EXPLOITING AND REVERSING

USING FREE TOOLS

(PART 10)

After having seen some basic examples of exploitation and reversing, in this second stage we are going to take a step forward, gradually adding different protections and mitigations that we will find.

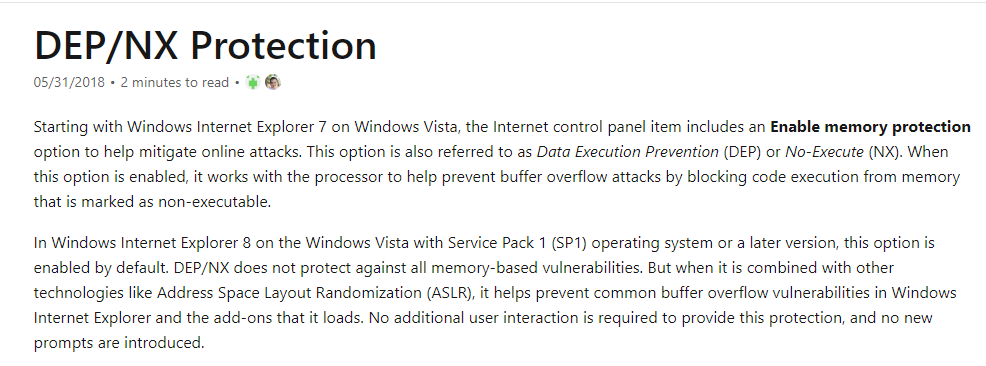
What are mitigations?

Over the years, new generic defense methods have been added to the new systems, which of course, as the name implies, do not prevent but mitigate or make exploitation more difficult.

<https://docs.microsoft.com/en-us/windows/security/threat-protection/microsoft-defender-atp/customize-exploit-protection>

The first thing we will add will be a simple 32-bit exercise, similar to the ones we were seeing but that has the protection we call DEP enabled, then we will do the same with the first 64-bit exercise, where DEP is enabled for all processes, so, we will do it gradually to learn how to manage in these cases.

What is DEP?



That is the definition in English that we can find on the MICROSOFT page.

The point is that if a process has DEP enabled, the memory areas used to input or manage data like heap or stack do not have permission to be executed, i.e. they can only have at most read and write permission.

On the other hand, only the code sections of the executables and DLLs have permission to execute, which in turn does not have write permission.

This way, it becomes difficult to be able to execute our own code, because if we store our own code in a buffer that I want to execute when I jump to execute it, it gives me an error for not having execution permission, I will not be able to exploit it as we did in the stage 1 of this course.

In other words, when a program starts if it has DEP enabled:

**SECTIONS THAT HANDLE DATA = R or RW (non-executable, read and / or write-only)**

**CODE SECTIONS = X (executables)**

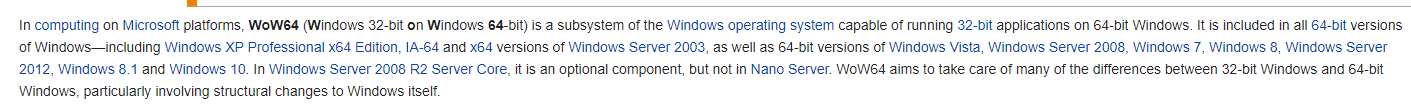
While without DEP enabled

**SECTIONS THAT HANDLE DATA = RX o RWX (executables + read and / or write)**

**CODE SECTIONS = X (executables)**

Another point to note is that on 64-bit Windows systems, the 32-bit processes running on WoW64 behave like a 32 bits.

What is Wow64?



Simply speaking to give you a better idea, it's like we had an emulator for a 32-bit Windows system running inside a 64-bit Windows system.

In other words, if the WoW64 subsystem did not exist within 64-bit systems, we would not be able to run 32-bit applications on them.

All this is transparent to the user, he can execute a 32-bit or 64-bit process without having to carry out any additional steps, just by double-clicking on an executable, the system will detect if it is 64-bit and will execute it directly and if it is 32 bits it will pass it to the WoW64 subsystem for it to run.

So all the executables we saw in stage 1 were compiled in 32 bit, they are actually running on the WoW64 subsystem.

BY DEFAULT:

**PROGRAMS compiled in 32 BITS or WOW64: depending on how a program is compiled with or without DEP, it will run with or without DEP respectively.**

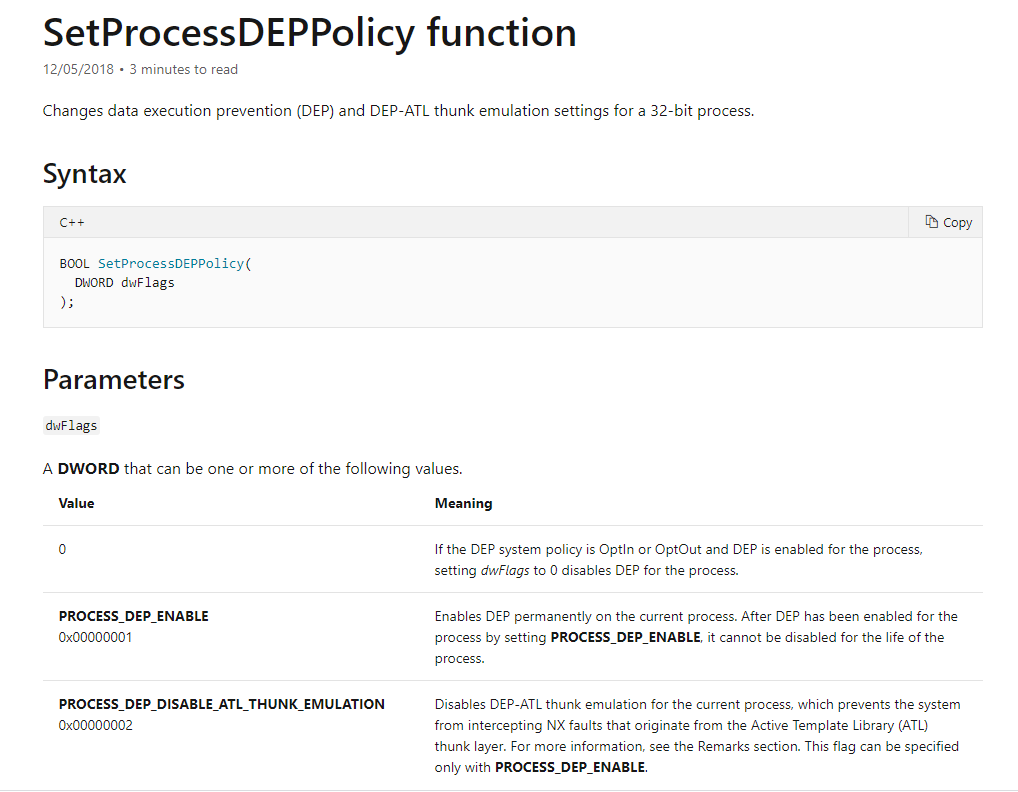
**PROGRAMS compiled in 64 BITS: It does not matter how this compilation will always have DEP ENABLED.**

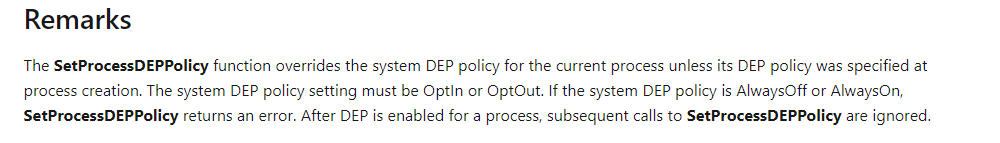
Of course, we are talking about the default system configuration which can be changed, for example in 32-bit systems there is the possibility of changing to all processes having DEP activated as in the default configuration of 64-bit systems, but here in more, we will always refer to each system with its options as they come by default.

In addition to all this there is a windows function that a program can use to activate DEP at runtime in 32-bit processes.

What is SetDEPProcessPolicy?

<https://docs.microsoft.com/en-us/windows/win32/api/winbase/nf-winbase-setprocessdeppolicy>





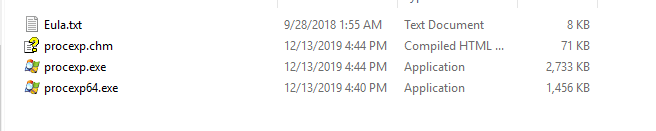
The issue is that it is only normally used to enable DEP, since it only serves to disable DEP if it was enabled by the same function, DEP enabled with another method cannot be disabled using this.

So we have many different possibilities for a process to have DEP or not enabled, so just seeing statically if an executable was compiled with DEP or not, it is not enough to know if it will finally have it enabled when it runs.

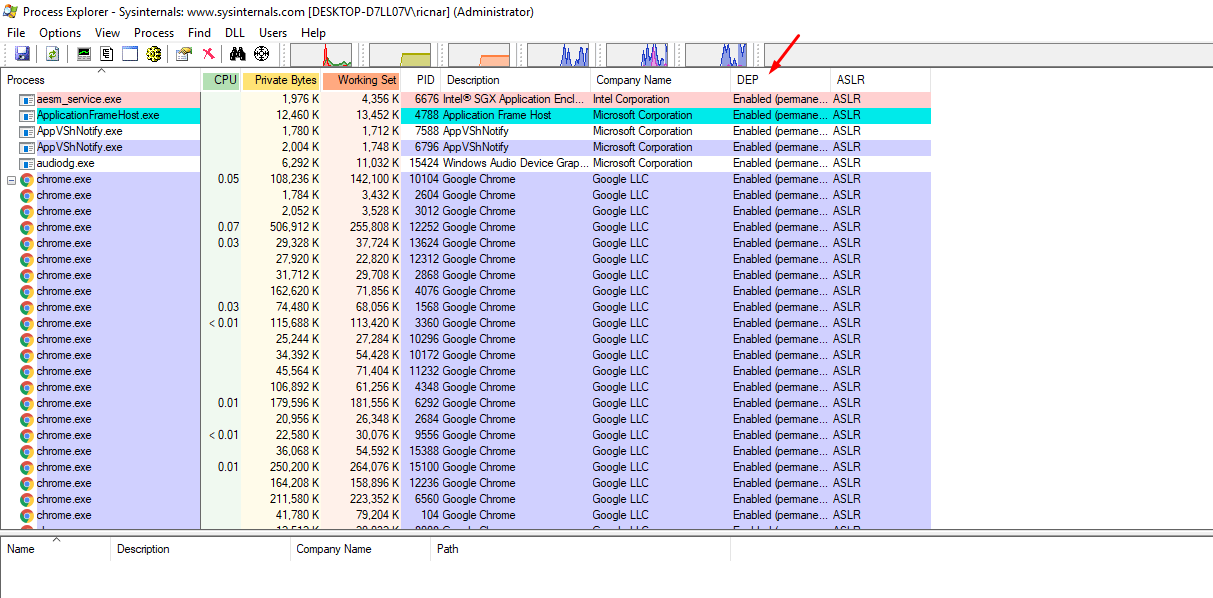
So the best way is to use the tool called PROCESS EXPLORER which is from MICROSOFT.

<https://docs.microsoft.com/en-us/sysinternals/downloads/process-explorer>

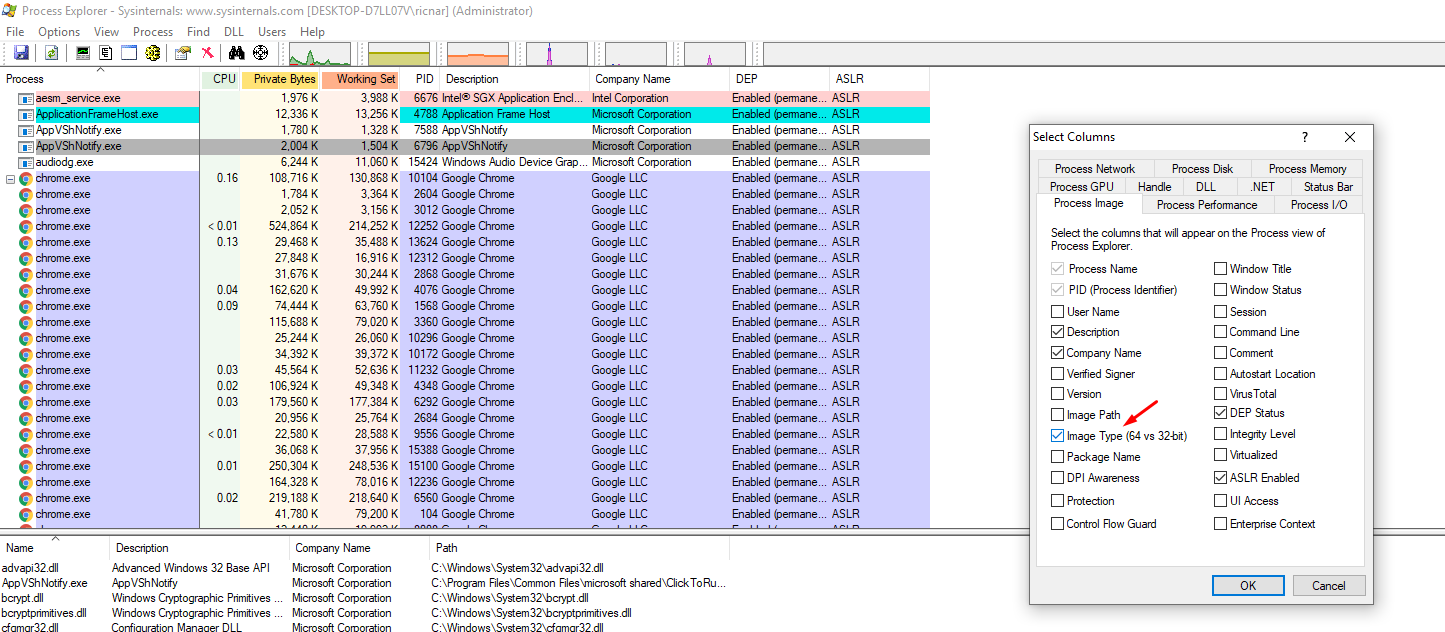
Once we run it.



