EXPLOITING AND REVERSING

USING FREE TOOLS

(PART 11)

ROPING STEP BY STEP

There are usually tools that can automatically build an ROP for simple cases.

In difficult cases, these tools generally do not solve them or only partially do so, leaving one to complete by hand the work that the tool could not do.

What points does an EXPLOIT WRITER look at, to know if an ROP will be difficult or easy?

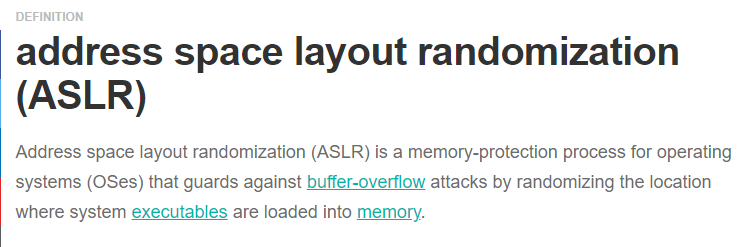
Here is a list of points to observe, the more YES we can answer the easier it will be, while some NO of this list will complicate the work, some more others less. Let's look at the list, then review the responses and their consequences.

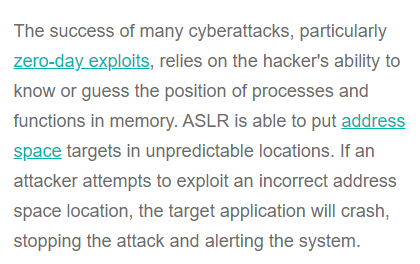
Answer these questions before building an ROP with YES or NO (listed in order of importance).

1. Does the process have modules that do not have ASLR?
2. Do you have the VIrtualAlloc or VirtualProtect function imported in any module that does not have ASLR?
3. Is the data already located in the stack to start roping?
4. Can I pass any character or is there no invalid characters or are there few?

To answer the first question, let's first see

WHAT IS ASLR?





The executables compiled with ASLR are not always positioned in memory at fixed addresses, which complicates the operation, because if all the executables and DLLs of a process are compiled with ASLR, we will not have fixed addresses where we can jump and be able to build an ROP.

In other words, the first measure is to see if there is any module without ASLR, since it is not set by the complete process, but each executable when compiled, individually can be compiled with ASLR or not.

One possibility to avoid DEP + ASLR protection is to find some memory address leak in the system or in the same process, returning us one address of some module at runtime, allowing us to build the rop, according to this direction obtained by the leak.

As we go step by step, we will always start with the simplest cases first, let's see if in our process there is any module without ASLR

Has the process one or more modules that do not have ASLR?

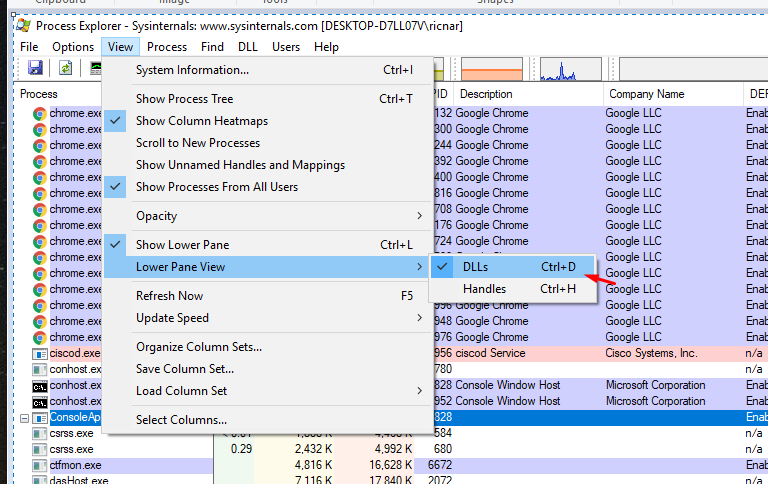
We run the executable of the exercise and look again in the PROCESS EXPLORER.

The process explorer 16.31 found on the Microsoft page has a bug that does not work showing the ASLR of each module.

