EXPLOITING and REVERSING

USING FREE TOOLS (13th PART)

RESOLUTION OF THE 64-BIT EXERCISE

We are in quarantine because of the coronavirus issue and on top of that it is a long weekend so there are 4 very boring days that you can't get out, so yesterday I did part 12 and today I started part 13, added the shellcode and analyzed and adapted a resolver for 64 bits.

I have to explain that the shellcode is not mine but it is quite public and I adapted it for the case.

COMPLETE SOLUTION SCRIPT

For a little variation here's the complete solution that runs NOTEPAD for a change.

Let's see how it works.



There we see the parts that I added to the partial solution that existed until now.

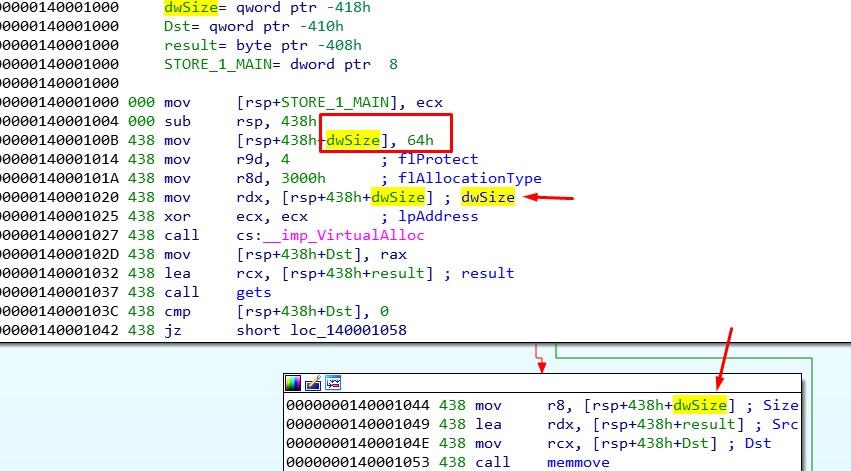
Above we see the shellcode with the resolver that we will explain later.

There is a gadget that I have changed

rop+=struct.pack(**"<Q"**,0x1400060b7) *# ADD RAX, 0x20 # CHANGED to have more space*

This gadget was adding 0x28 to RAX before running so I changed it for a similar one but adding 0x20 so as not to waste because there is little space.

Remember that the memcpy does not copy everything we send to the space reserved in the heap with VirtualAlloc.



We see that it only allocates and copies 0x64 bytes so it is convenient to avoid wasting space.

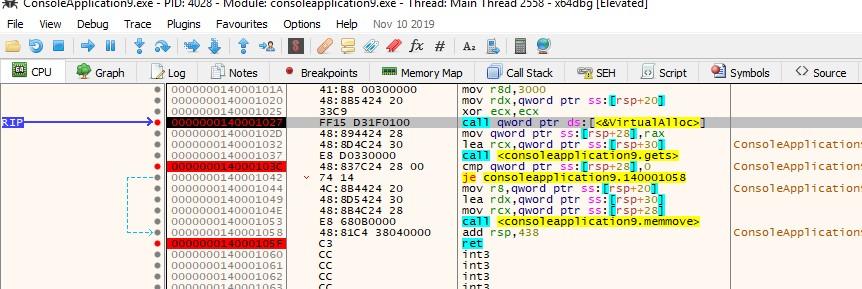
We see, in a code made by me, that the memmove is going to copy in the reserved area created in the heap and it is the one that is going to prepare everything to execute the final shellcode, we will see it when executing.

And the shellcode is in the stack below the ROP without execution permission, you could ask me why I don't put it to the shellcode in the heap, instead of that code directly and execute it.

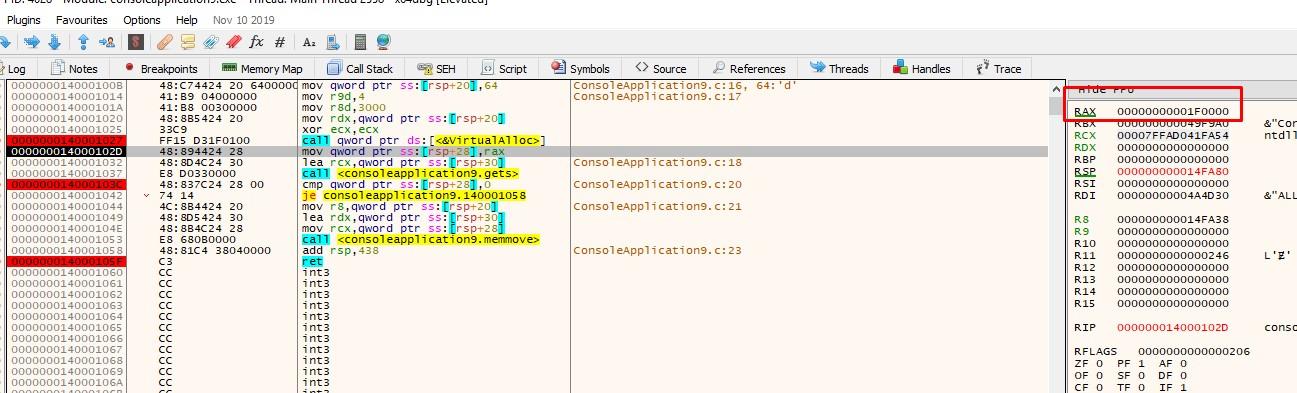
The answer is because it doesn't fit, we already saw that the space in the heap it only reserves 0x64 bytes and we also lose the first 0x20 when we jump, we have 0x44 bytes left and it doesn't fit in that little space.

Let's run it and let’s explain it.

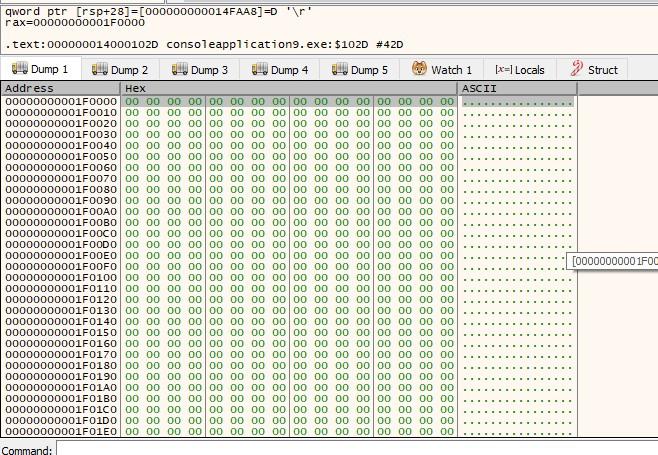
I run the script, it stops at the breakpoints I put.

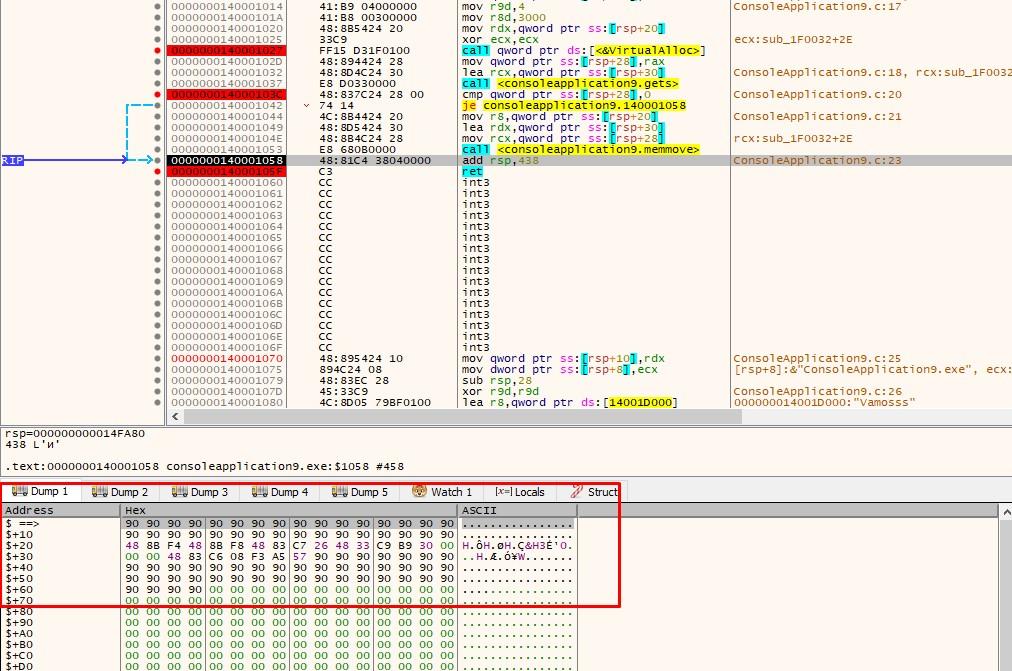


There I allocate the 0x64 bytes.



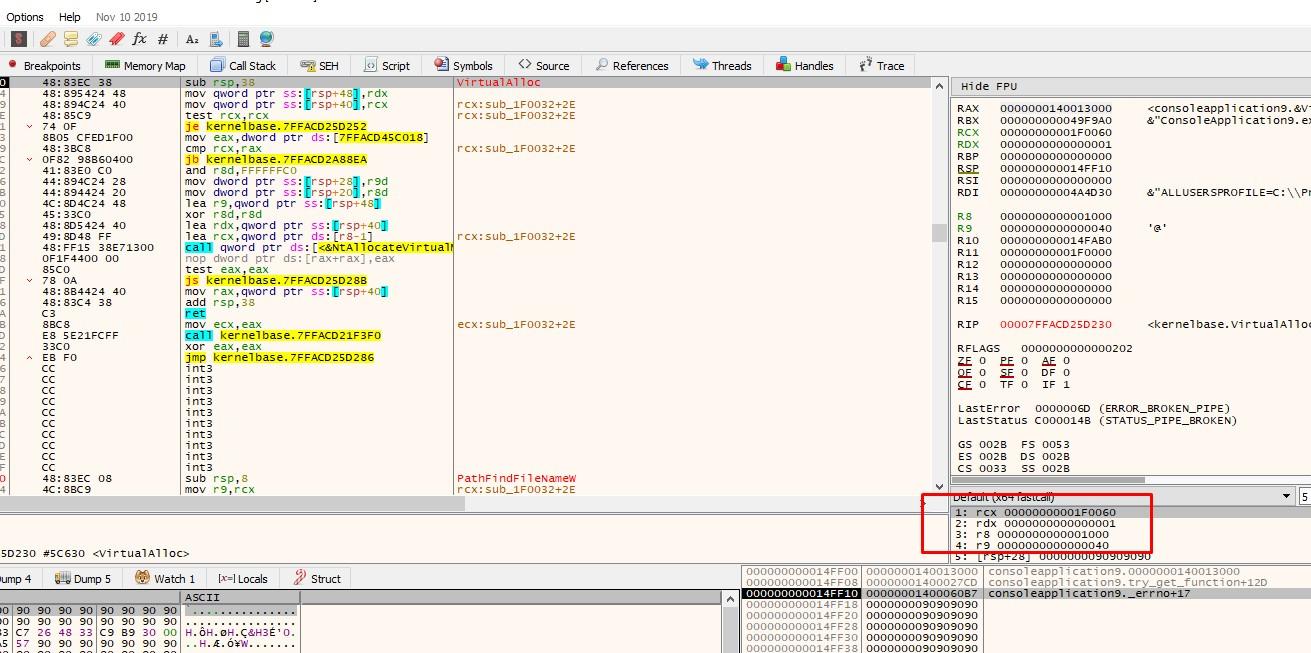
We see that it actually allocates more than 0x64, the problem is that it strictly copies only 0x64 bytes.