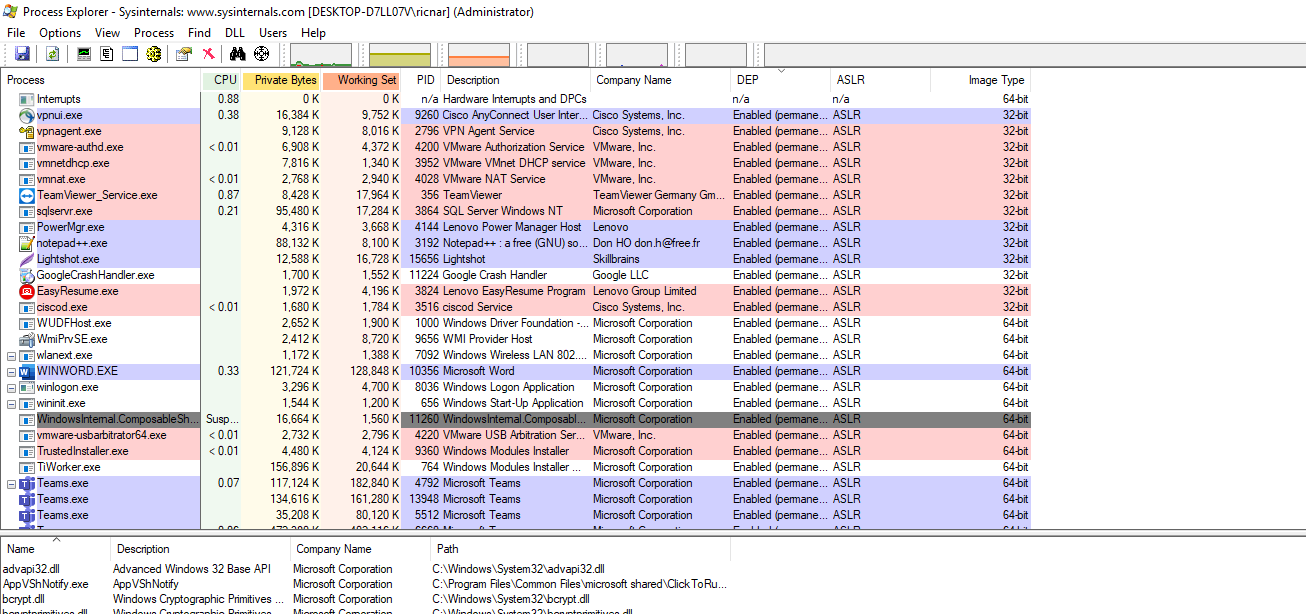
Let's run the 64-bit version with administrator permission, it will show all the processes on a machine that has a 64-bit Windows system, be 64-bit or Wow64.



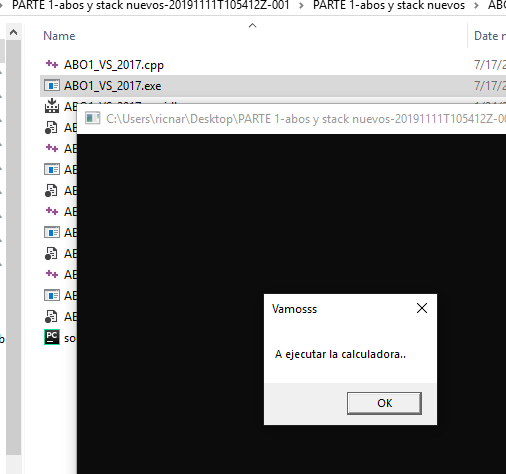
There we see the DEP column and we see that as we said the 64-bit processes have DEP enabled.



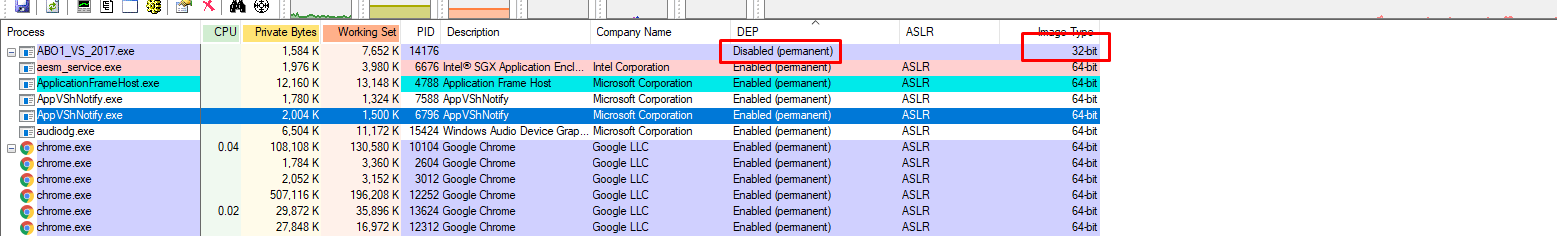
We can add that column with RIGHT-CLICK -SELECT COLUMNS in the column bar.



We see that the 32-bit processes, that is, they run on WoW64, have DEP enabled in this case, but if I run any of the exercises that we saw in the previous installments.

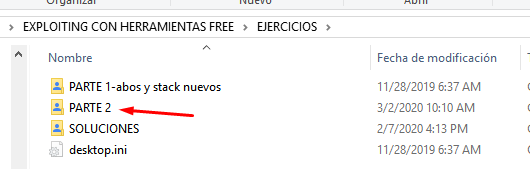


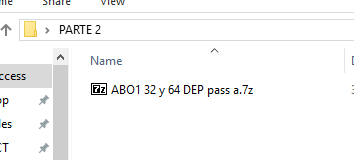
Let's look at this process in the PROCESS EXPLORER.



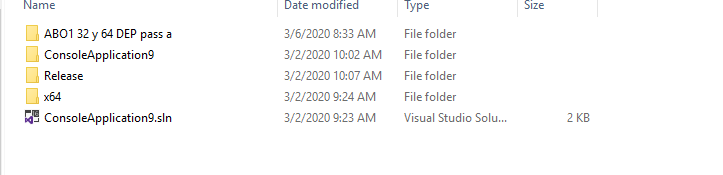
The same as it was compiled in 32 bits and without DEP it runs without DEP enabled.

Now if I take the first exercise that we are going to see after stage 2.



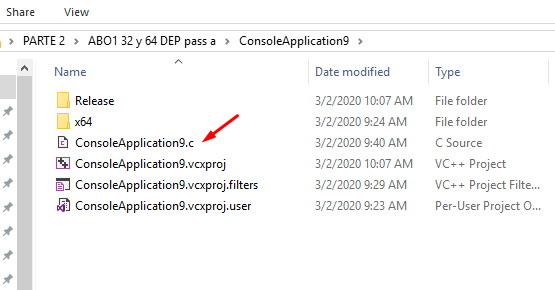


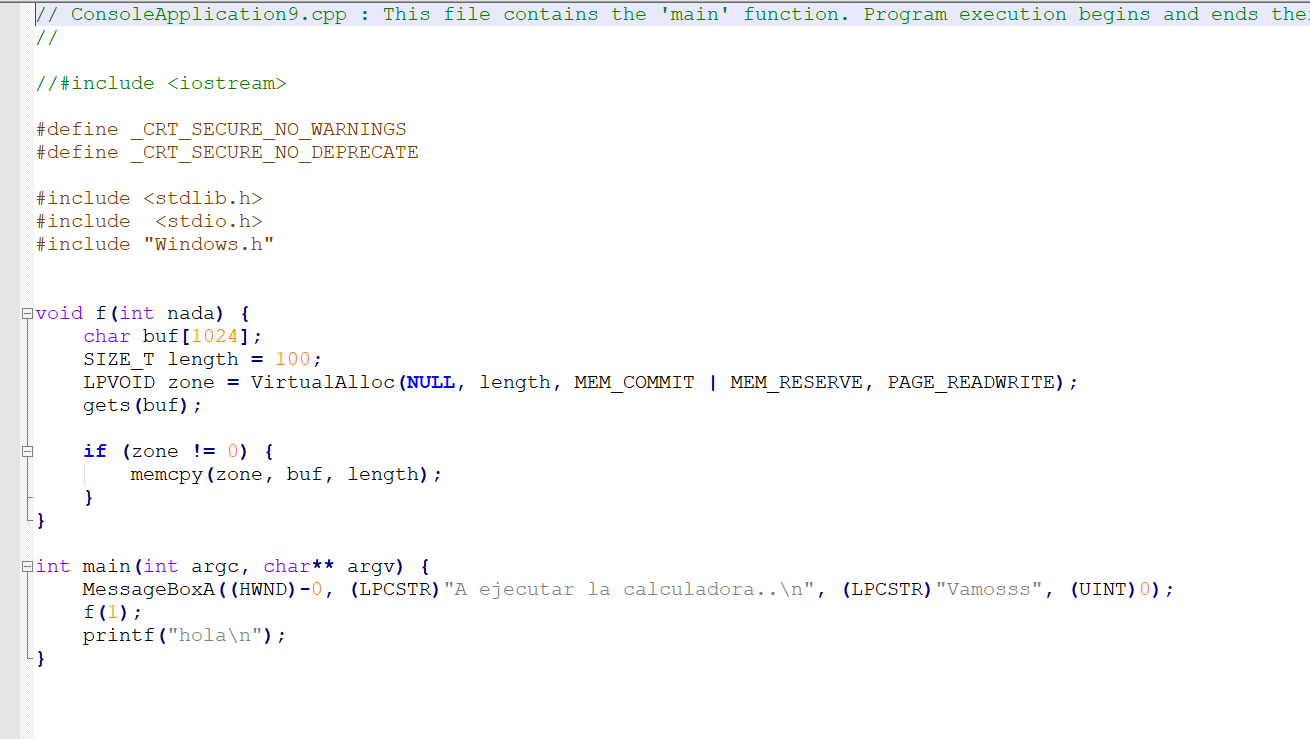
After decompressing it (you have to use the password **a)**, there are the following two exercises.



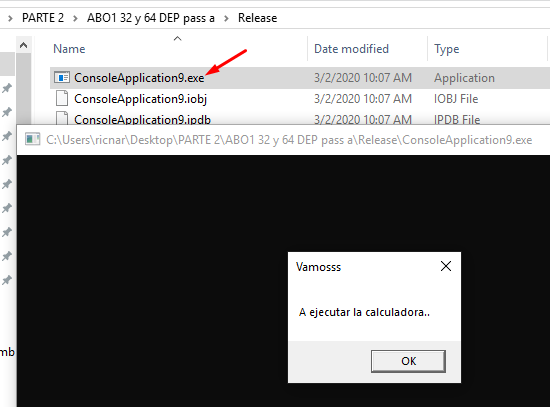
We see that it is the same exercise compiled for 32 and 64 bits, inside the Release folder is the compiled 32-bit version and in the x64 folder the compiled 64-bit version.

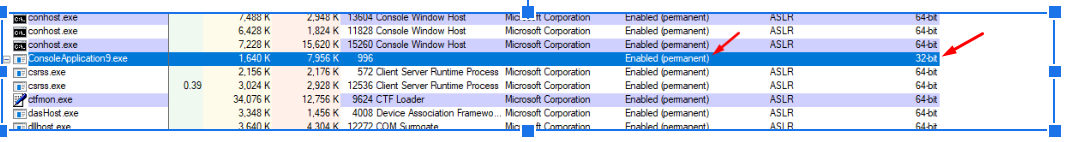
If you want to see the source code, it is the same for both.



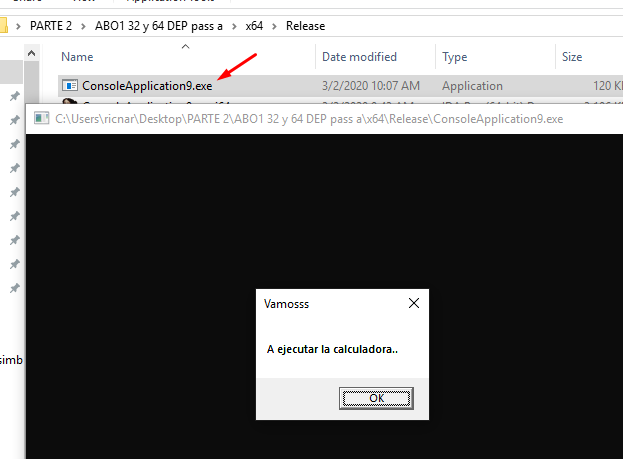


Let's run the 32-bit version and look at the PROCESS EXPLORER.





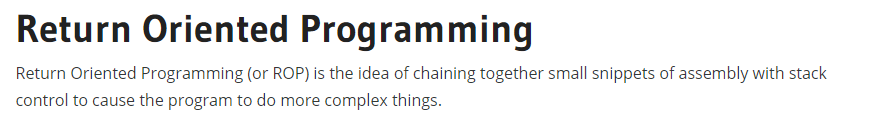
We see that this one has DEP enabled permanently, and if I run the 64-bit it also has it enabled.





So our next objective will be the DEP mitigation, and how to bypass it, for this we must study the technique called ROP.

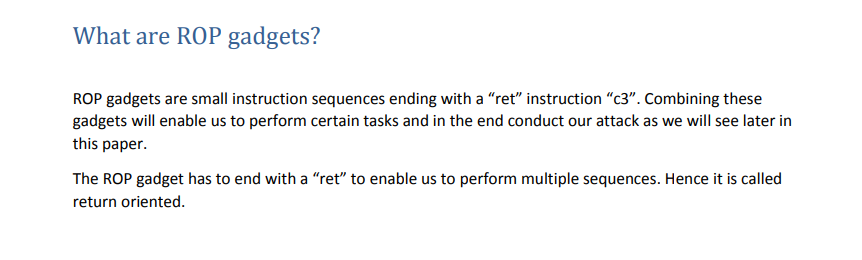
What is ROP?



As it says there is a technique of finding small pieces of code in the executables or DLLs of the process, using the code sections of the same process, that is, those allowed by DEP since they are marked as X or executables.

These pieces of code are called GADGETS.

What are GADGETS?



Not only can they end in a RET (C3), gadgets can also end in C2 (RETN CONST) and even in some sophisticated cases they can end in a CALL or JMP.

We will make them build a call to a Windows function such as VirtualProtect or VirtualAlloc, which allows us to give execution permission to a data section that did not have it initially or create a new section that has read, write and execute permission, copy there our code and then jump to execute it.