I put the link to version 16.21 that if it works and is the one that I use:

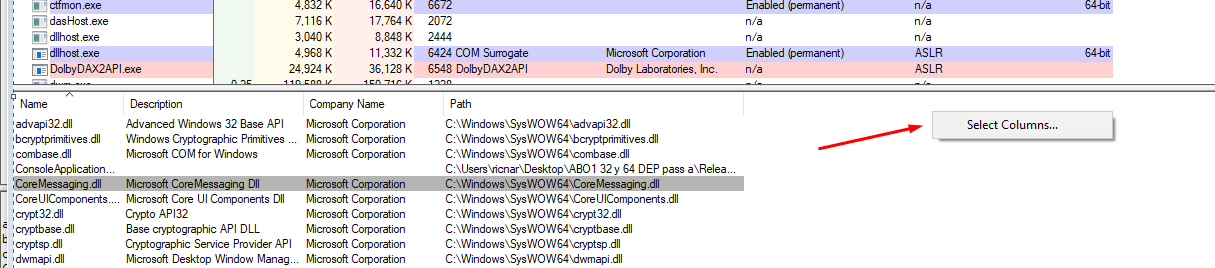
https://drive.google.com/file/d/1cgF49ZS\_GUskxCUJ7ZLDEsoz710mq106/view?usp=drivesdk

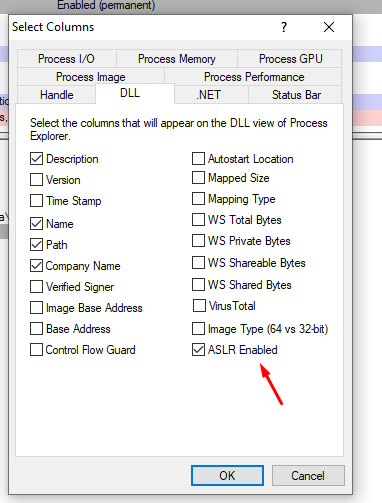




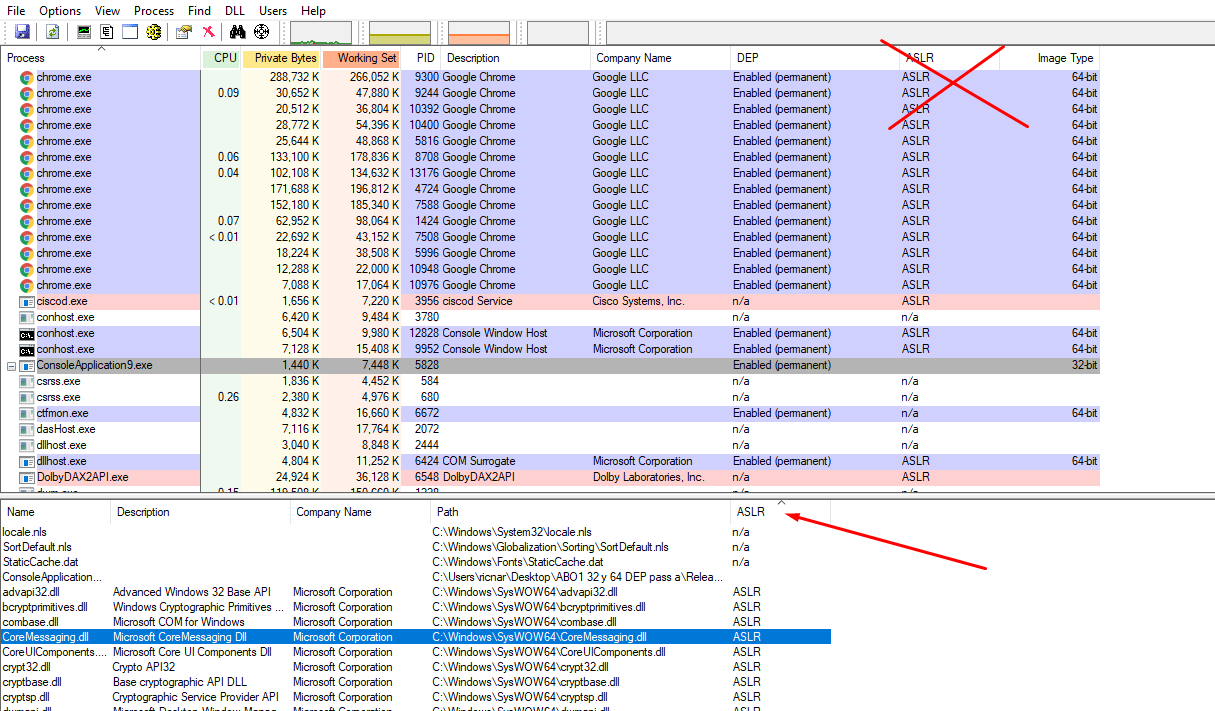
We are going to configure it so that the modules of a process can be seen at the bottom.