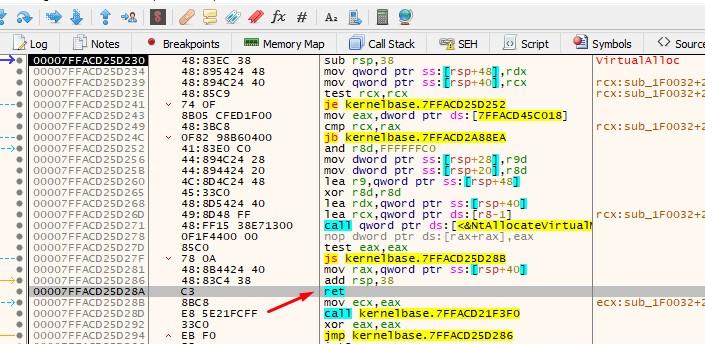


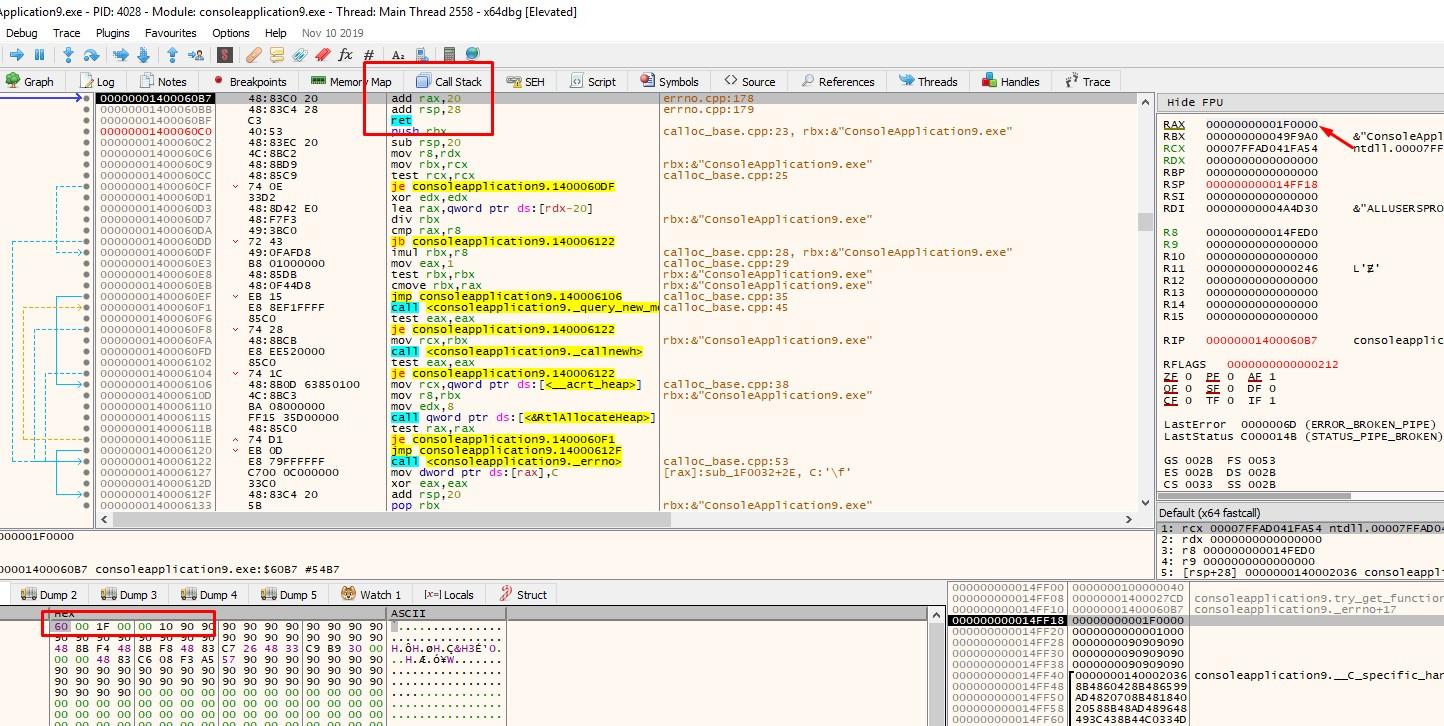
There I copy, let's not forget that the rop, breaks some bytes from the beginning and to jump I have to avoid them and the only thing I found is the gadget add rax, 20, which leaves me little space for the shellcode.

I traced the shellcode to VirtualAlloc.



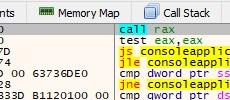
There I arrive with the correct arguments, I can go to the RET, put the cursor there and press F4 to not trace so much.

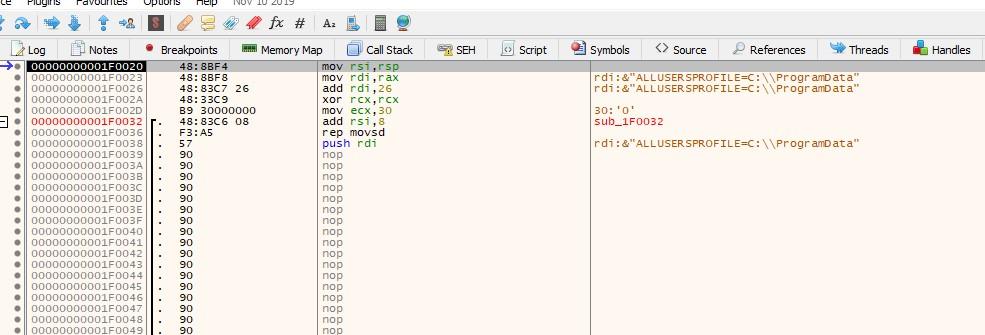




This is the GADGET that adds 0x20 to RAX to jump to execute, avoiding the bytes that were broken at the beginning.

Then we jump to execute.





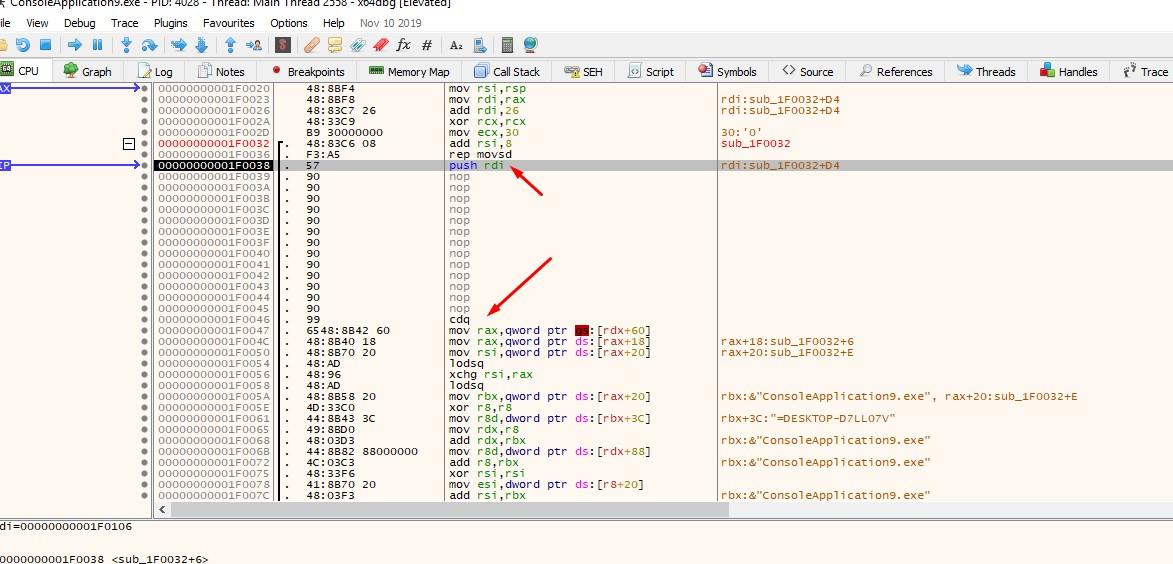
There we see the code that I made, what it does is to look for the shellcode in the stack, and to copy it here below since place is, the **memmove** did not copy it completely by the limit of size=0x64 when copying, but I can copy it completely.