In other words, if we have a process with DEP enabled:

**SECTIONS THAT HANDLE DATA = R or RW (non-executables, read and / or write-only)**

**CODE SECTIONS = X (executables)**

We will be able to add X (execution) to some part of the memory.

**SOME SECTION THAT HANDLES DATA = RX or RWX (we added execution already had read and / or write)**

**CODE SECTIONS = X (executables)**

In other words, the way to bypass DEP is not to disable it for the whole process, which is not possible, but add execution permission to some part of the memory that handles data, to be able to execute our code there.

The idea is that since we control and overflow the stack, we can jump to any direction we want, and if instead of jumping as before to a CALL ESP, CALL EAX or a code that directly jumped to execute our code, let's jump for example to a gadget that makes POP ECX-RET.

This will execute POP ECX that will copy the value that is stored in the stack that we control to ECX, and then the RET will make us jump to the next gadget, that we also control because it will be below the value that was copied to ECX.

This way we are chaining the execution of gadgets. After the first one the chain continues with the second gadget and thus, each one is moving the values ​​that we need to each register to build a call to the VirtualAlloc or VirtualProtect function.

Obviously chaining existing gadgets is something that will not always be similar, there are times that we will find the gadgets that we need in the modules that do not have randomization and have fixed addresses where to jump. Otherwise it needs to be complemented with address leak techniques, that allow us to obtain module addresses to avoid ASLR and sometimes it will not be so easy and you will have to dig deeper.

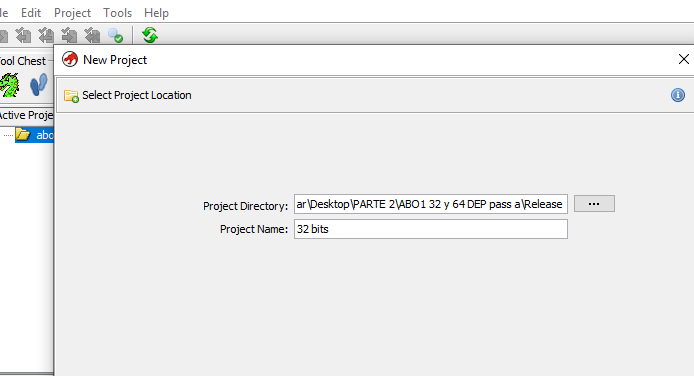
That is why ROP techniques must be practiced a lot, they are very changeable depending on the environment and the practice of the exploit writer depends a lot on whether a successful ROP can be performed or not.

Let's reverse the 32-bit executable with DEP in GHIDRA.

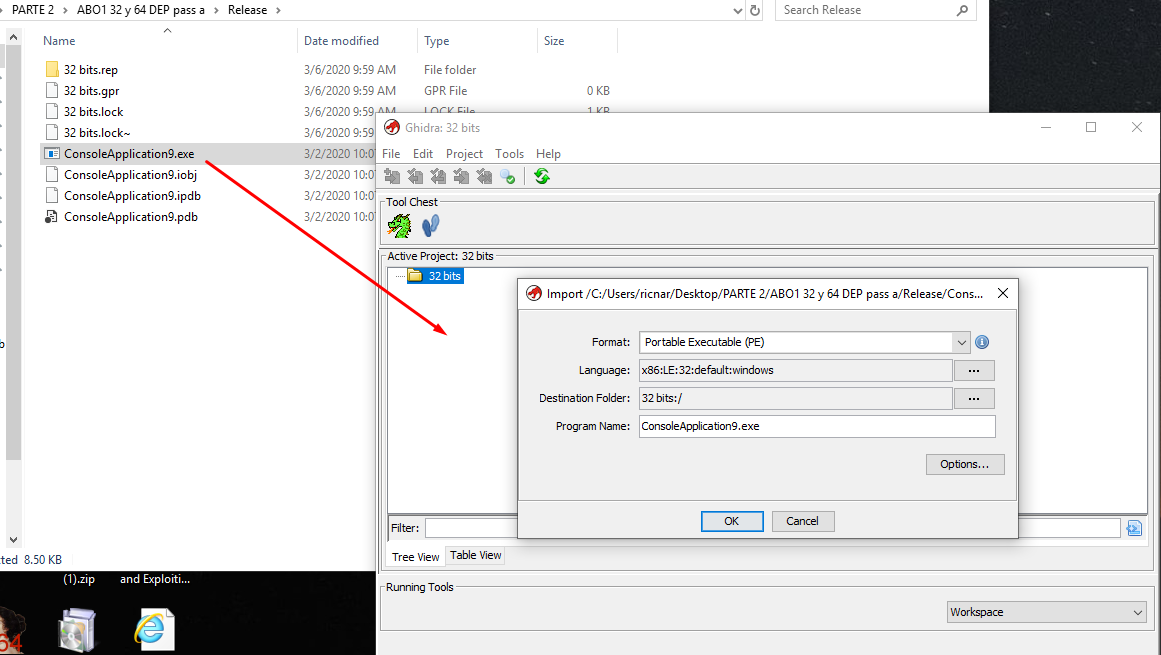
You can upgrade to the newest version right now:

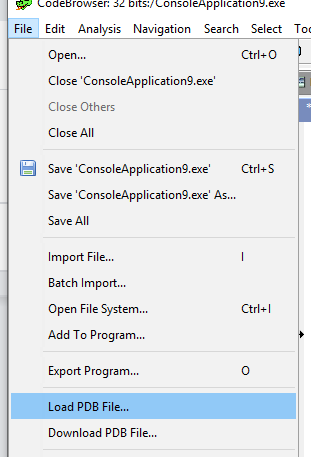


I delete the project I was working on previously and create a new one.



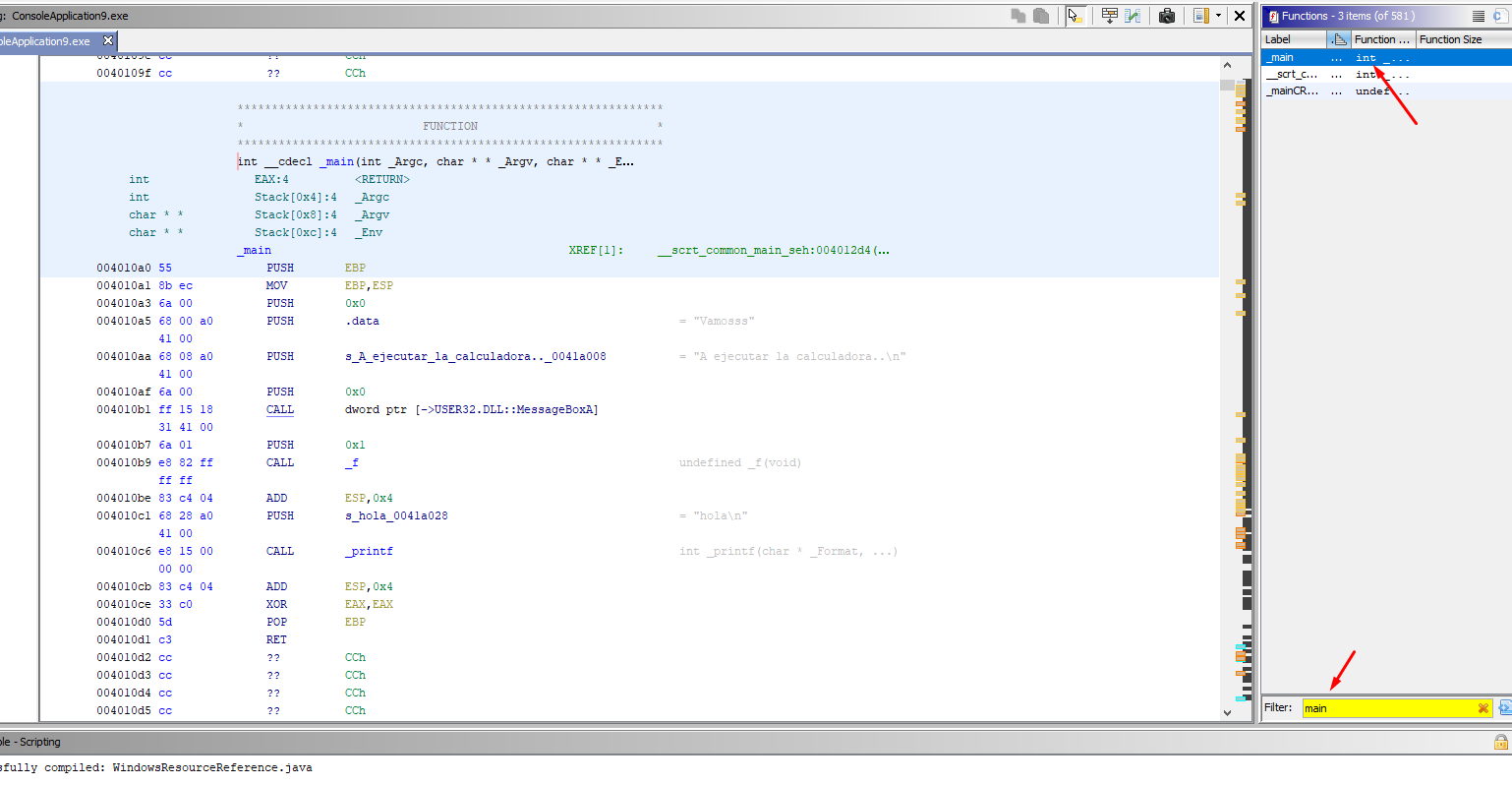
I drag and drop the executable into the created project.





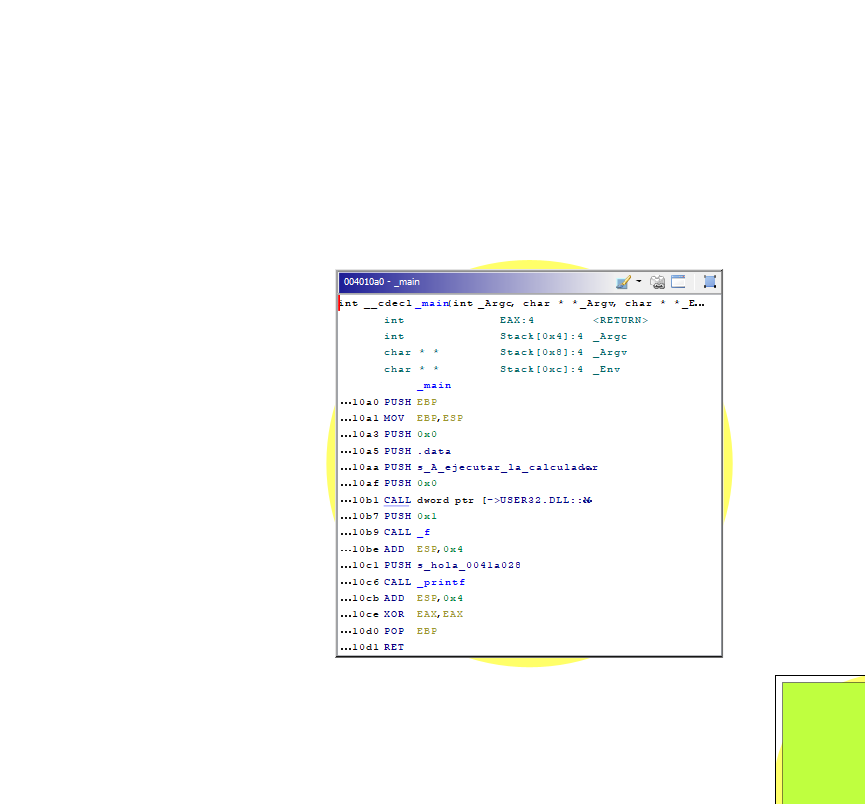
Once it parses, if I try to load the PDB with the symbols, it tells me I already load it, so you don't need to load them again.

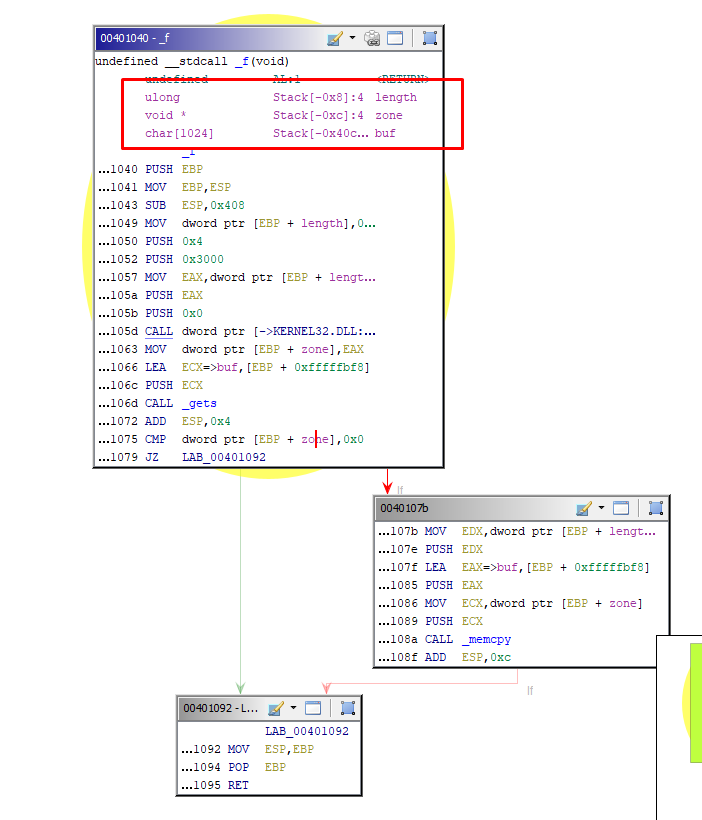
We see the main function.



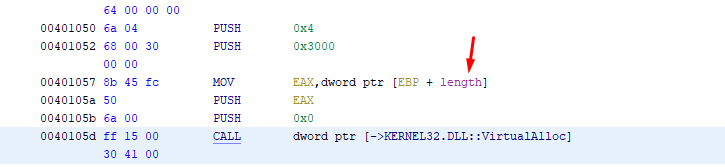
We look for it in the function search engine that is included, and then double click to go to the function.

We can see it in graphical mode with WINDOWS - FUNCTION GRAPH.