At the bottom, we right-click on the columns and choose SELECT COLUMNS.





We choose “ASLR Enabled” and check it.

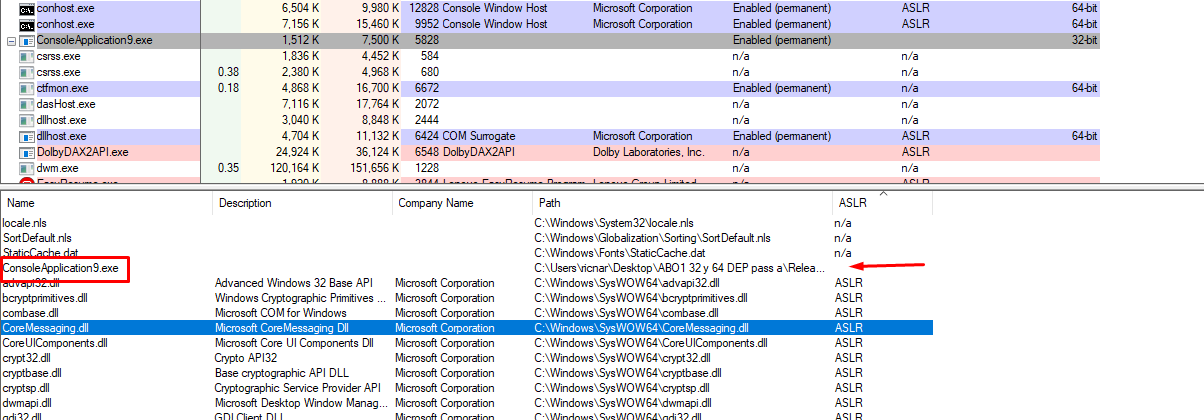


So we see at the bottom, the important column that marks whether each module was compiled with ASLR or not.

The column at the top, also called ASLR, will not serve us because in the same process, there can be modules with and without ASLR, so the generic value for the whole process does not work.

If I click on the ASLR column that we add, to sort the modules in case it has ASLR or not, we see that there is one that does not have ASLR, that is, its addresses will be fixed and we can use it for rop.

Any DLL or EXE without ASLR that is in the lower list of our process will do.



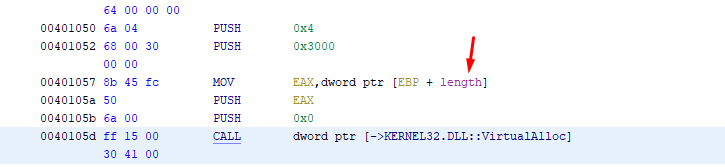
In our case, the executable itself was compiled without ASLR.

So, the first and most important question on the list is YES, and we know that the maximum difficulty or even the impossibility of roping if we can't leak an address, is avoided.

Let's continue with the second question.

Does the process have the VirtualAlloc or VirtualProtect function imported into any module that does not have ASLR enabled?

Seeing that the code we analyze the answer is YES, VirtualAlloc is imported, let's remember from the analysis of part 10, that in our executable module without ASLR there was a call to VirtualAlloc.



