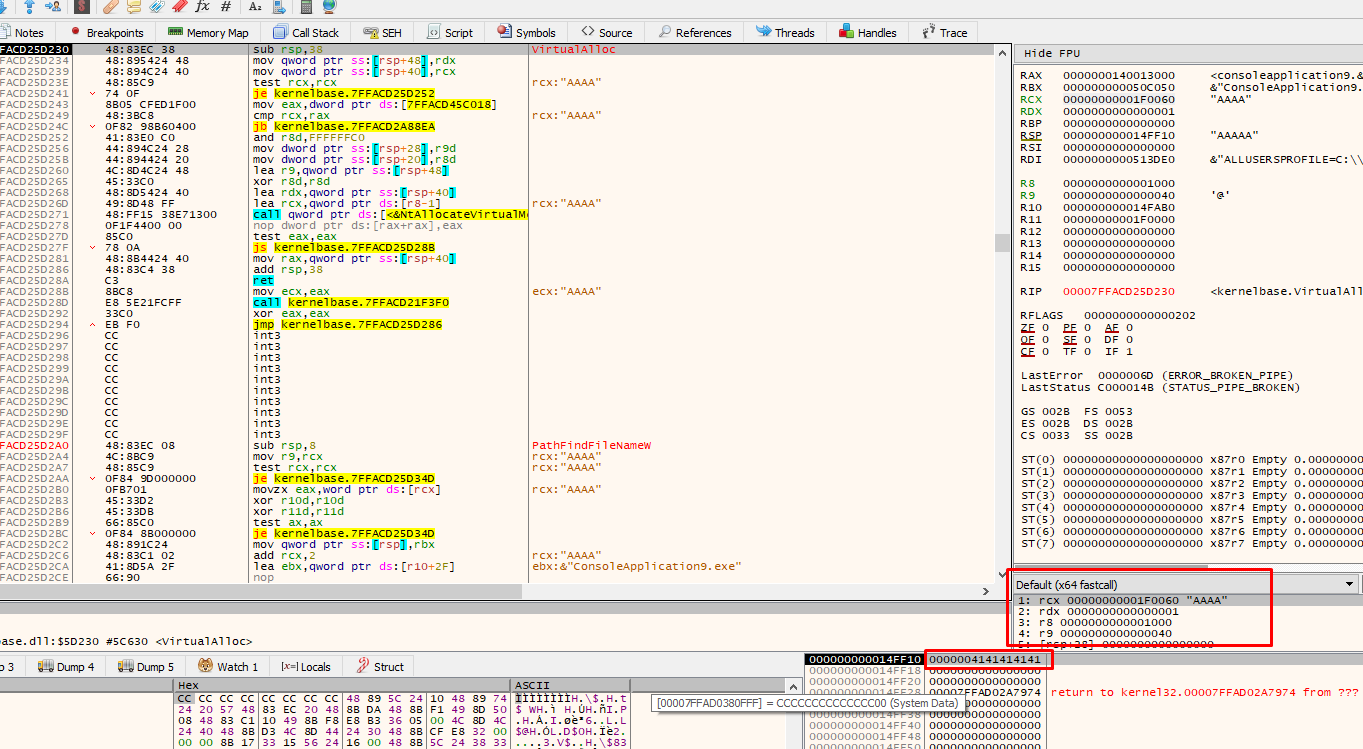




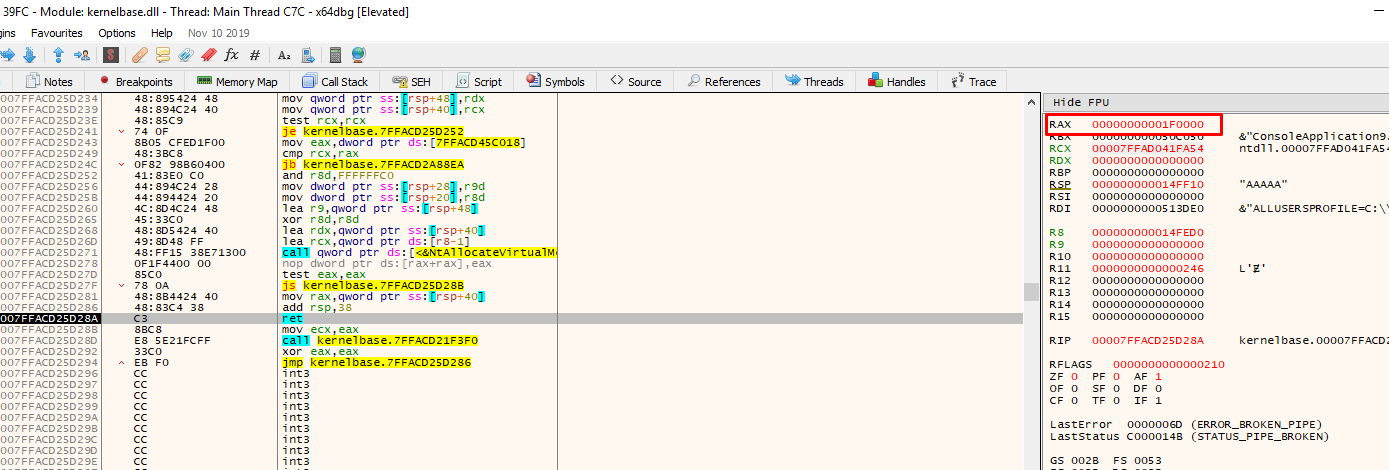


And the last gadget that jumps to VirtualAlloc.