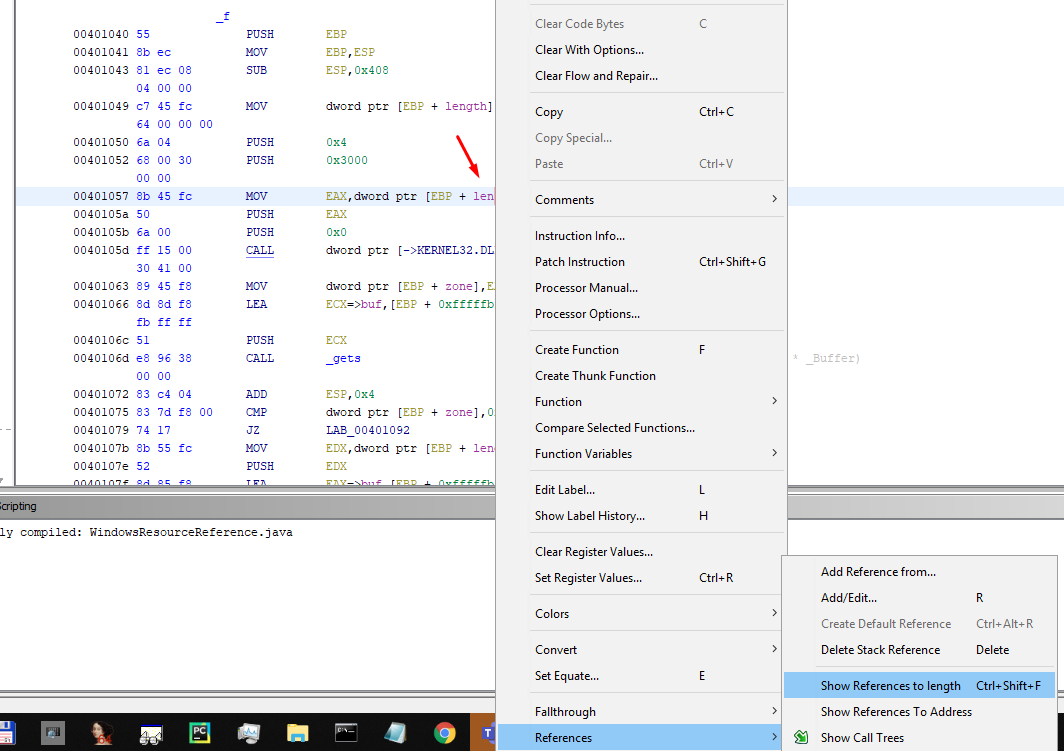


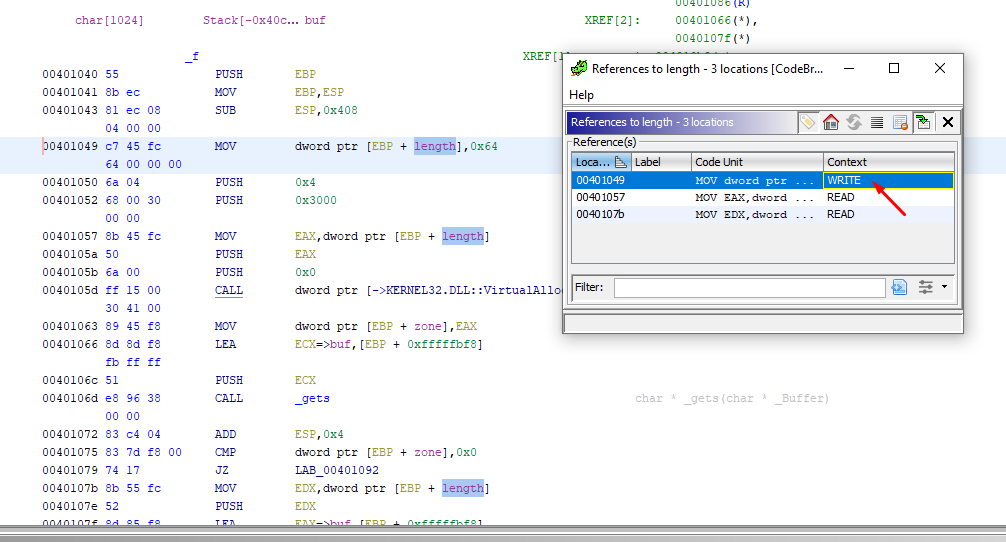
It is a single block and we saw that there is nothing here, only the call to MessageBoxA and then the call to the function f, which is where the overflow can be.

There we see that it is quite similar to the exercise we have already seen, although it has some difference, a call to VirtualAlloc to reserve in memory a number of bytes, given by the **length** argument passed there.



If we right-click on the **length** variable, we can see where it is used.

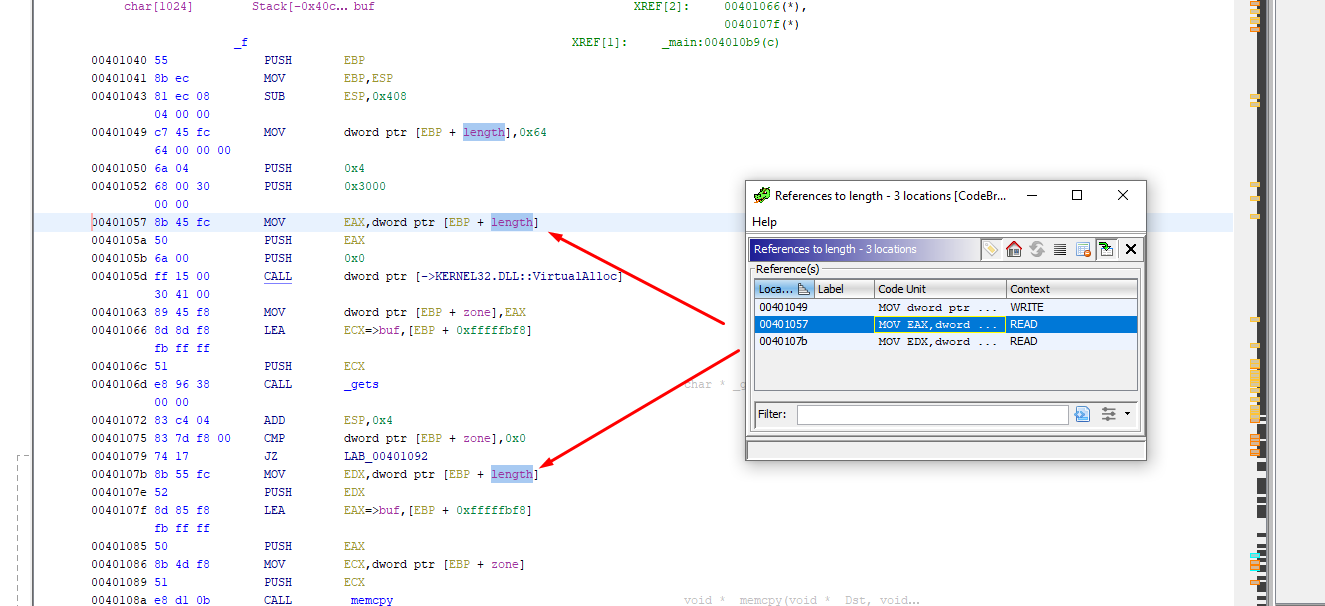




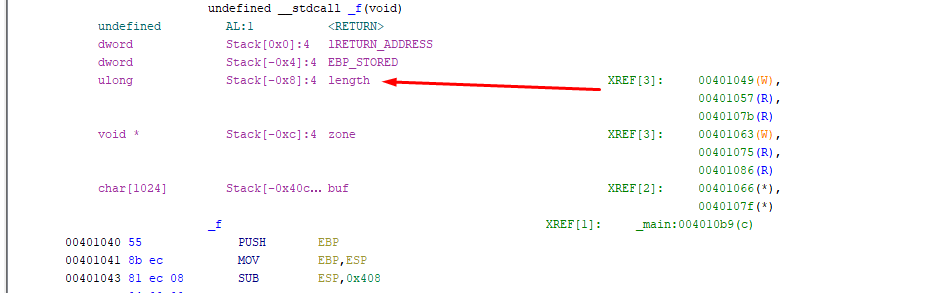
It marks the places where it is used, when it is written to it (WRITE) and the times its value is read (READ).

We see that the first time it is accessed it saves the value 0x64 in it, and then there are two more accesses to use the value, one as an argument of VirtualAlloc, and the other for a use in memcpy.

That last use is to copy the data that enters the buffer by using **gets,** to that allocated memory of 0x64.



Also show the references to the **length** variable here.

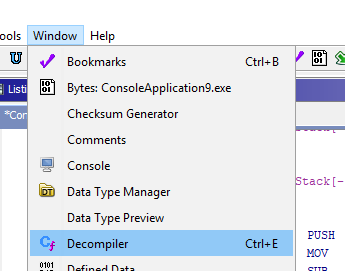


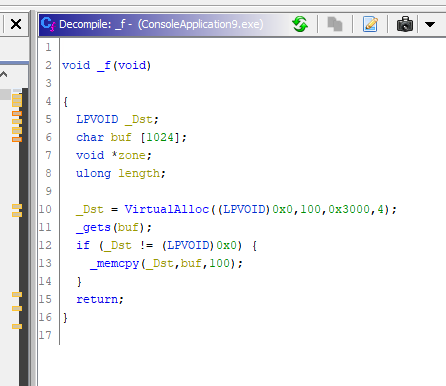
With the use of **gets** we have an overflow since it will copy without control.

Initially in the **memcpy** there is no problem because although the input buffer is bigger than 0x64 bits, as it only copies 0x64 bytes to a buffer with size = 0x64, there will be no overflow.

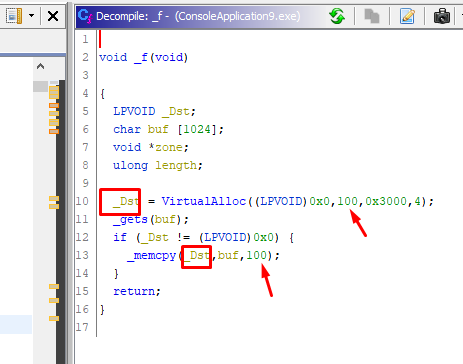
As long as the size to copy is equal to the size of the destination buffer, as in this case, there will be no overflow, since it will not copy more than the size of the buffer.

We can decompile since GHIDRA has a decompiler.



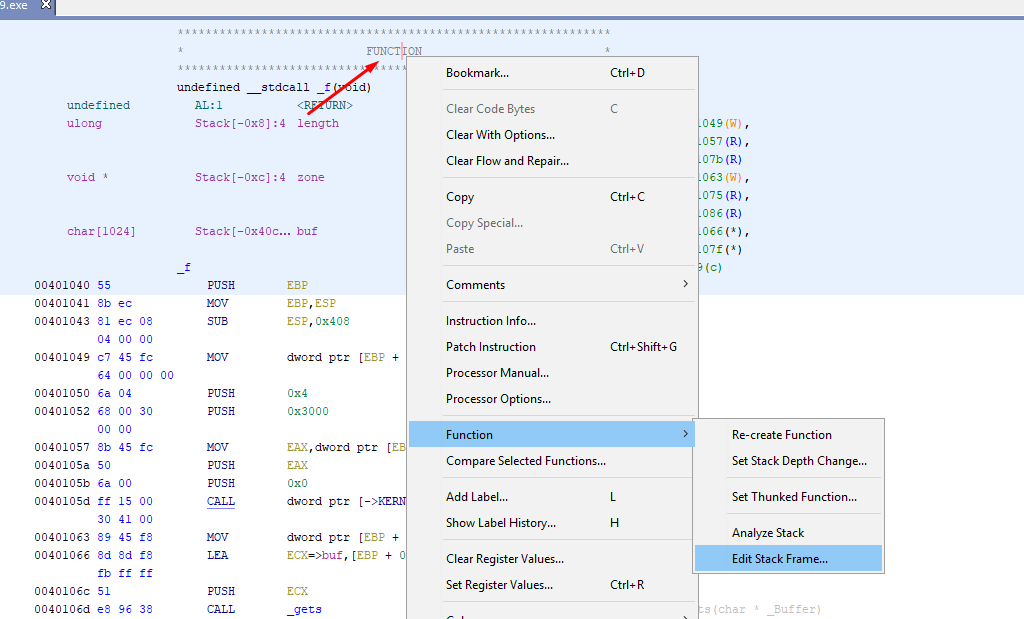


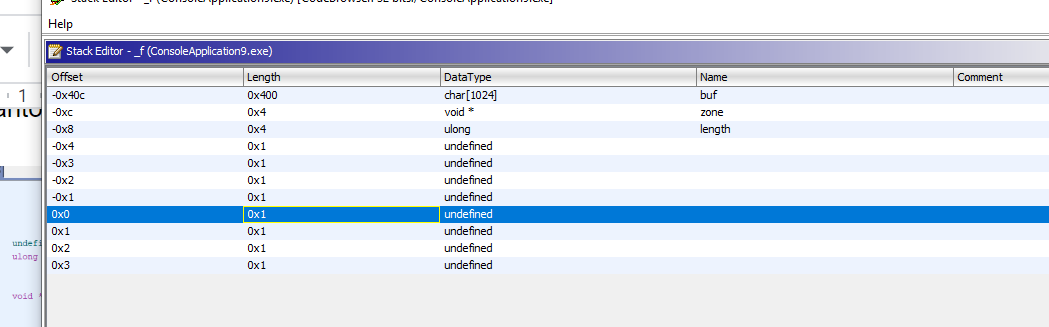
The function was decompiled almost exactly and you can see what we were saying: the call to VirtualAlloc reserves 100 bytes (0x64h), the only thing here is that it does not use the length variable but replaces it with the constant 100 directly.



But as we have seen, the overflow occurs in the gets function. The memcpy function is correctly used with a \_Dst of size 100 bytes and the amount of copied bytes is also 100 bytes, so it will not overflow.

How many bytes must we copy to smash the return address?