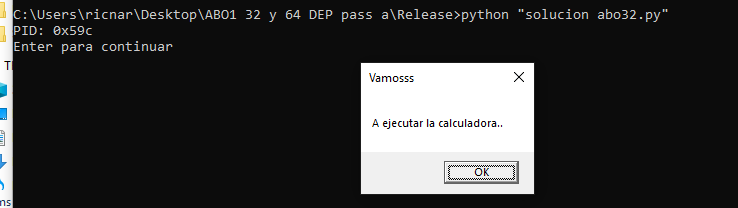
If neither of the modules without ASLR has imported neither VirtualAlloc nor VirtualProtect, we can also solve it, but the ROP will be longer and more complex because we will have to build a call to GetModuleHandleA and GetProcAddress, and our work will be more complicated.

So we already have two YES.

Let's look at the third question.

Is the data already located in the stack to start roping?

Let's run the partial solution we made so far.



Let's attach the X64Dbg and get to the RET that jumps to the first GADGET of our minirop.

There we see that our ROP is located on the stack ready to jump to the first GADGET.

If this were not the case and our ROP was not located on the stack, and we had a single jump to a single possible address, we will not be able to chain more GADGETS, because when the stack is not being controlled by us, when we finish executing the first GADGET and we reach its RET, we will not be able to chain a second GADGET and everything would end there.

For that purpose, there is a special type of gadget called ROP PIVOT, whose function is to move our data to the stack to continue ROPING normally. After that, when it reaches its own RET, everything should be in place to continue running the following GADGETS.

Later we will see examples of the use of ROP PIVOT.

For now the answer to the question is YES, and therefore in our case it is not necessary to use an ROP PIVOT to start ROPING.

So we already have three YES, let's look at the fourth question.

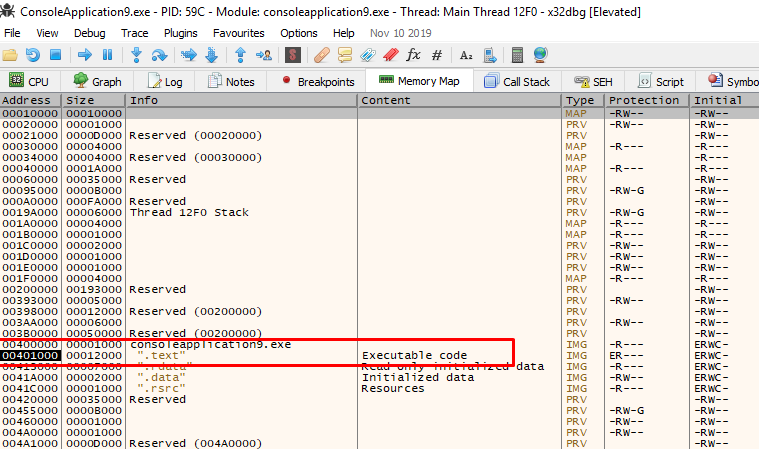
There are no invalid characters? Can I use any character I want?

The more invalid characters I have in my process to exploit, the more complicated the ROP will be, sometimes even making it impossible depending on the case.

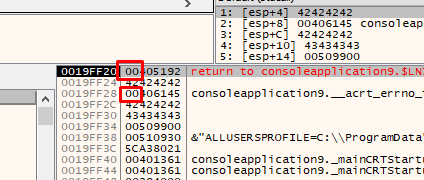
Obviously, as we must locate addresses of our module without ASLR in our ROP, if these addresses have a character that we cannot pass, things can get complicated

For example, the executable section where we can find gadgets in our example, is located from 0x401000 to 0x413000, only there we can find GADGETS since it is the only module without ASLR and its executable section is located there.

If, for example, 0x00 were an invalid character, it would prevent us from jumping to GADGETS in that section because 0x00 is essential to put together the address to jump to.