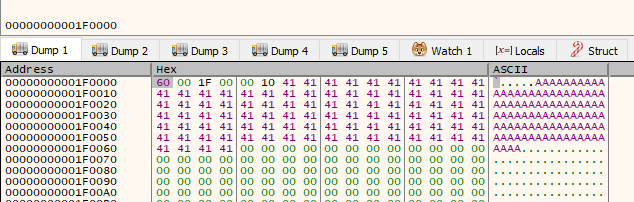
Let’s continue to the RET and trace if everything works well:



Function returns correctly the address where I need execution permissions.

We also control the RETURN ADDRESS because we jump from a JMP[RAX] and as it’s not a CALL the program doesn’t store the RETURN ADDRESS, and uses the one I left on the stack, so we just need a CALL RSP or PUSH RSP-RET and we will be executing.

Remember RAX was pointing to the beginning of my data, but something brokes there, because the gadgets wrote at the beginning, but we can resolve it.



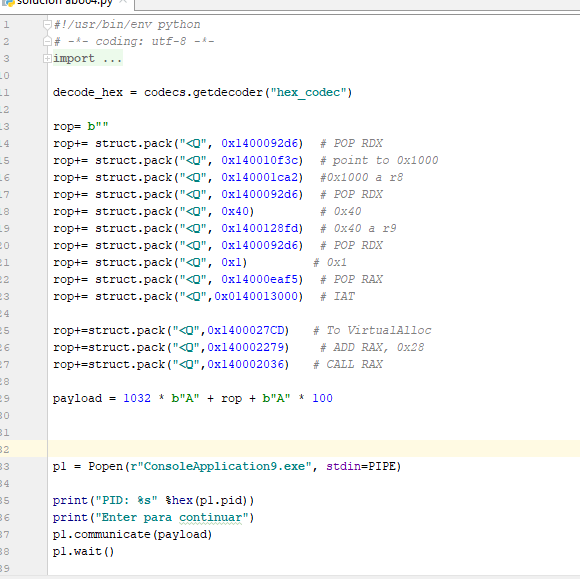
0x00001679: add rax, 0x28 ; add rsp, 0x28 ; ret ; (1 found)

0x00001436: call rax ; (1 found)

With this we will add to RAX the value 0x28 before jumping

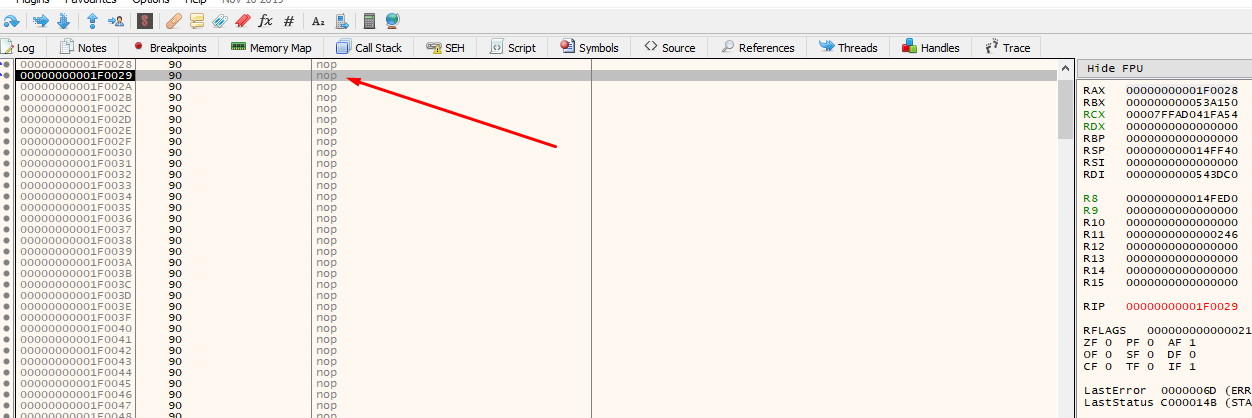
hex(0x1679 - 0x400 + 0x140000000 + 0x1000)

'0x140002279'

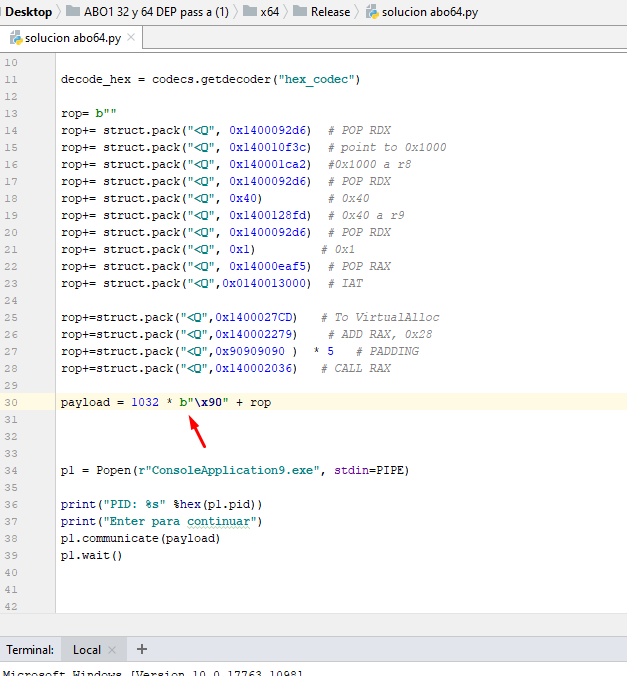


As it contains an ADD RSP,28 the CALL RAX goes lower.

So finally we’re executing, and we won to DEP.



Remember that As in 32 bits was 0x41, and it was an executable instruction, here is not so we write NOPs



We just need to add a shellcode, but we will do it in the next part where we will explain the RESOLVER in 64 bits, this part took too much.

See you in part 13.

Ricardo Narvaja

Translated by Fare9

04/12/2020