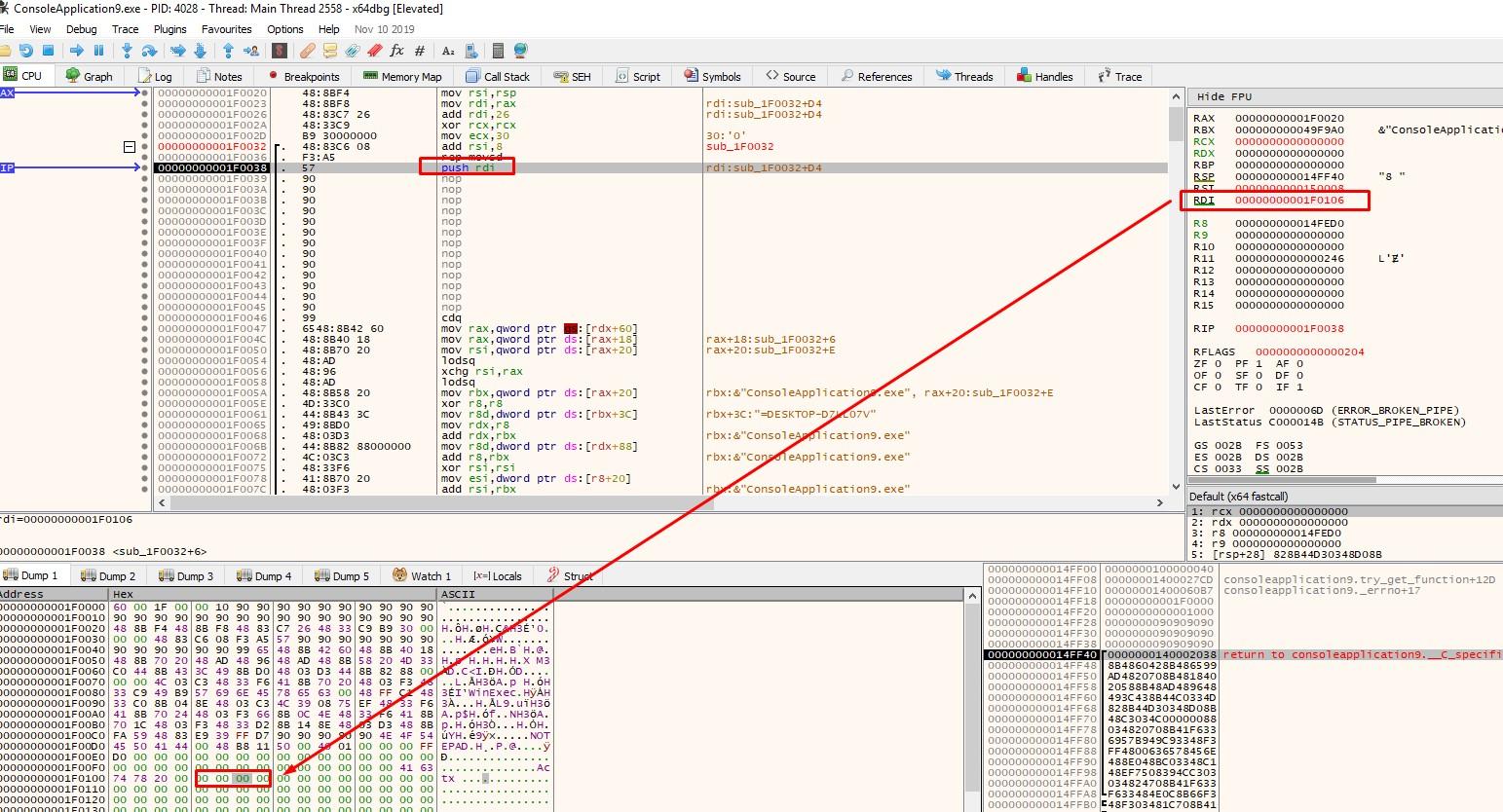
RSI will have the source

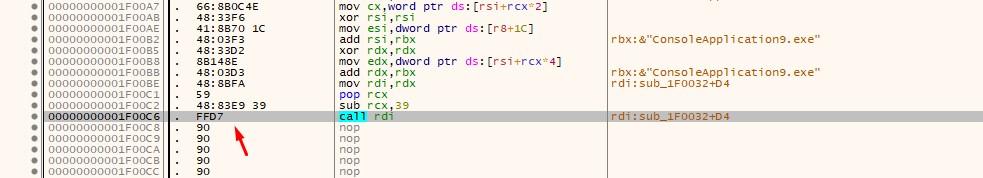
RDI the destination

RCX the size to copy in dwords

When I get to the REP MOVSD, I copy the shellcode and do a PUSH RDI to save the address of the destination where I copy, which is in RDI.





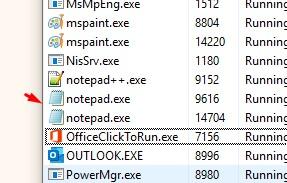


I can put the cursor there and press F4.

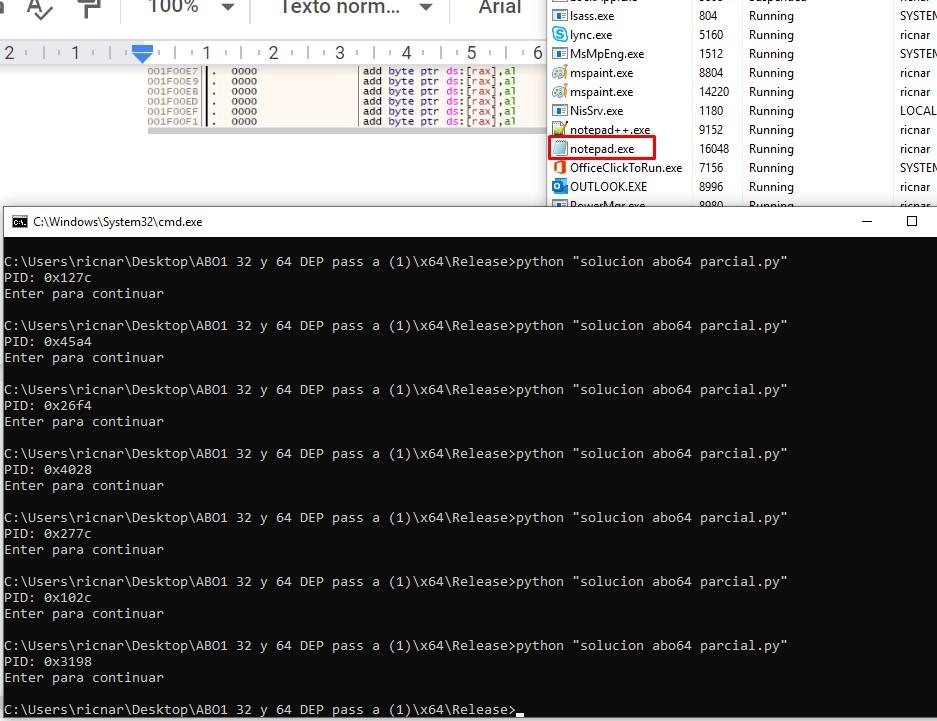


We see that you are going to call WinExec with the string NOTEPAD as an argument, which will execute the NOTEPAD.

Well, I already have the shellcode to execute, I just have to explain it, which is not very good in x64dbg, so I will run the script and attack with Windbg so I can see the structures and symbols needed, even if I give RUN first I verify that it runs a NOTEPAD.

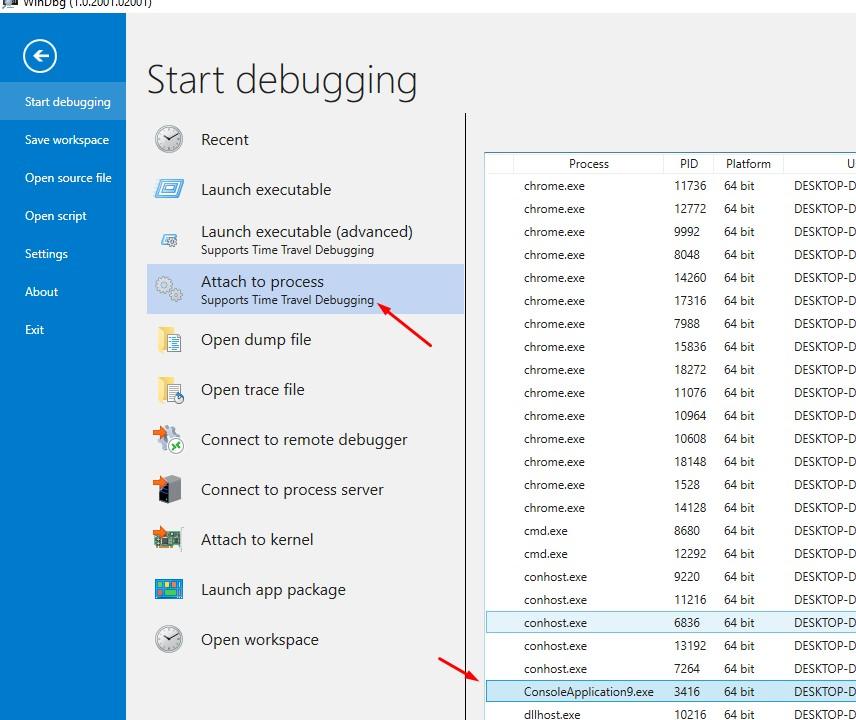


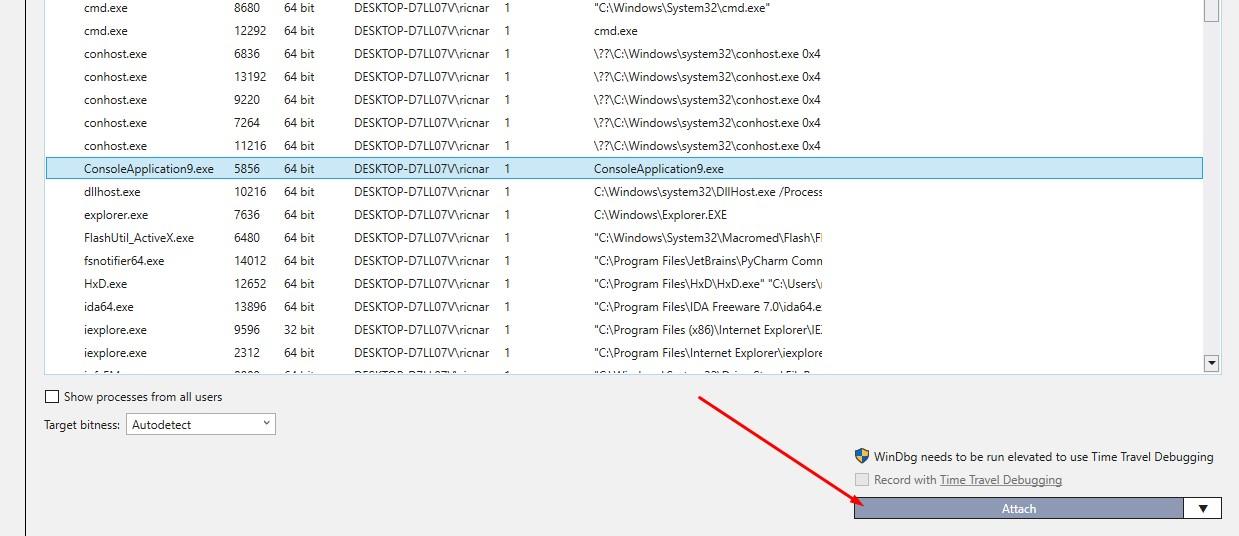
We see that the NOTEPAD runs and then goes to ExitProcess and closes



If I run it without debugging I see that it runs NOTEPAD and closes correctly.