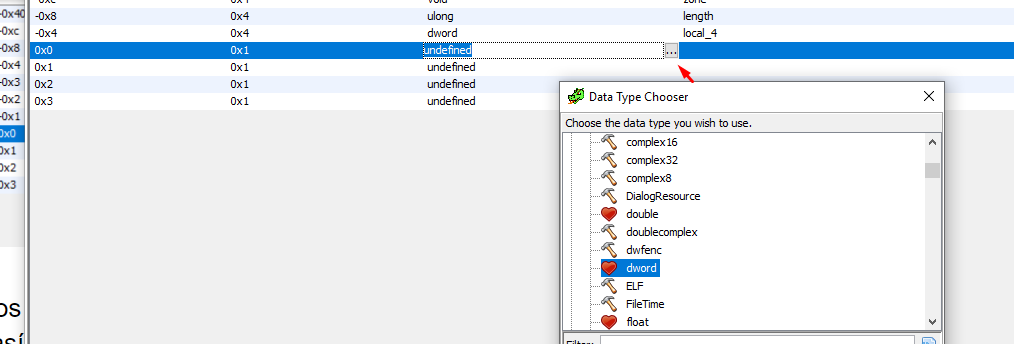


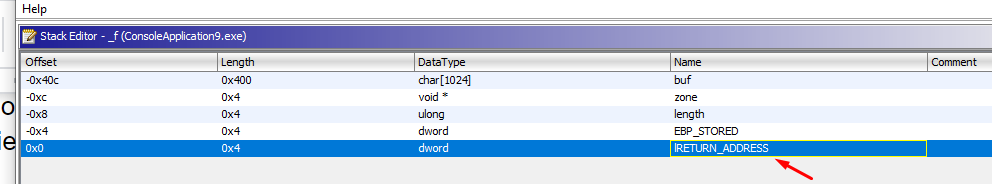


We see the three variables **buf, zone and length**. We also see that **length** is located below **buf,** so as we saw that in the **memcpy** it only copied 100 initially, it will be able to copy more by overflowing **buf** and smashing the value of **length**.

Below are the SAVED EBP and RETURN ADDRESS, each one of 4 bytes.

We change to the type to DWORDS and rename.

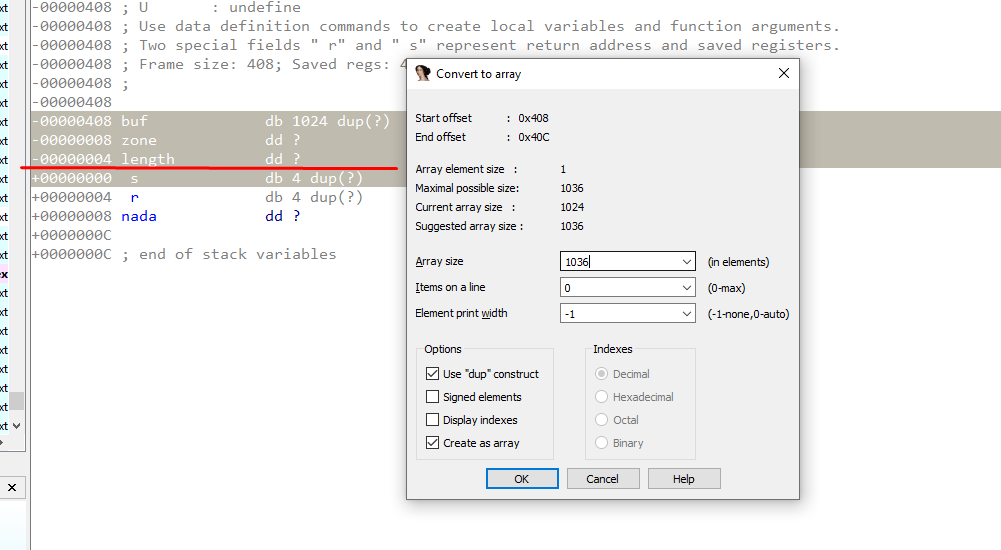




Now it’s clearly displayed.

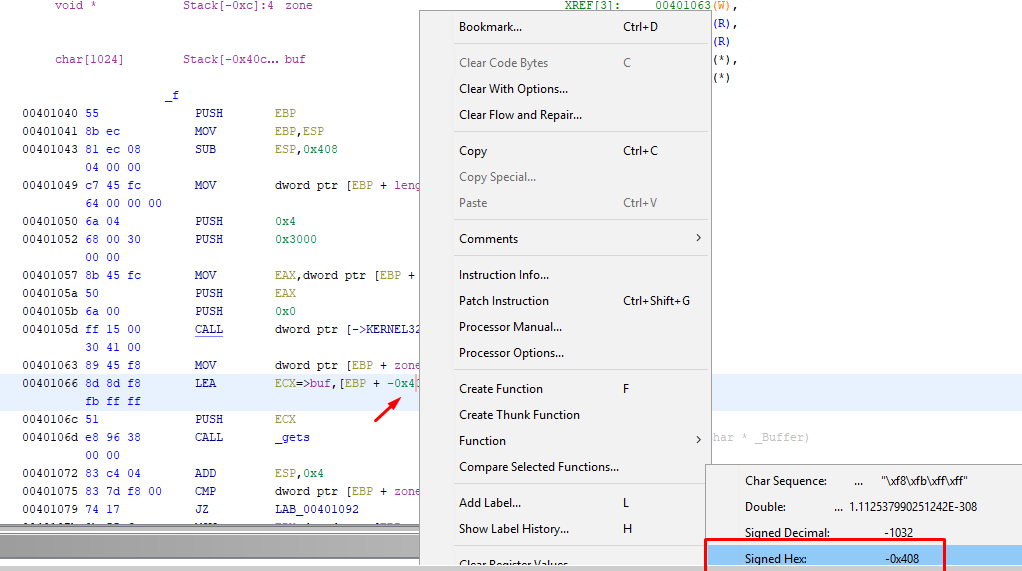
As in GHIDRA, the HORIZON is the value of ESP at the beginning of the function, just below it will be the RETURN ADDRESS that is offset 0x0, and above there are 0x40c bytes.

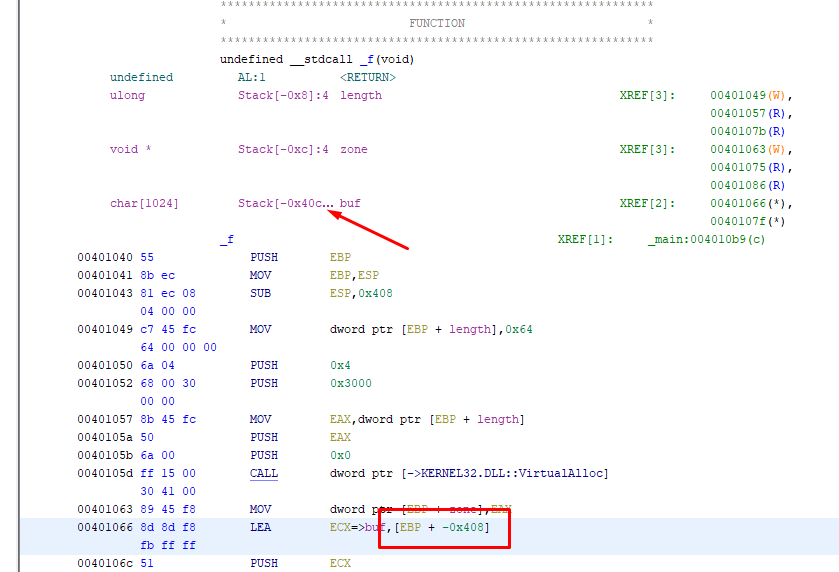
In IDA the HORIZON is the value of EBP after being set in the PROLOG, and with RIGHT CLICK- ARRAY shows us that the data to fill just before the RETURN ADDRESS would be 1036 bytes.



The variable **buf** would be located in IDA at offset = - 0x408, since references are taken from EBP.

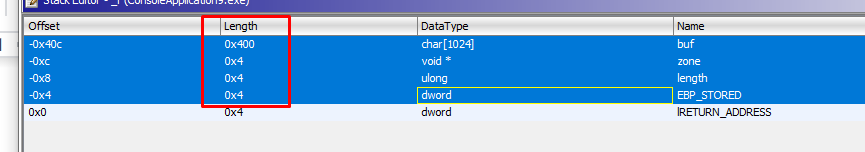
But GHIDRA has ESP as a reference also, bear in mind that GHIDRA has a duality displaying it.





In the usage of the variable, it shows EBP-0x408, taking EBP as a reference like IDA, but in the definition of the function and in the distances, it shows us 0x40C which is the distance to ESP at the beginning of the function. (more like GHIDRA)

And well, you have to take this into account to avoid confusion.



If we add the lengths of the variables to fill until just before the RETURN ADDRESS it would be

0x400 + 4 +4 +4 = 1036

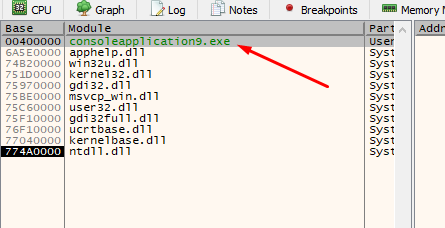


So if one does not get confused and knows what one is doing, one gets the same result in IDA as in GHIDRA.

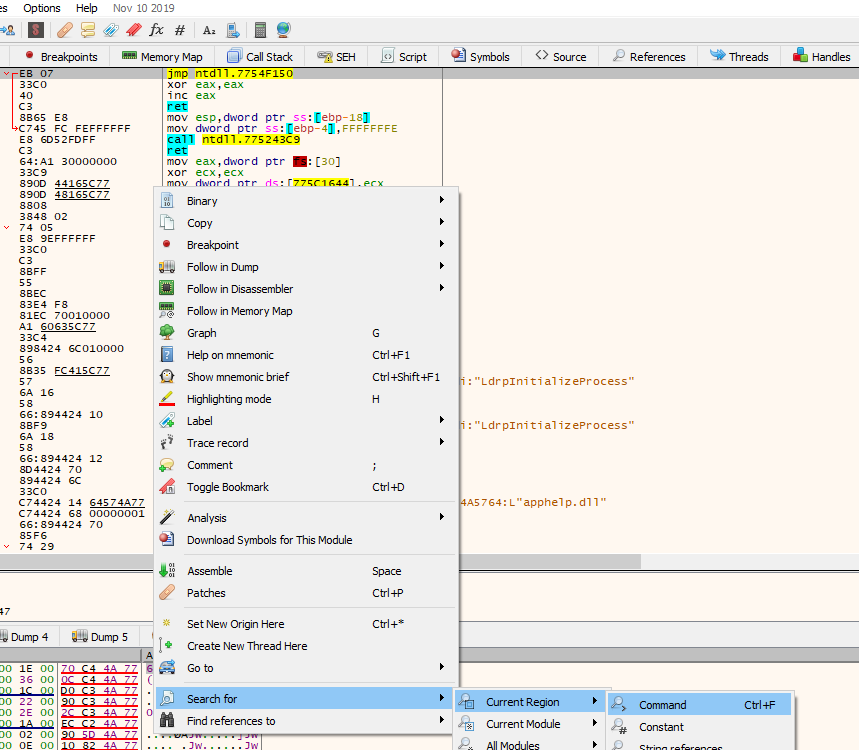
If I didn't have DEP activated, the script to exploit this exercise would be.

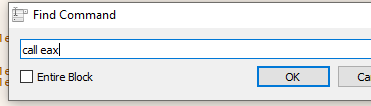


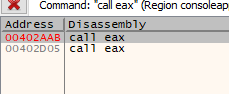
We open it in x64dbg, and we can use VIEW-MODULES to locate the executable.



In the executable we find a CALL EAX instruction, since EAX remains with the address of **buf**, to jump there to execute.

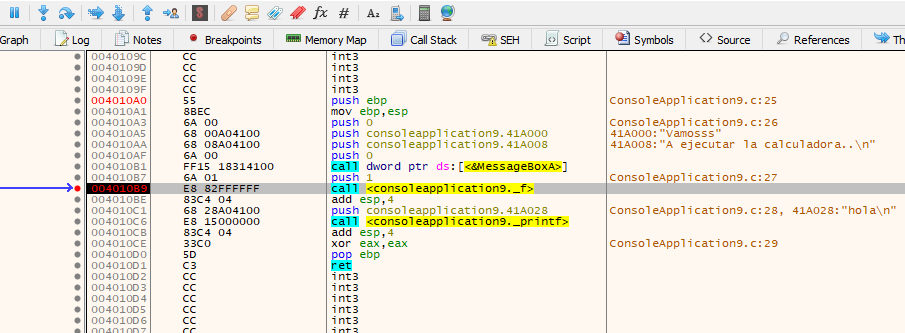






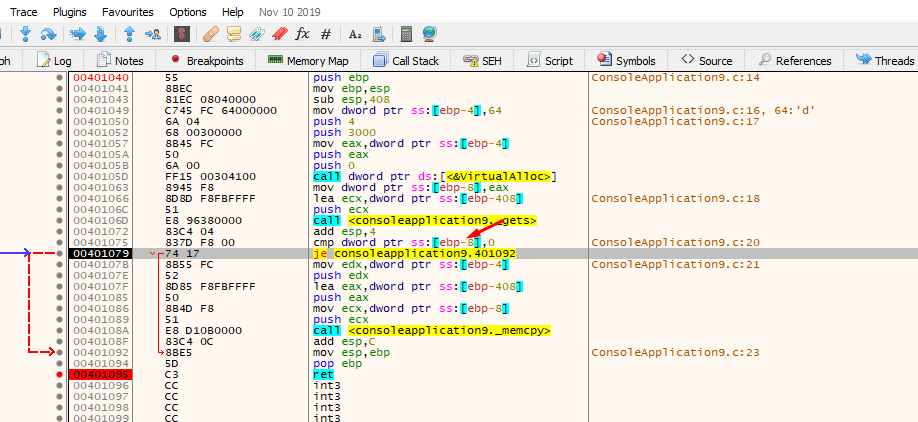
Using any of those CALL EAX and placing the SHELLCODE at the beginning of the buffer, if it did not have DEP activated, it could perfectly jump to execute.