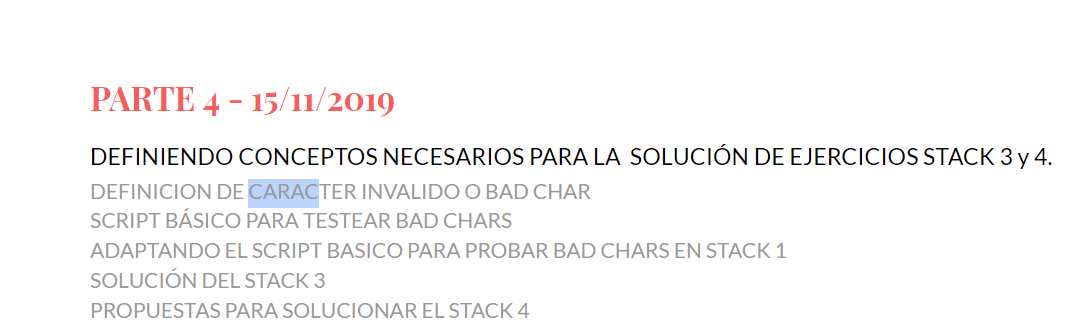


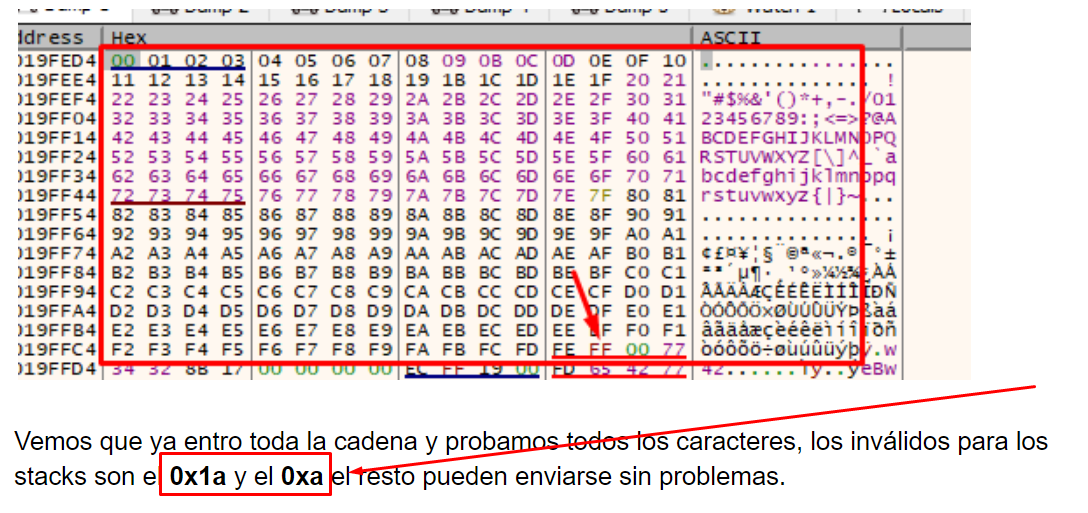
As we see in our minirop, the zeros were necessary, since the gadgets are located in addresses that start with zero.



If we had 0x00 as an invalid character in our case, we could only rop if we can leak a module address.

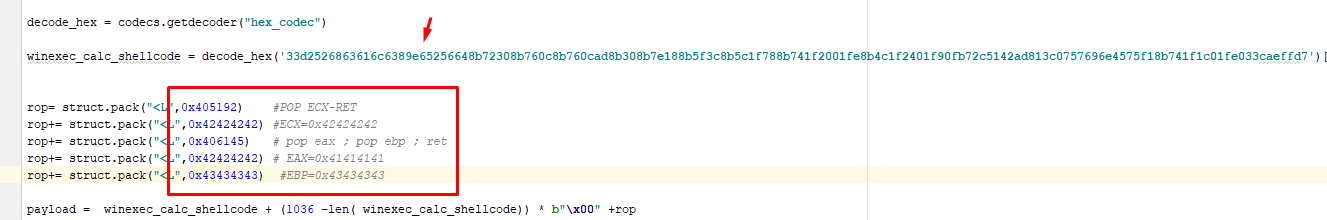
Now let's remember our study of the invalid characters we did in part 4, since the function to enter the data is still **gets()**, just like the exercise in part 4.