Well, I'm going to attach it with the Windbg to trace the RESOLVER.



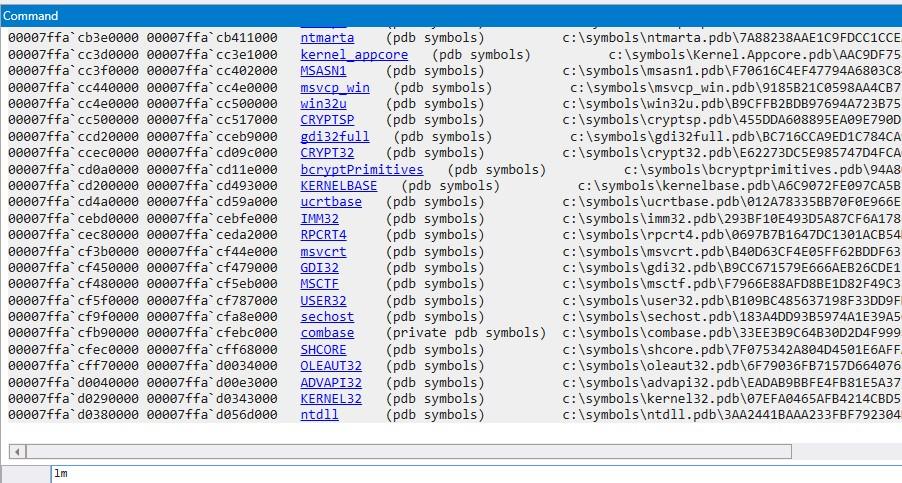


After it stops

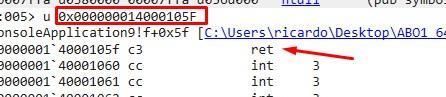
.reload /f

and then it finishes downloading all the symbols.

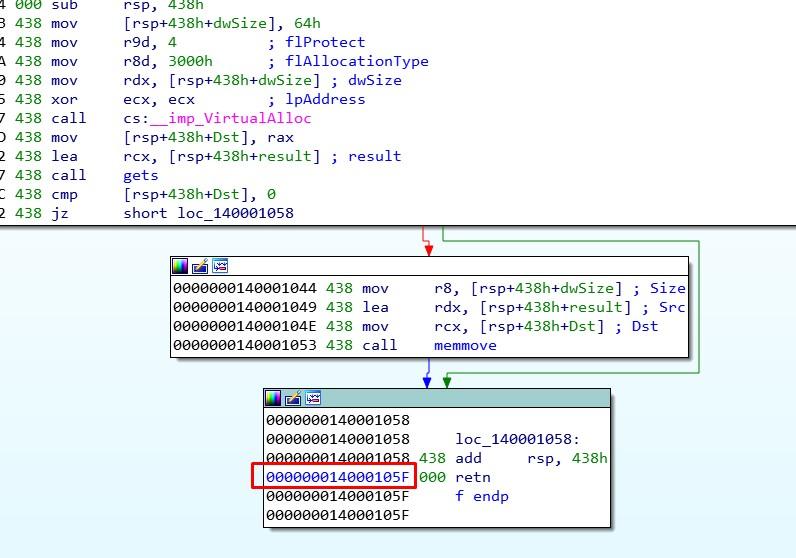
lm

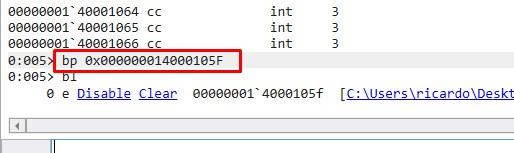


Well, I have the symbols.

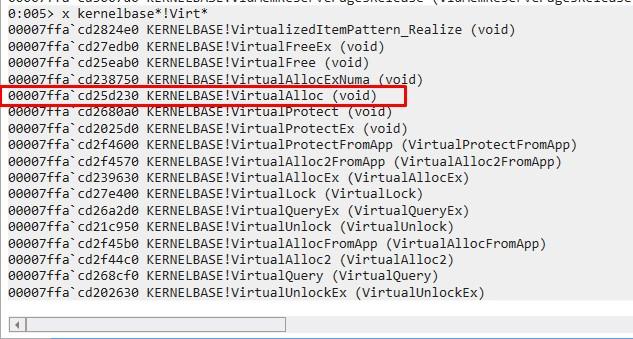


I put a breakpoint there.



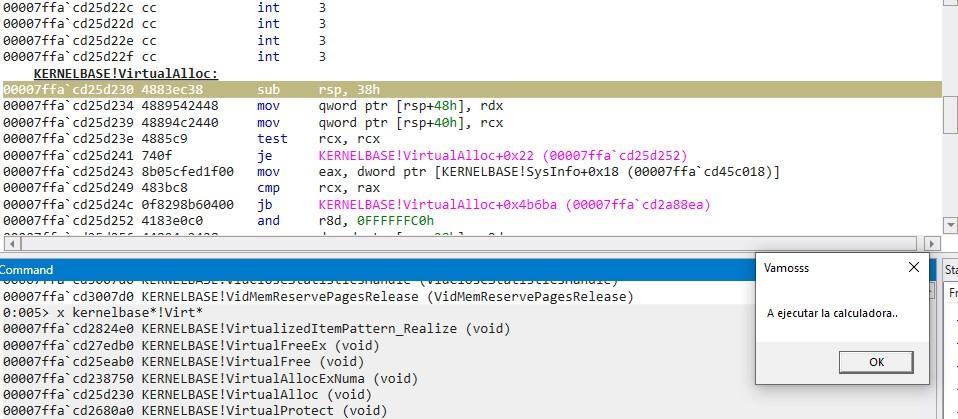


Well now to trace to the shellcode, I can put a Breakpoint in VirtualAlloc



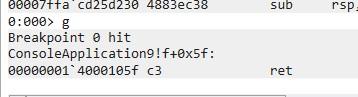
bp KERNELBASE!VirtualAlloc

I give RUN with G and accept the MessageBoxA and stops when the program calls VirtualAlloc at startup, I continue with G.

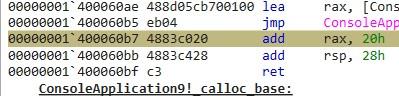


Stops again in VirtualAlloc but first we must stop in the RET and the one that counts is the following stop in VirtualAlloc we continue,

There I stop at the RET

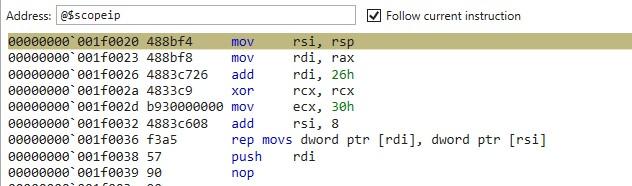


I press G again and now it stops in VirtualAlloc, SHIFT plus f11 is STEP OUT to exit the function just after the RET.



I trace with f11.

We get to my code.



I keep tracing with F10 to pass the REP MOVS so it doesn't repeat.

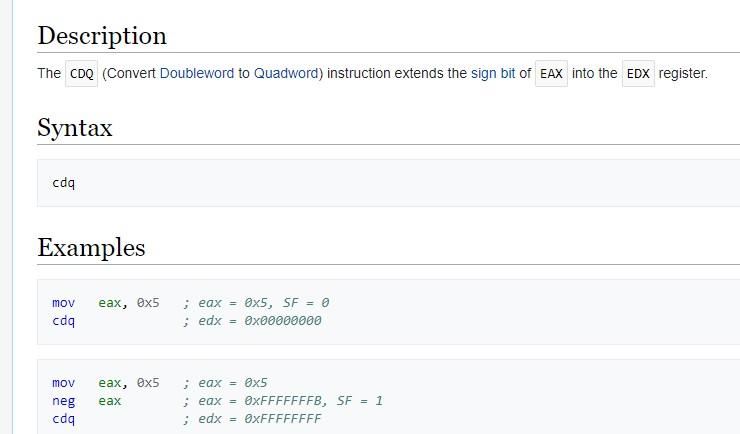
Here begins the SHELLCODE RESOLVER.

RESOLVE 64 BITS

FINDING THE KERNEL32 IMAGE BASE



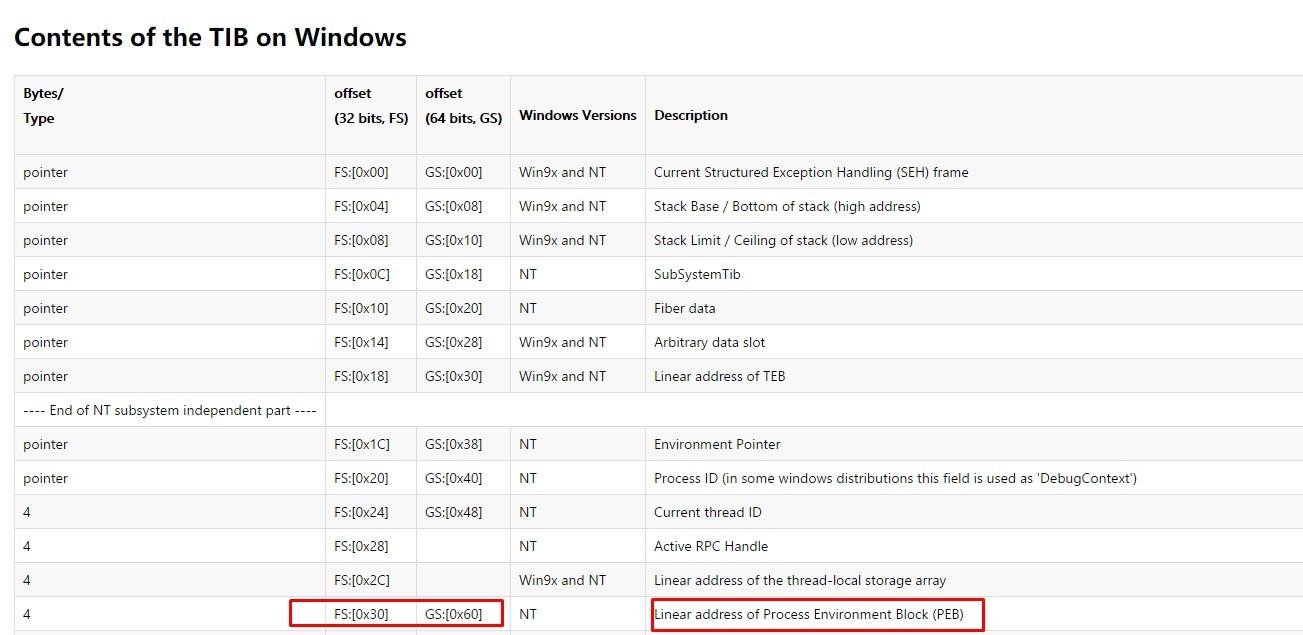
CDQ if the SF flag is zero sets RDX to zero, it could have been an XOR RDX, RDX



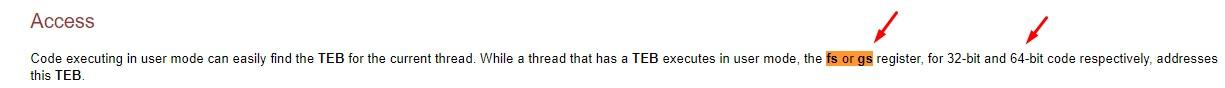


It's zero so it says RDX=0.