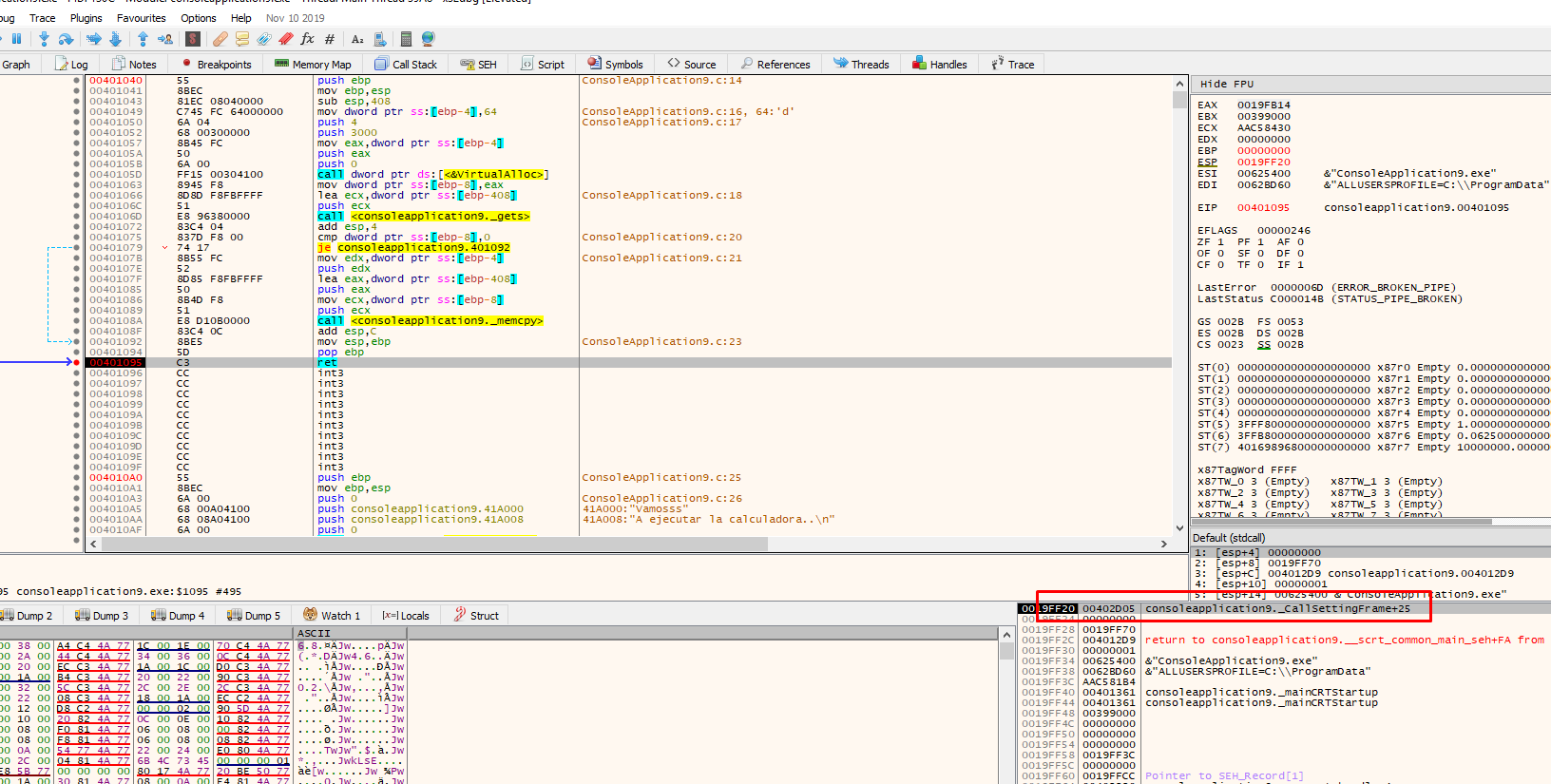
Let's run it, attach the x64dbg, put a breakpoint when returning from the MessageBoxA and the program stops there.



Trace with F7 to enter function **f**.

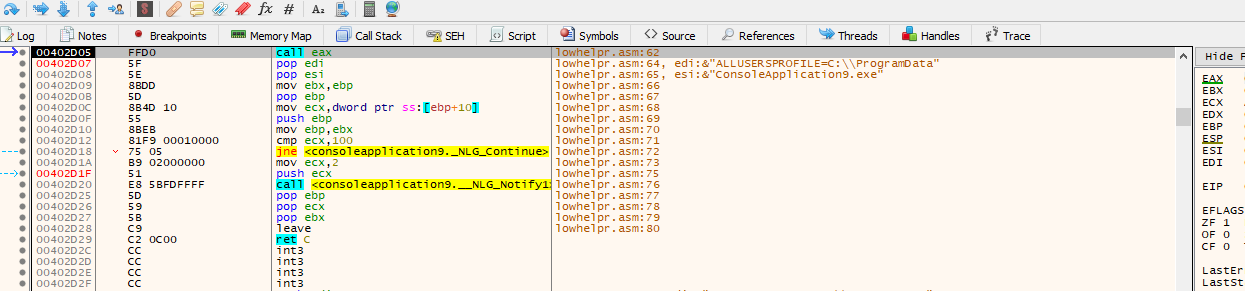
As the buffer was filled with the shellcode and zeros to complete, the value of flag which makes it go to memcpy or not, is zero, jumping above memcpy.



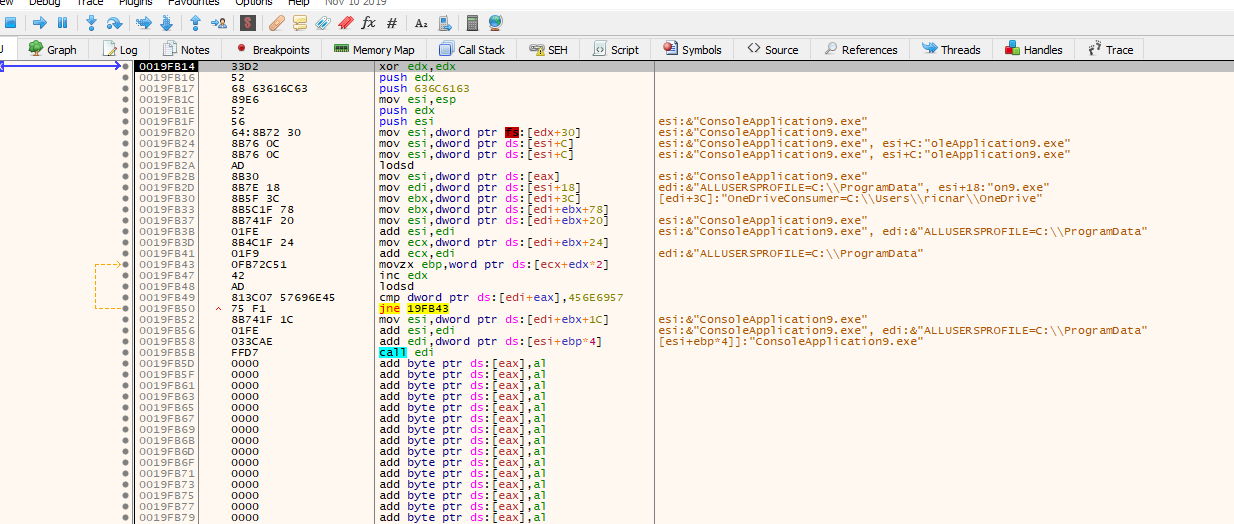


When we reach the return address, it points to the CALL EAX.

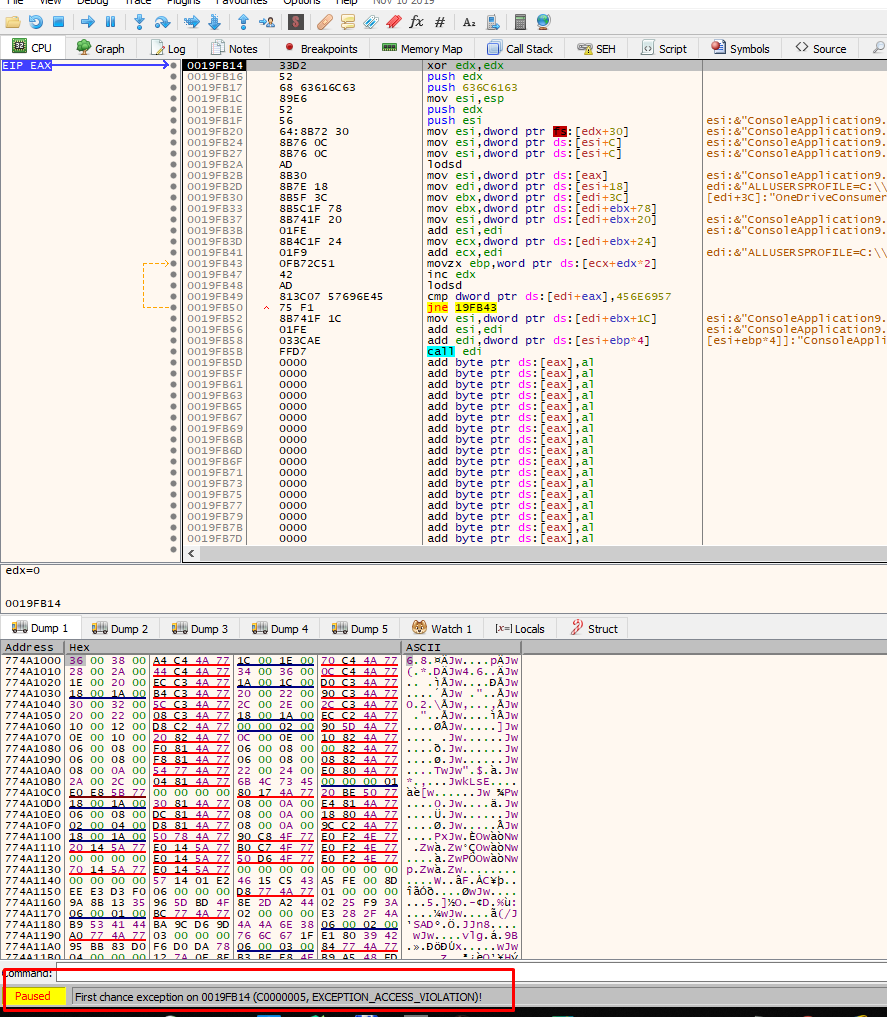
If I keep tracing with f7, the CALL EAX will be executed, because it is code that belongs to the code section of the executable module, which has execute permission.



If I continue tracing with f7, I reach the shellcode.



If I want to continue executing my code, since the stack does not have execution permission, it will crash here.



And it's game over, it will not execute the shellcode and it will close, this is what DEP protection is about.

But as we saw the CALL EAX was executed, since it belongs to the code section, it has execution permission, otherwise the program itself could not be executed.

PRACTICE MINIROP

So what we have been seeing, is the basis of the ROP.

I want to write a script with a sample mini ROP that set this values

EAX = 0x41414141 ECX = 0x42424242 EBP = 0x43434343.

I can find three gadgets

1. POP EAX-RET
2. POP ECX-RET
3. POP EBP -RET

I could also find a single gadget that sets multiple registers, for example

POP ECX-POP EAX-RET

Let's see what we found in the code.

We will use a FREE tool to find GADGETS that stores in a file all the gadgets of the module.

It's called RP ++

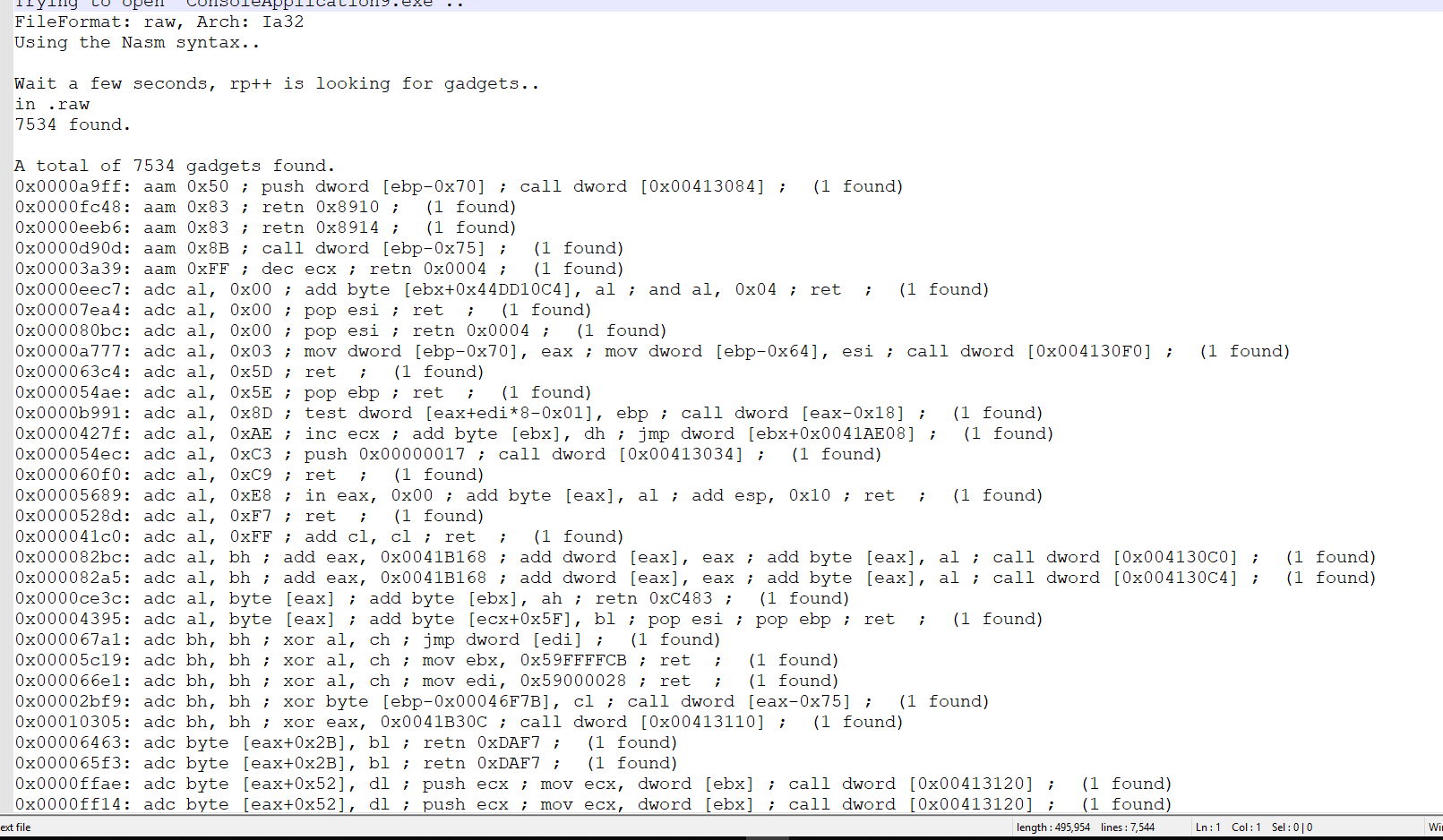
<https://drive.google.com/open?id=1M3LeiU5WzbsEqSnSwrEJupKCH_2wORnV>

We put the executable in the same folder to make it easier.