Therefore, we have some restriction, which is not being able to use addresses that contain 0xa and 0x1a, but we will not be able to use those characters in the values ​​that we move to the registers either.

In our minirop, we move the values ​​0x41414141, 0x42424242 and 0x43434343 to ECX, EAX, and EBP without using any 0x1a or 0xa. Also when we build the ROP, we must take into account this restriction, there should be no invalid characters anywhere in the ROP or in the shellcode.



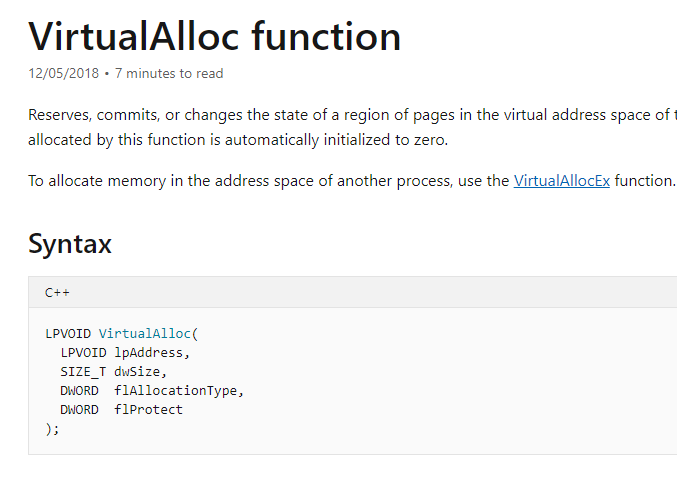
Therefore, when building the ROP there can be no 0x1a or 0xa in the ROP or in the SHELLCODE, the answer to the last question in our case is a NO, but it is a NO that affects little, if they included the 0x00 it would be much worse, fortunately, the restricted characters in our example are not very important.

So we are in one of the easiest cases, later we will increase the difficulty with cases where there is more NO.

We will search for the gadgets and, using x64dbg, we will convert the FILE OFFSETS to their virtual addresses.

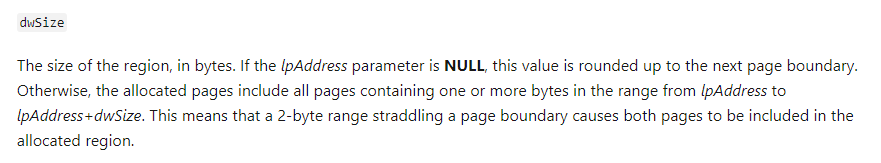
In this case the problem we have is that the RP ++ if there is a gadget ended with a CALL, it does not show us what instruction follows, which can be useful for most of the cases but in our particular case complicates us a little.

Let's look at the VirtualAlloc function on MSDN.



Consequently, the first argument that should be on the stack would be the address to which we should add execute rights (lpAddress).