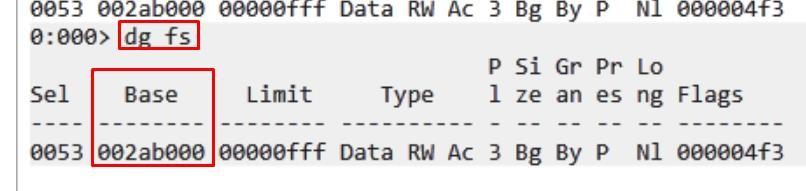
Remember that in 32 bits, the TEB or TIB was pointed by FS.



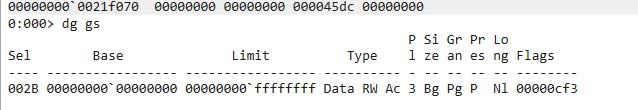
In 64 bits, the GS register is used for the TEB.



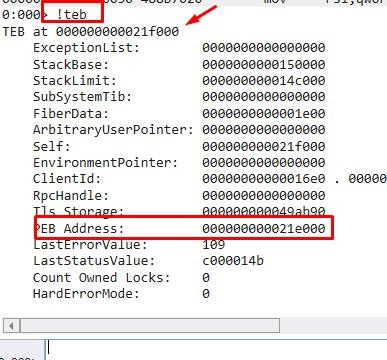
In 32 bits we could use the command **dg fs** to see the value of FS.



It doesn't work with GS.



And we have more tricks, hehehe, there is the **!teb** command



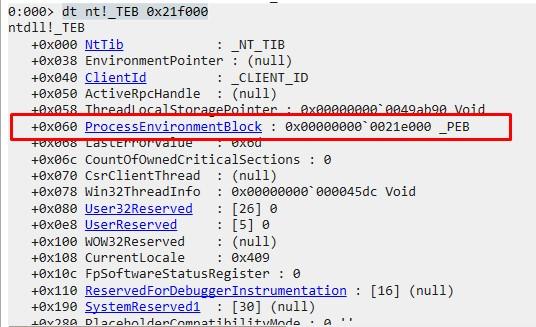
There we see the content of the TEB, its base in my computer is 0x21f000 and the address of the PEB is 0x21e000.

If I trace the first instruction it is reading the PEB from the 0x60 field.

**mov rax, qword ptr gs:[rdx+60h]**

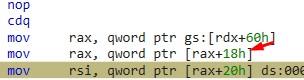
Since I know the address of the TEB in my machine which is 0x21f000 I can use the **dt** command and see it better.

**dt nt!\_TEB 0x21f000**

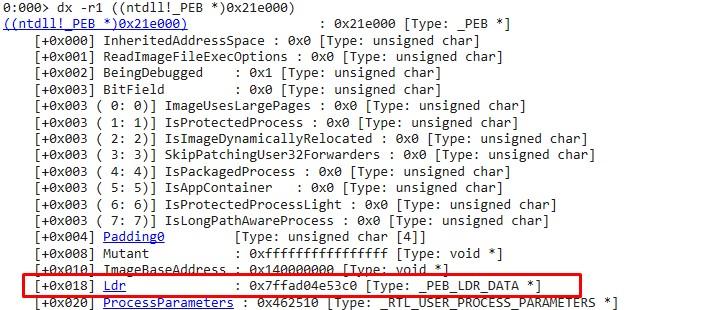


I also have a link to show the PEB that will work for me.

The second instruction is:



Read the field that is in the 0x18 offset of the PEB, as I have the link I can click and see the list of the PEB.



It reads PEB->Ldr



Well, let's get on with it.

We can click on LDR or list \_PEB\_LDR\_DATA



We see that in the 0x20 offset it loads InMemoryOrderModuleList

Microsoft says



In some web pages (and when we did the 32-bit RESOLVER part)

LDR\_DATA\_TABLE\_ENTRY is also called LDR\_MODULE which is shorter but the same.

It is still convenient to call it LDR\_DATA\_TABLE\_ENTRY since that way it is listed in the Windbg.



As we can see, the first field is of \_LIST\_ENTRY type and, as the documentation says, it has its FLINK that points to a similar structure that corresponds to the following module, being a linked list.





So as we see in the image, the structures are connected between them, by means of the FLINK and BLINK, as FLINK is a pointer to the following structure, only finding the content of it, we will have the FLINK of the following structure.

**mov rsi, [rax + 0x20]**

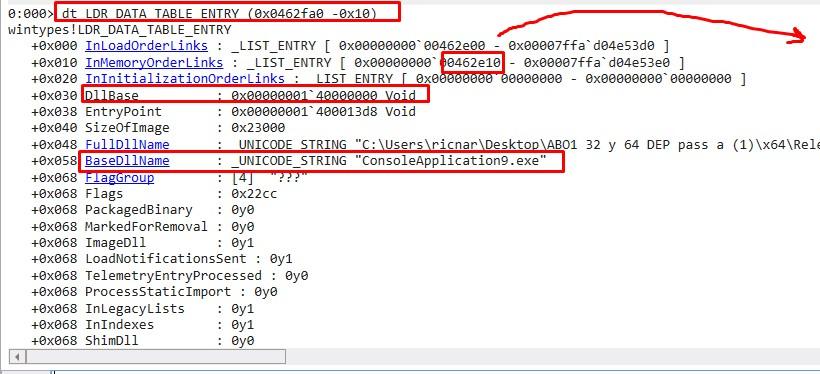
So in this instruction you load in RSI=InMemoryOrderModuleList that as we saw is the beginning of the linked list and at the same time it belongs to the first module of the LDR\_DATA\_TABLE\_ENTRY string.

Those who followed the tutorial of the 32 bits resolver will remember that there was used the first field InLoadOrderLinks, both are two lists with the same information about the modules, only it will change the order in which they are located and that in this case instead of being in the OFFSET 0x0 of the structure as it was InLoadOrderLinks, we are with our FLINK always in the offset 0x10.



RSI is in the 0x10 offset of the first LDR\_DATA\_TABLE\_ENTRY , in my machine it's 0x0462fa0.

I can list in the Windbg.



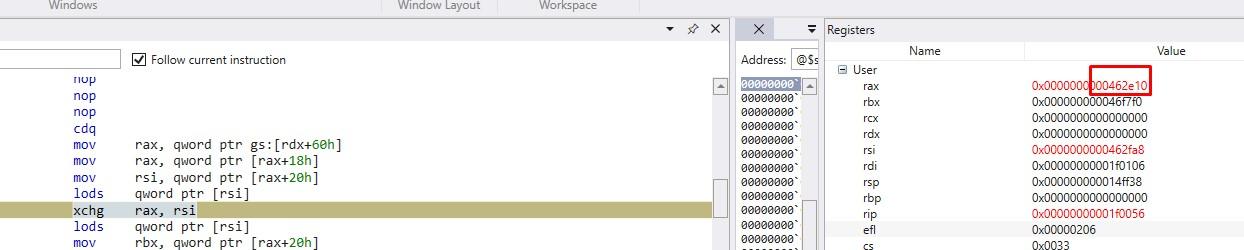
There we see that we were at offset 0x10 so we had to subtract 0x10 to list the structure

dt LDR\_DATA\_TABLE\_ENTRY (0x0462fa0 -0x10)

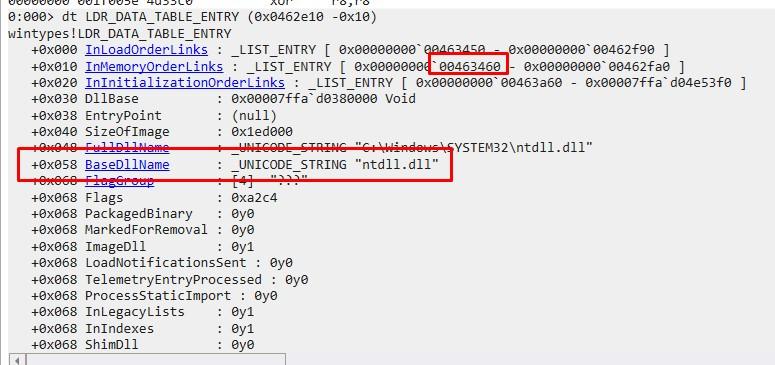
We see that it corresponds to the executable module, which is always the first one in the chain, there we see its ImageBase and its name, we also see the FLINK to the structure of the second module.

This is done programmatically by finding the ESI content since the LODS instruction reads the ESI content and moves it to EAX.

lods qword ptr [rsi] ds:0x00462fa0=**0x0462e10**

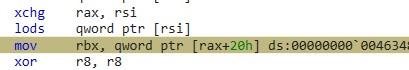


EAX is again in the offset 0x10 of the second structure we can look to see which module it corresponds to.

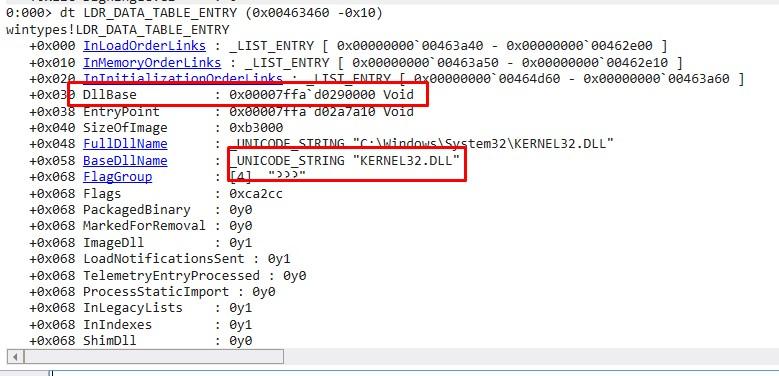


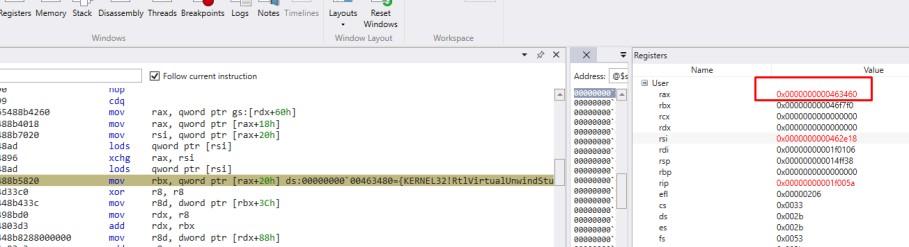
We see that the second LDR\_DATA\_TABLE\_ENTRY corresponds to ntdll.dll and that the third one is pointed by the FLINK will be 0x463460.