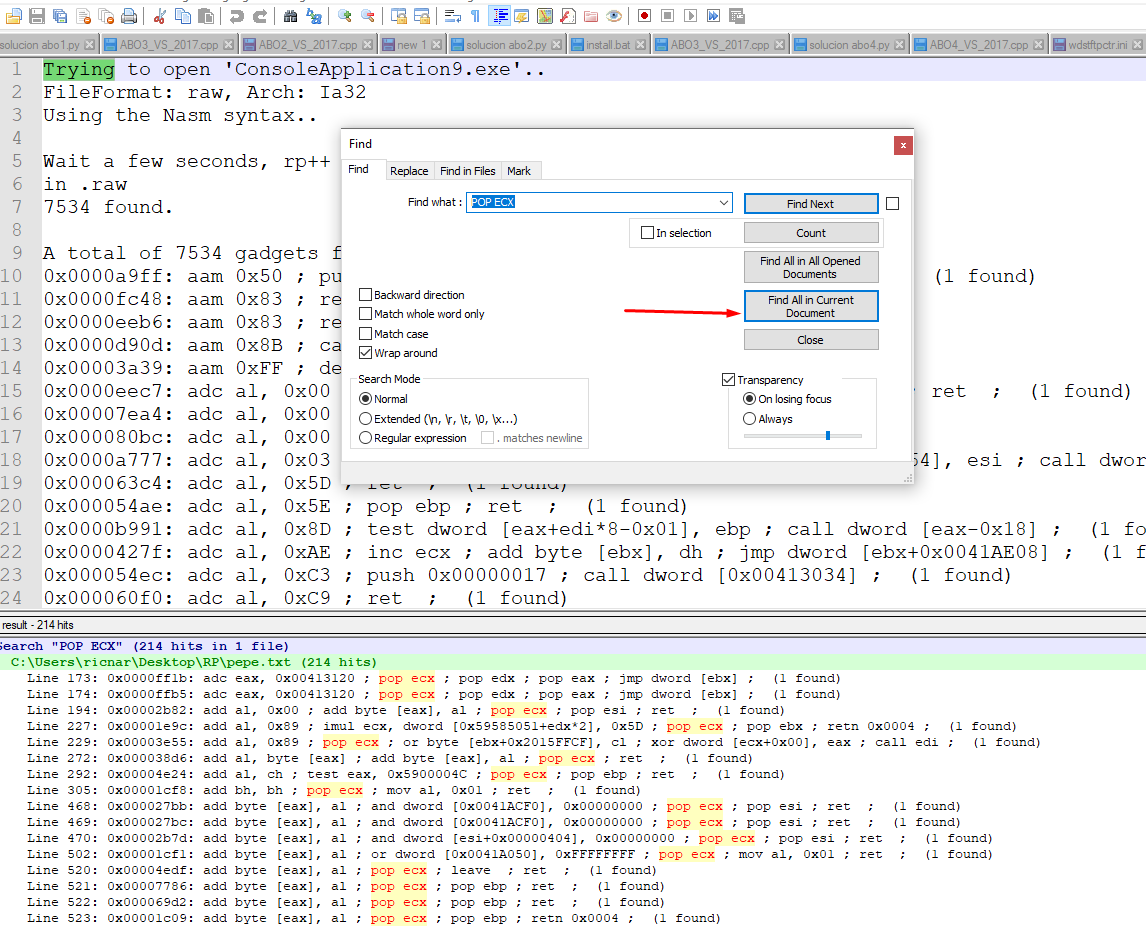


**rp-win-x86.exe --file = ConsoleApplication9.exe --raw = x86 --rop = 4> pepe.txt I**

We must use the 32-bit executable **(rp-win-x86.exe),** and we must look for 32-bit code (**raw = x86)**, we find 4 instructions maximum length **(rop = 4).** and we save in a file **(pepe.txt).**



I search in NOTEPAD++, all POP ECX to see what comes out.



There I have the gadgets.

0x00004828: pop eax; pop ebp; ret; (1 found)

0x000033e3: pop eax; pop ebp; ret; (1 found)

0x00005545: pop eax; pop ebp; ret; (1 found)

0x000108d9: pop eax; pop ebp; ret; (1 found)

0x00011126: pop eax; pop ebp; ret; (1 found)

0x000110ad: pop eax; pop ebp; ret; (1 found)

Also

0x00004461: pop ecx; pop ebp; ret; (1 found)

0x0000447c: pop ecx; pop ebp; ret; (1 found)

0x00004c65: pop ecx; pop ebp; ret; (1 found)

0x00004e2b: pop ecx; pop ebp; ret; (1 found)

0x00007788: pop ecx; pop ebp; ret; (1 found)

0x000069d4: pop ecx; pop ebp; ret; (1 found)

0x00007648: pop ecx; pop ebp; ret; (1 found)

0x00009646: pop ecx; pop ebp; ret; (1 found)

And

0x00000718: pop ecx; ret; (1 found)

0x000005dd: pop ecx; ret; (1 found)

0x00001820: pop ecx; ret; (1 found)

0x00002849: pop ecx; ret; (1 found)

0x00003d06: pop ecx; ret; (1 found)

0x000038da: pop ecx; ret; (1 found)

0x000044b9: pop ecx; ret; (1 found)

0x00004592: pop ecx; ret; (1 found)

0x000048c7: pop ecx; ret; (1 found)

0x000049f2: pop ecx; ret; (1 found)

0x00004a4d: pop ecx; ret; (1 found)

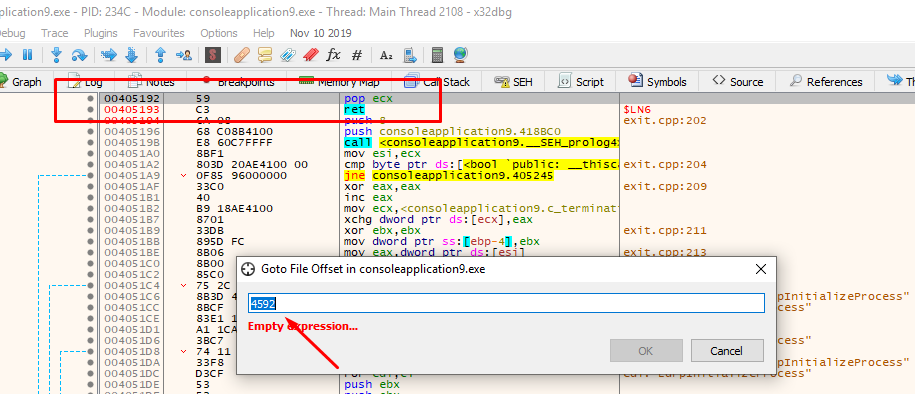
0x00004d55: pop ecx; ret; (1 found)

0x00004dfc: pop ecx; ret; (1 found)

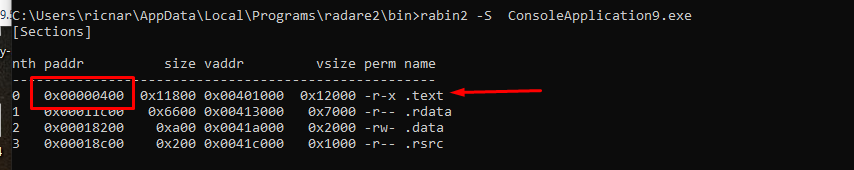
We can build the rop with those gadgets to test, the problem is that this tool returns the file offset, since it opens the file statically.

In previous parts of this course, we have already seen how to calculate the virtual memory address from the file offset.

If you want to do it quickly with the x64dbg, go to GOTO-FILE OFFSET put the file offset of the gadget and get the virtual address.

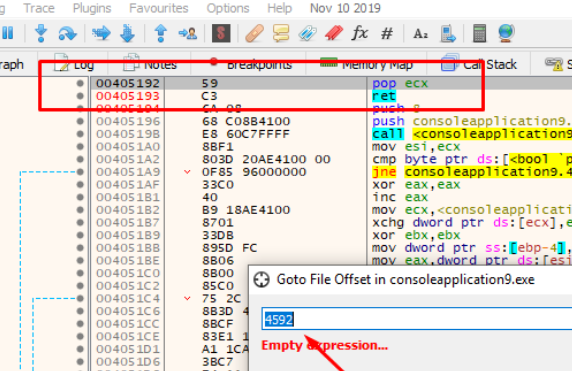


If we want to calculate it, we see with rabin2 that the code section begins at 0x400 on disk, so subtracting from the value of the file offset - 0x400 gives us the offset within the start of the first section on disk, and adding where the code section begins at memory (the image base plus the size of the header 0x401000) should correctly give us the virtual address.



0x4592- 0x400 + 0x401000 = 0x405192





How I want my ROP to set these values:

**EAX = 0x41414141 ECX = 42424242 EBP = 43434343 I will**

I will use this other gadget too.

0x00005545: pop eax; pop ebp; ret; (1 found)

virtual address = 0x406145

hex (0x5545- 0x400 + 0x401000)

'0x406145'

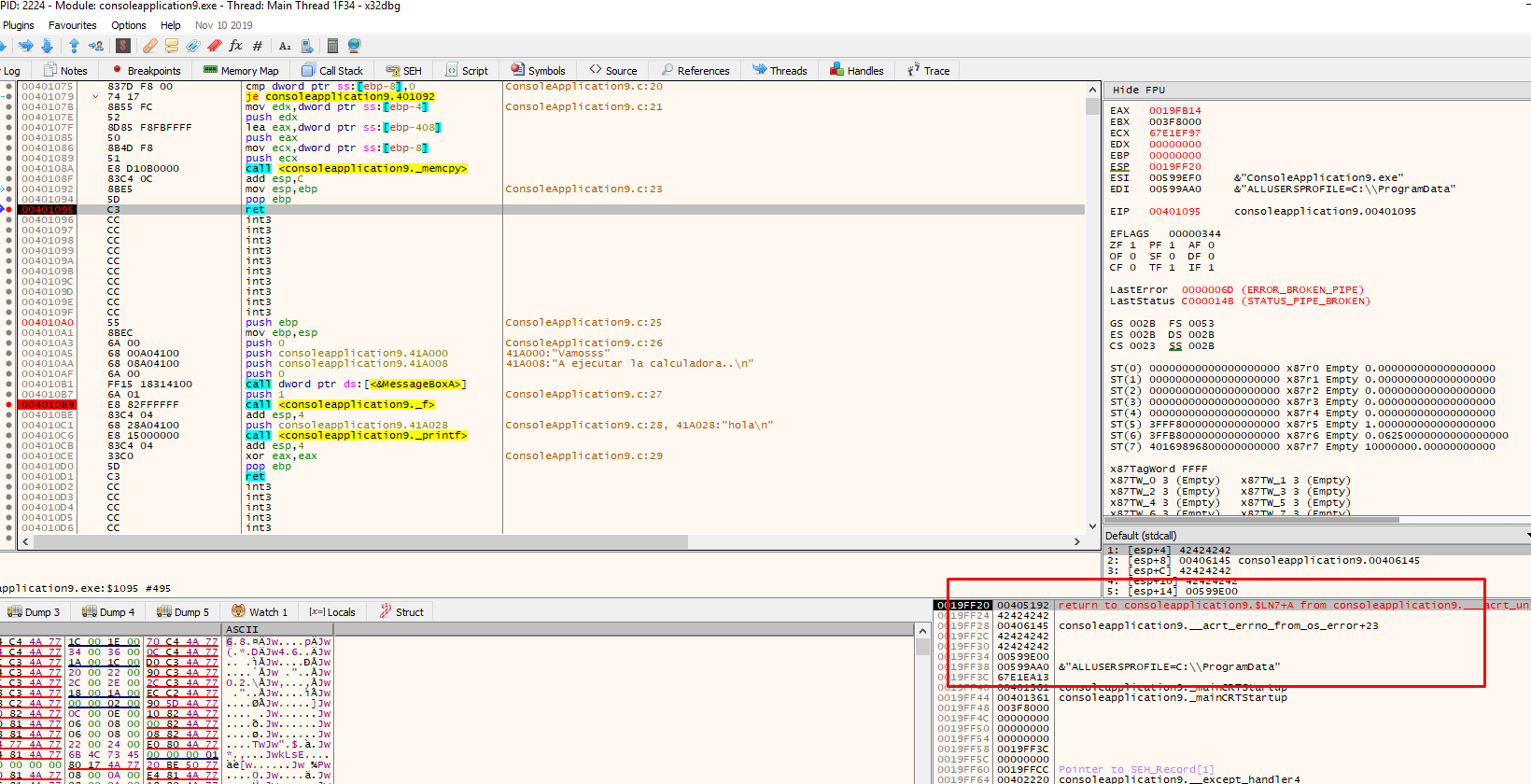
The script is as the image shows.