Then EAX moves it to ESI using XCHG.

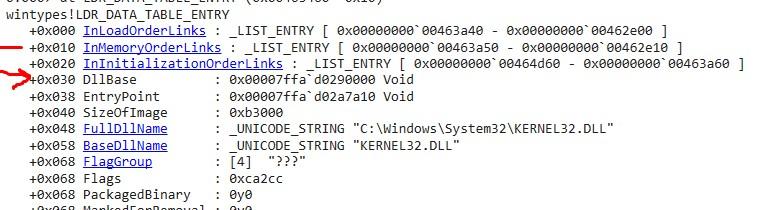


And then finds the content again using LODS and of course it matches will be 0x463460.

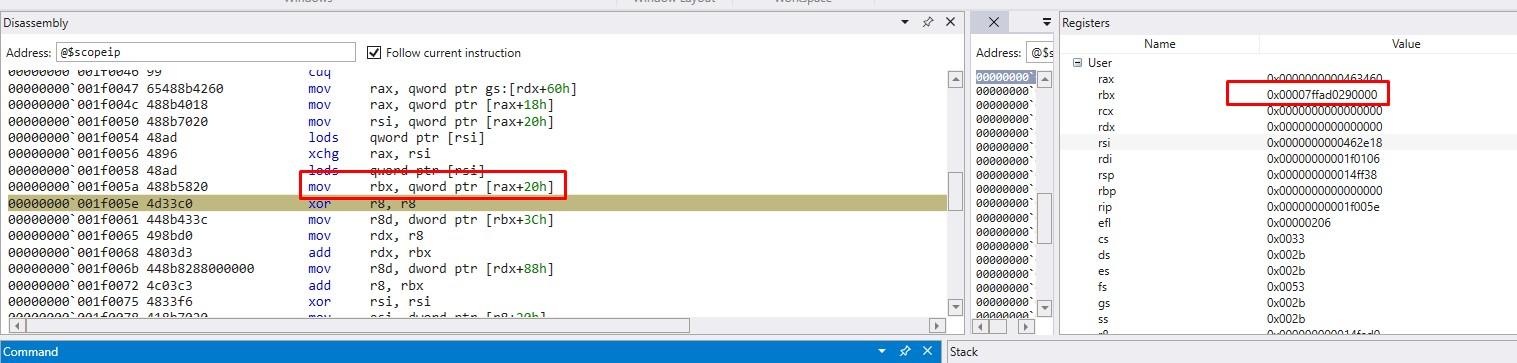




It corresponds to kernel32.dll and since EAX is positioned at offset 0x10, to read the base of kernel32.dll you have to add 0x20 to get to 0x30 where it is.

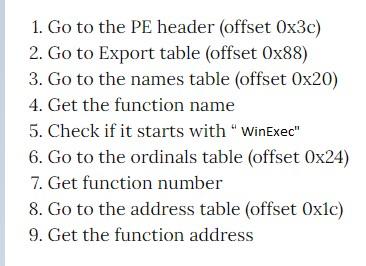


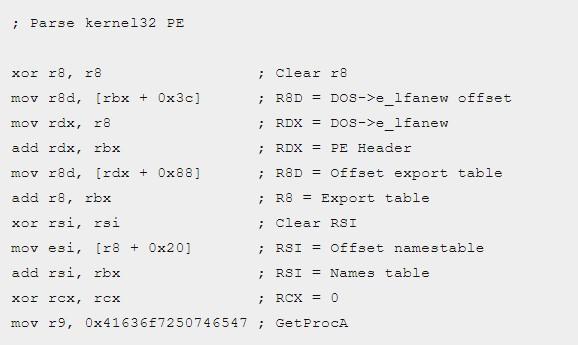
With this it already found the kernel32.dll base that was the first target to look for.



# FINDING WINEXEC'S ADDRESS

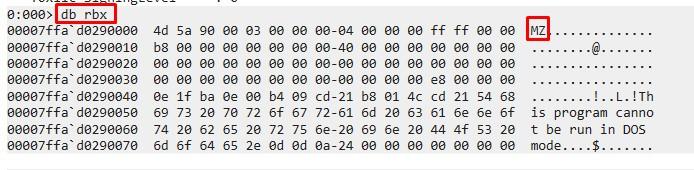
Once the base of Kernel32.dll is found, the steps to find WinExec or the function we want inside kernel32 are the following.



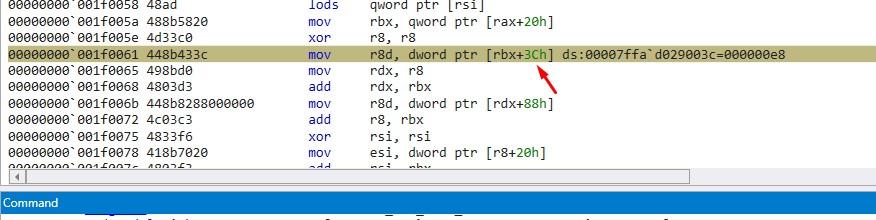


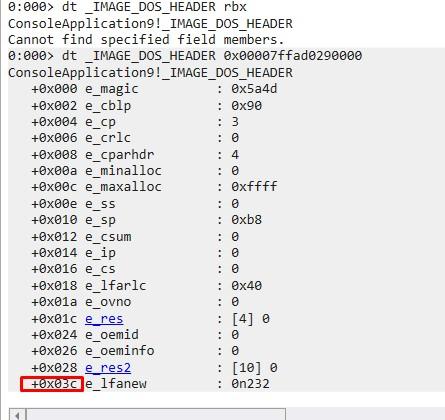
That's part of the code we will trace to check that everything corresponds.

The structure where the header starts, which of course is in the address of the kernel32.dll image we found, is called \_IMAGE\_DOS\_HEADER there we see the characteristic MZ the two bytes that are at the beginning in the DOS executables.



We see that in the shellcode we read the field of the 0x3c offset

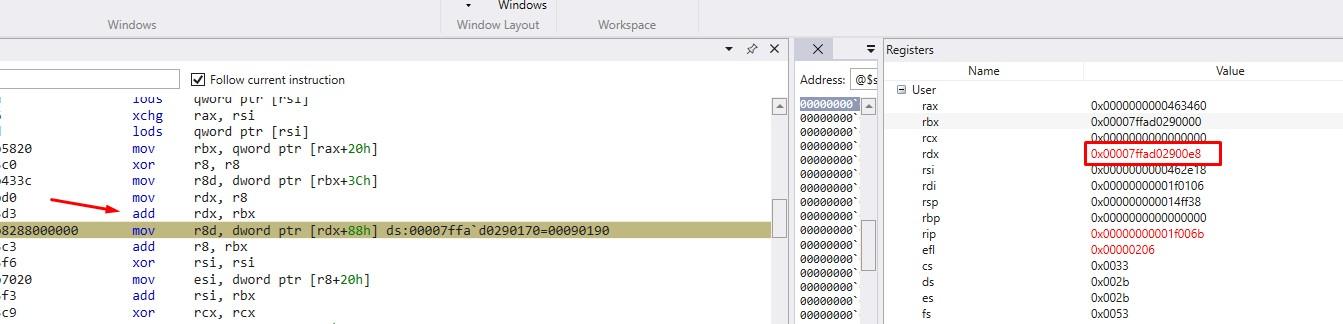




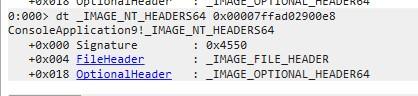
Which is 232 decimal, so 0xe8 (in hexadecimal) is the offset of **\_IMAGE\_NT\_HEADERS64**



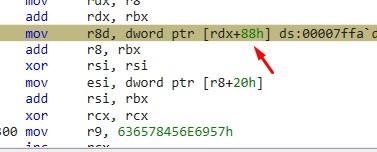
I adds the image base to get the address.

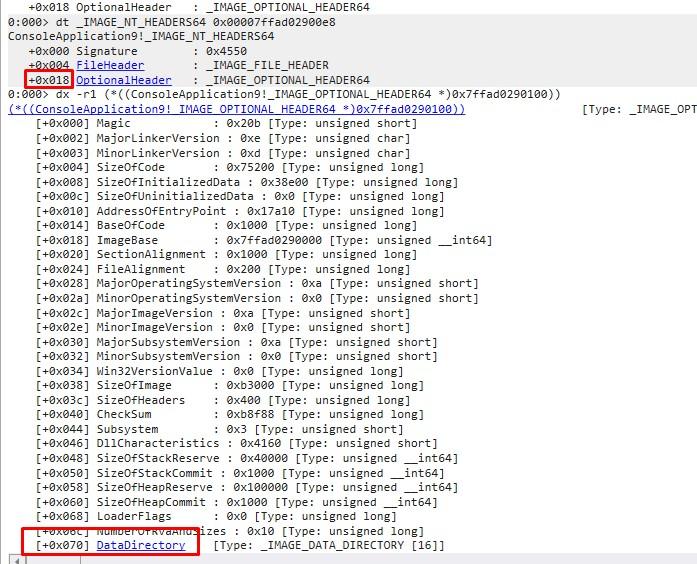


In RDX we have the address of \_IMAGE\_NT\_HEADERS64



Then look for the field 0x88 we see that it is inside OptionalHeader which is at 0x18.

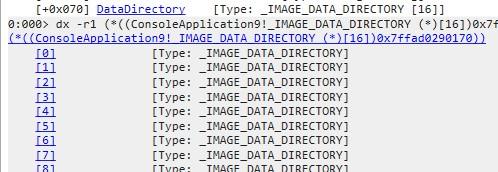


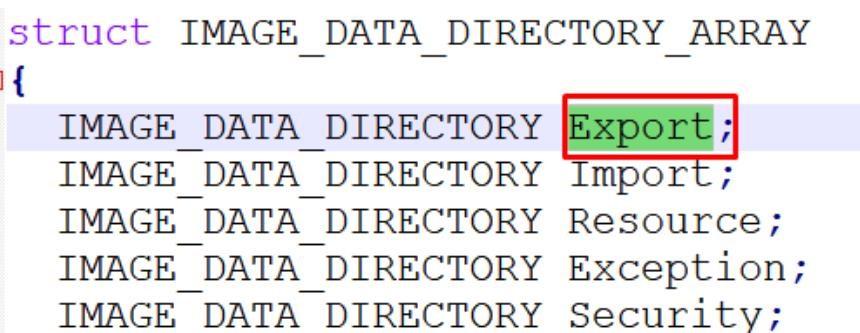


In 0x70 is \_IMAGE\_DATA\_DIRECTORY64 plus the 0x18 of

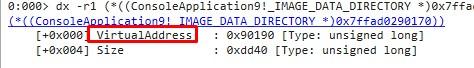
\_IMAGE\_OPTIONAL\_HEADER64, we are in 0x88 as it reads the shellcode.

That's an ARRAY of \_IMAGE\_DATA\_DIRECTORY

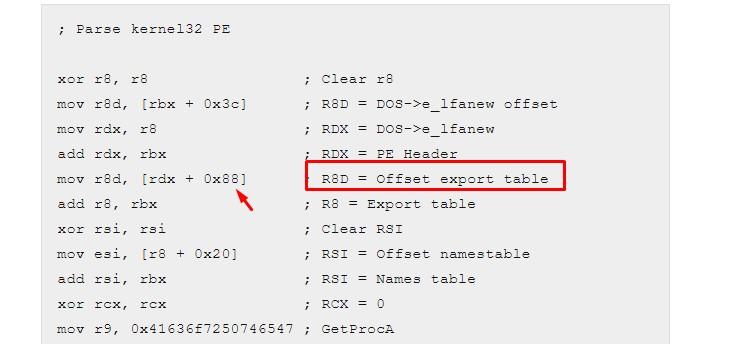




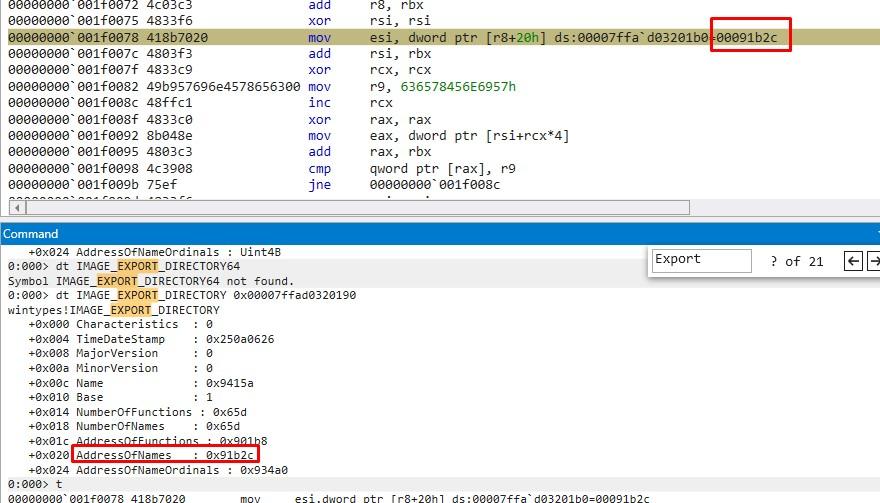
The first is the offset to the EXPORT TABLE its first field is the offset to the direction.



Then it adds the base to obtain the direction of the EXPORT TABLE.



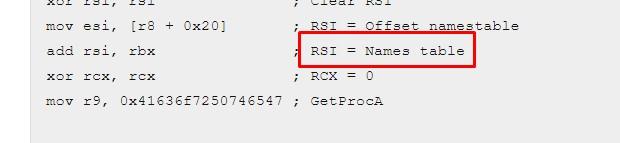




Read the AddressofNames for offset 0x20.

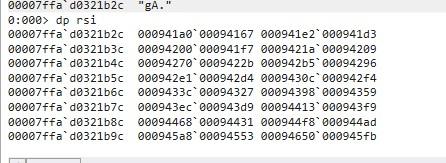


So, there are three AddressOfFunctions arrays at 0x1c, AddressOfNames at 0x20 and AddressOfNameOrdinals at 0x24.





So, RSI has the table or array of names.

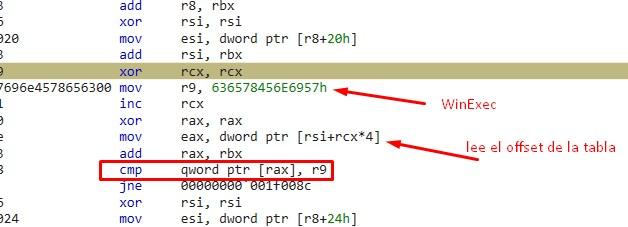


To each one of the table entries, to find the string with the name you have to add the base and that's what it does, it goes around the table reading the offset and adds the base and compares with WinExec (if you want another function you have to change here for the name of the one you want to find)

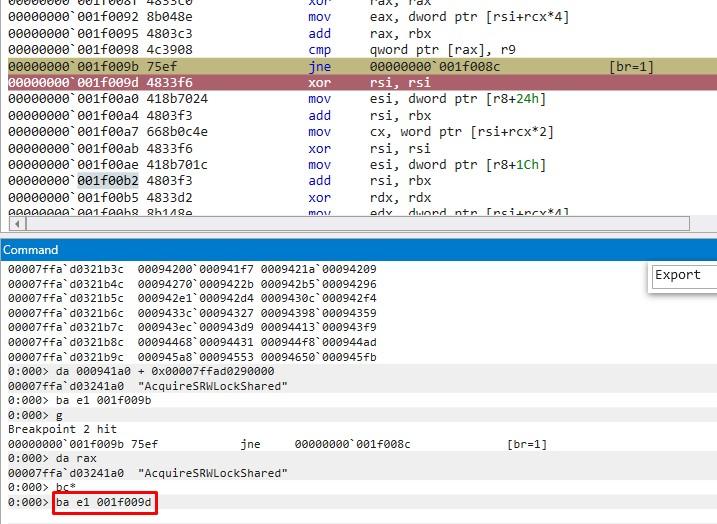
Let's see what the first one on the board is aiming at.



And so each of these offset plus the base points to the name of an exported function, so it makes a loop through the table, comparing each string with WinExec.



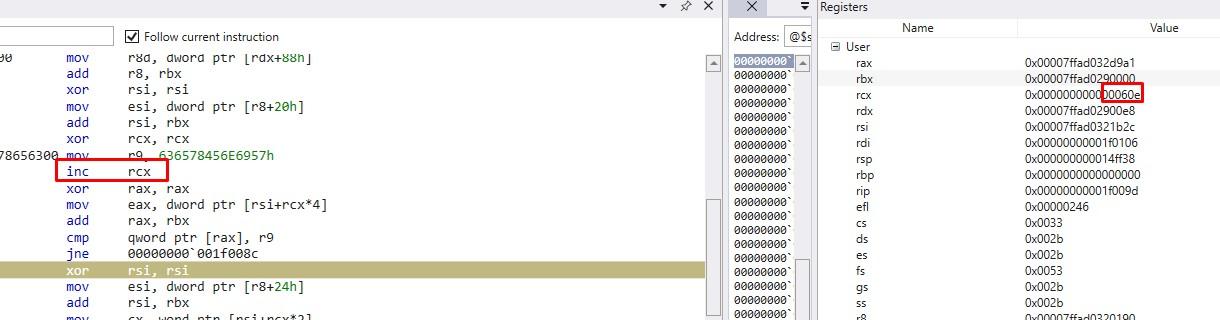
We can put a BREAKPOINT after JNE and press RUN to stop when it finds the name.



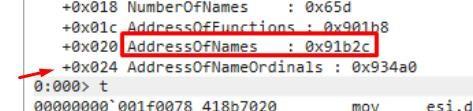
When it stops.

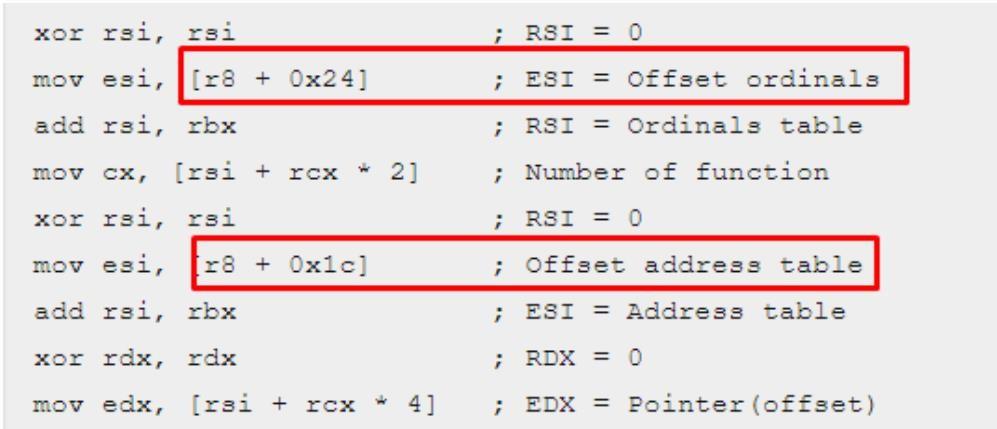


We see that it was increasing RCX which was the table index, so the position in the table for WinExec is RCX=0x60e.