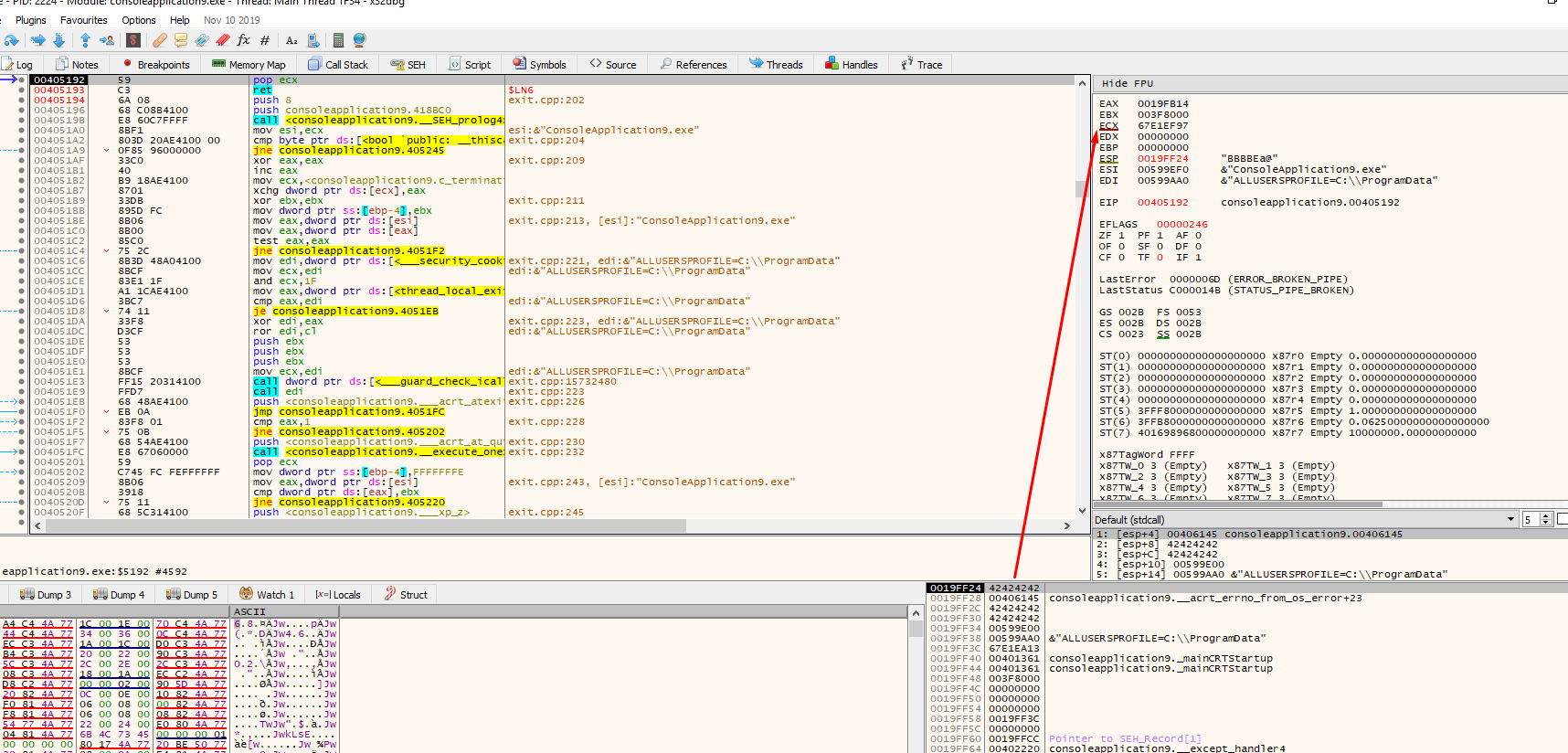
There we see the two gadgets, and the values​ interspersed, that it will read with the POPs and move to the registers.

Let's run it to see if we got what we wanted.

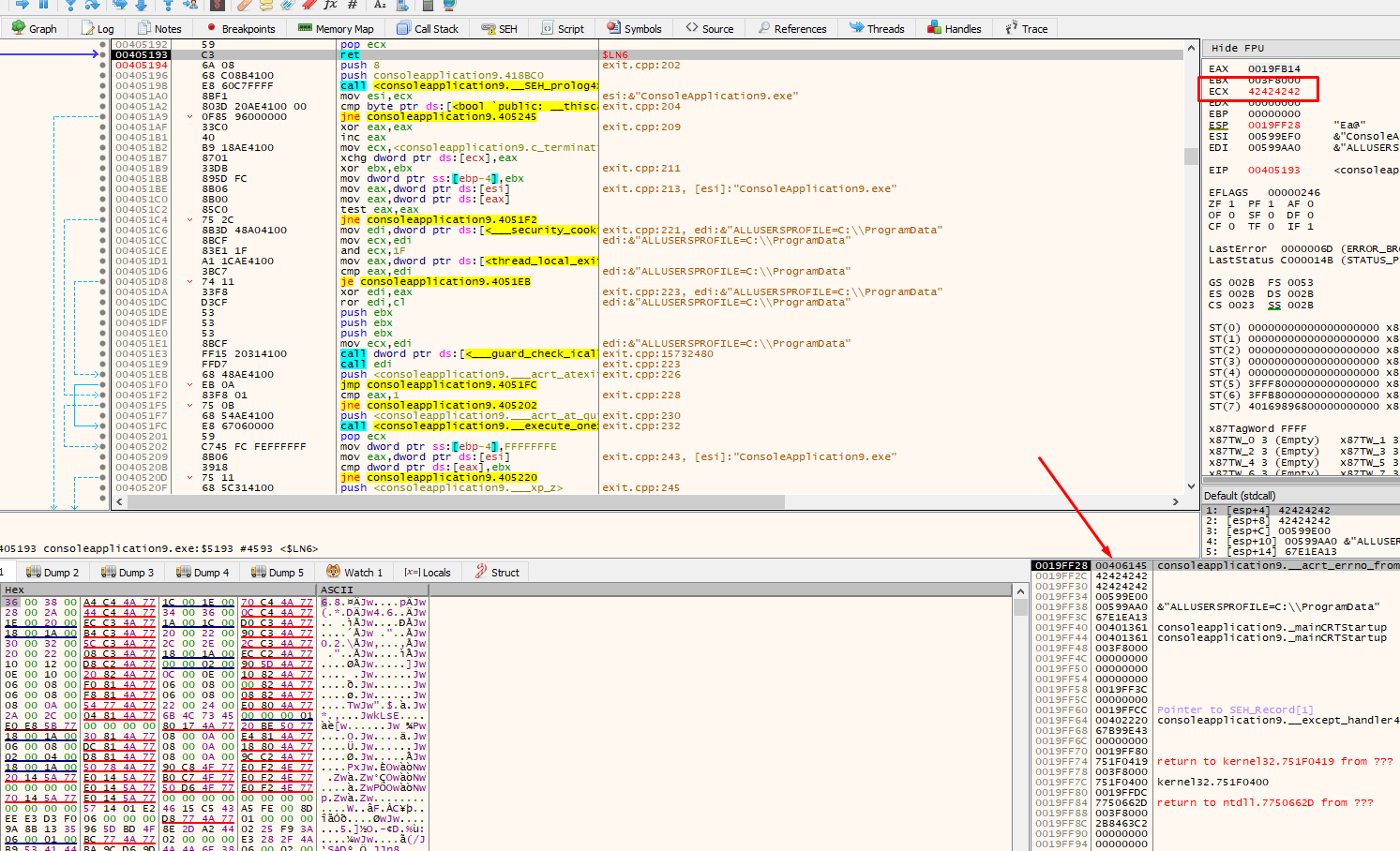
When I get to the RET I can see my ROP, and I can trace it with f7.



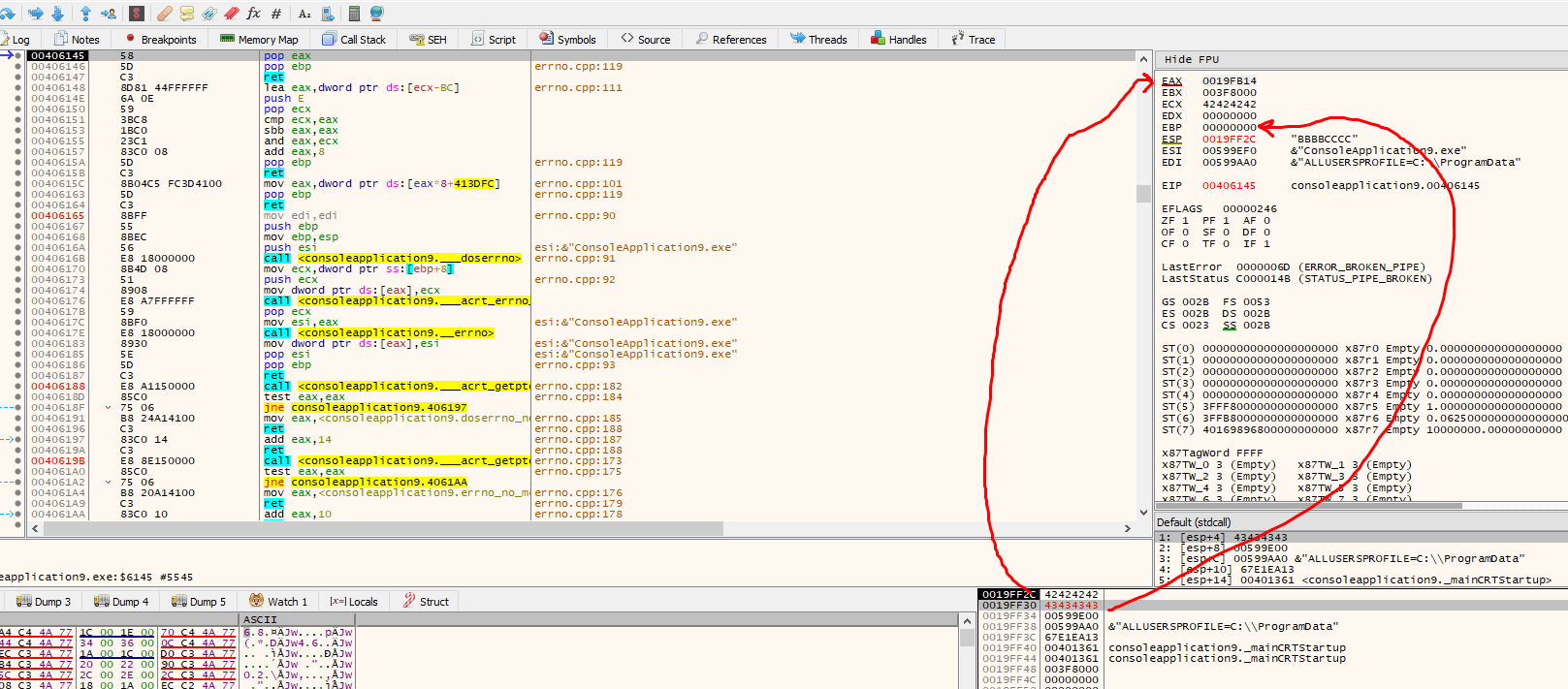
The first gadget is POP ECX-RET



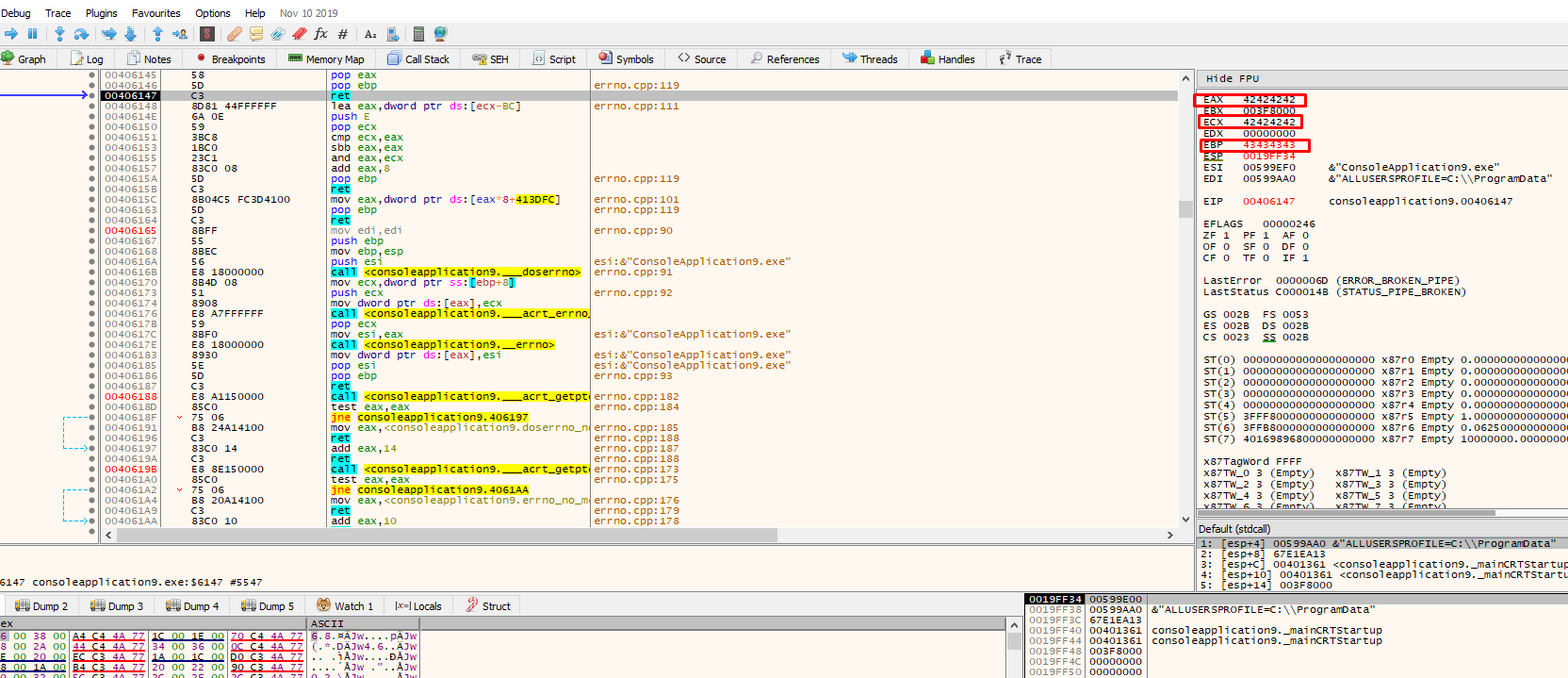
It moves the value 0x42424242 to ECX, with the POP ECX and when you reach the ret instruction the next gadget will be executed.



Executing the second GADGET.



It will move 0x42424242 to EAX and 0x43434343 to EBP.



We see that we managed to chain several GADGETS, and put the values ​​we wanted in the registers, where we needed them.

And if we wanted we could continue to put more GADGETS underneath and do different things.

In part 11 we will see the complete ROP to bypass the DEP for this exercise, I think that this was enough for today.

See you in part 11.

Ricardo Narvaja

03/06/2020