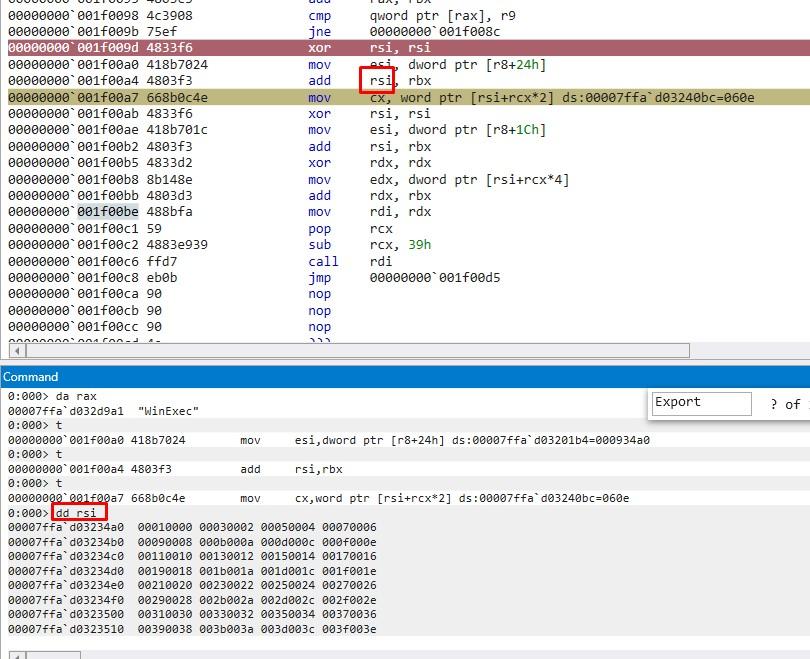


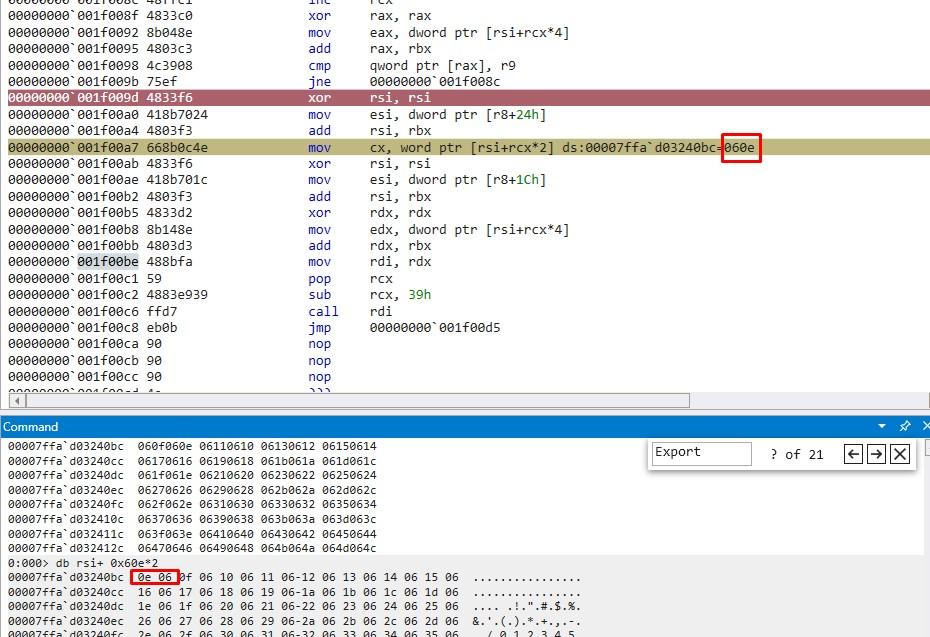
Remember that we add 0x20 to r8 and then the base to find the name table, if we add 0x24 and then the base we find the ordinal table.



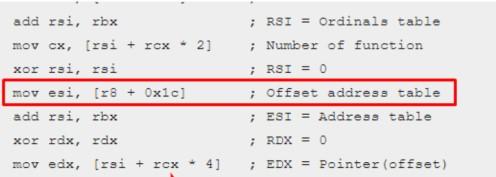


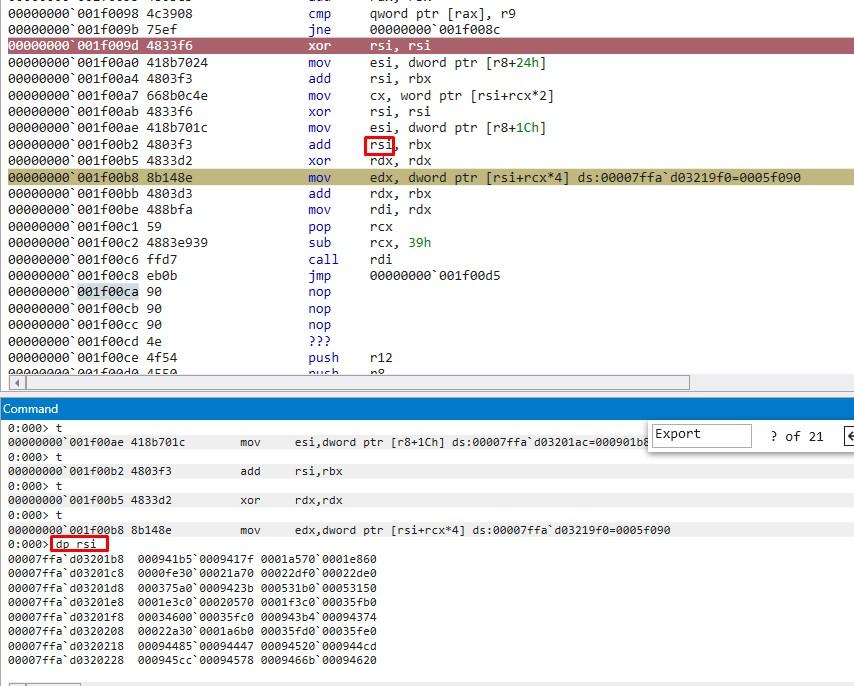
With this in the ordinal table without looping using the index value that is in RCX of the position of the table of names, we read the number of the function of that table.



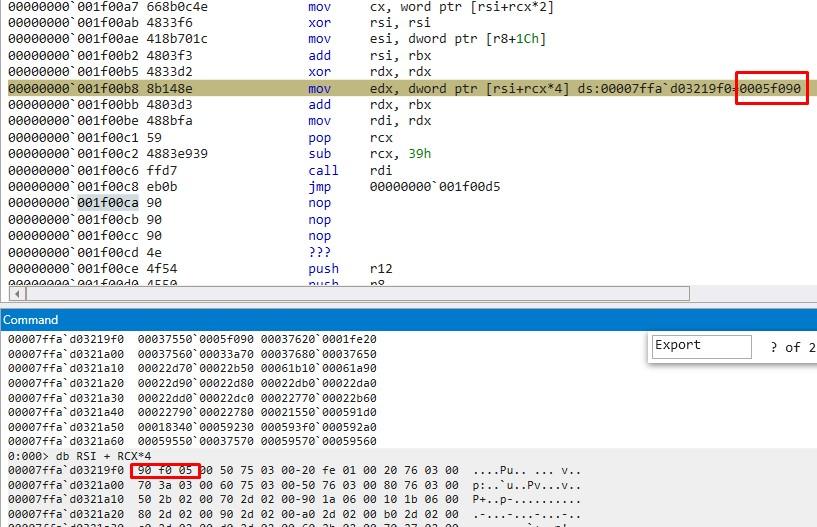


It is also 0x60e, that is used in the last table to find the offset of the WinExec function.



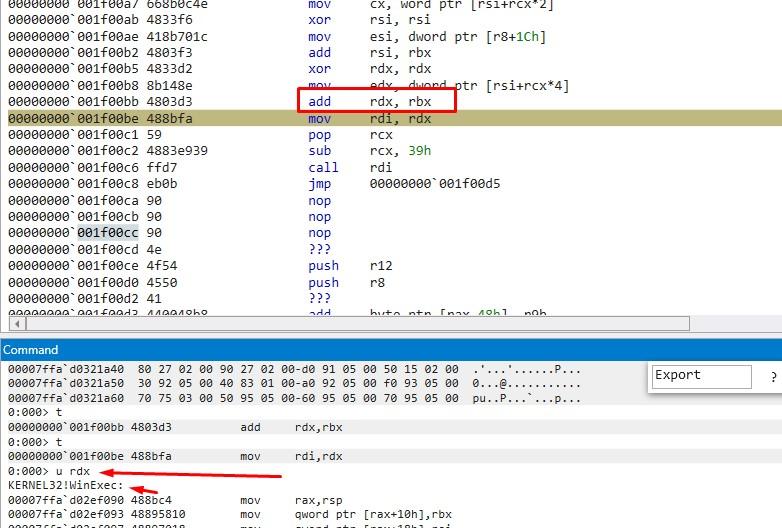


So, RSI + RCX\*4 gives us the offset of WinExec

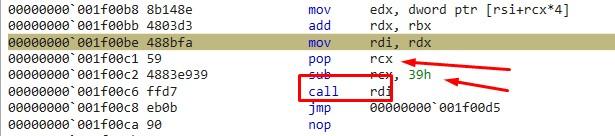


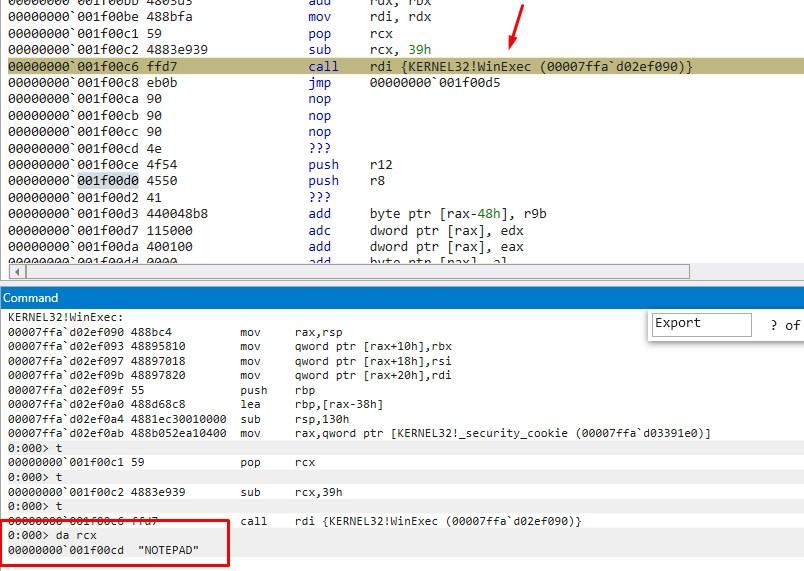
To which we add the base and we have the virtual address of WinExec.

# CALLING WINEXEC

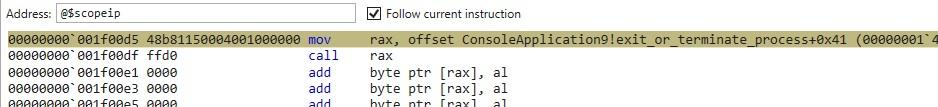


And that's it, then you arrange a NOTEPAD string to pass and jump to run WinExec with the "NOTEPAD" argument.

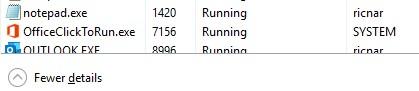




And then if we pass with f10 the call and continue it calls ExitProcess to close it.



There is the NOTEPAD running.



Well with this we finish the 64 bits RESOLVER.

See you in the 14th part.

Ricardo Narvaja

03/23/2020