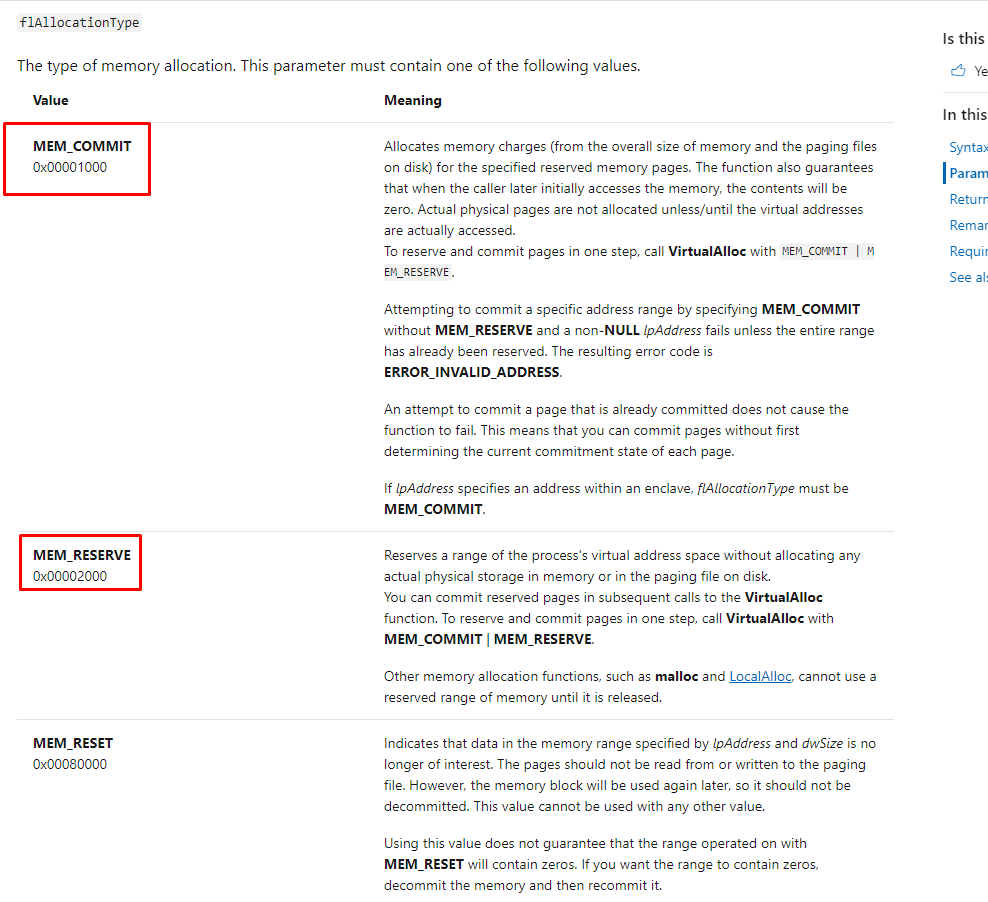
Then the second argument is dwSize.

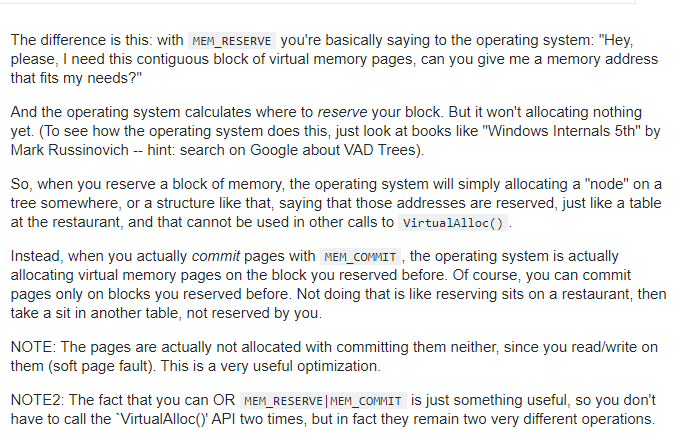


In a practical way, we will unprotect a memory page or more depending on the size that we place, if for example we put the value 1, 0x1000 bytes will be unprotected (0x1000 is generally the size of the memory pages) placing any size between 1 and 0x1000 will unprotect 0x1000, a size between 0x1001 and 0x2000 will unprotect 0x2000 and so on.

In any case, it is not convenient to put a very large value because if this value added to the lpAddress falls outside the section, the function will return an error, while if I put a lower value the check that makes the function at its beginning will pass, and it will work unprotecting the complete remaining page.

The third argument is flAllocationType.



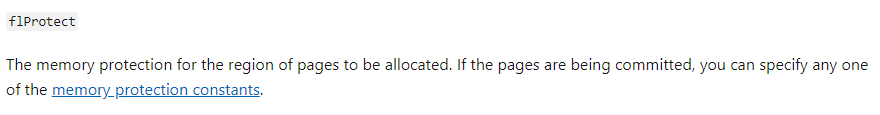


We see that if we create a new section, we need to call twice, the first time using MEM\_RESERVE (0x2000) to reserve it, and the second time using MEM\_COMMIT (0x1000) to finally assign it, the good thing is that you can do an OR between the two values ​​and do the two operations in one.

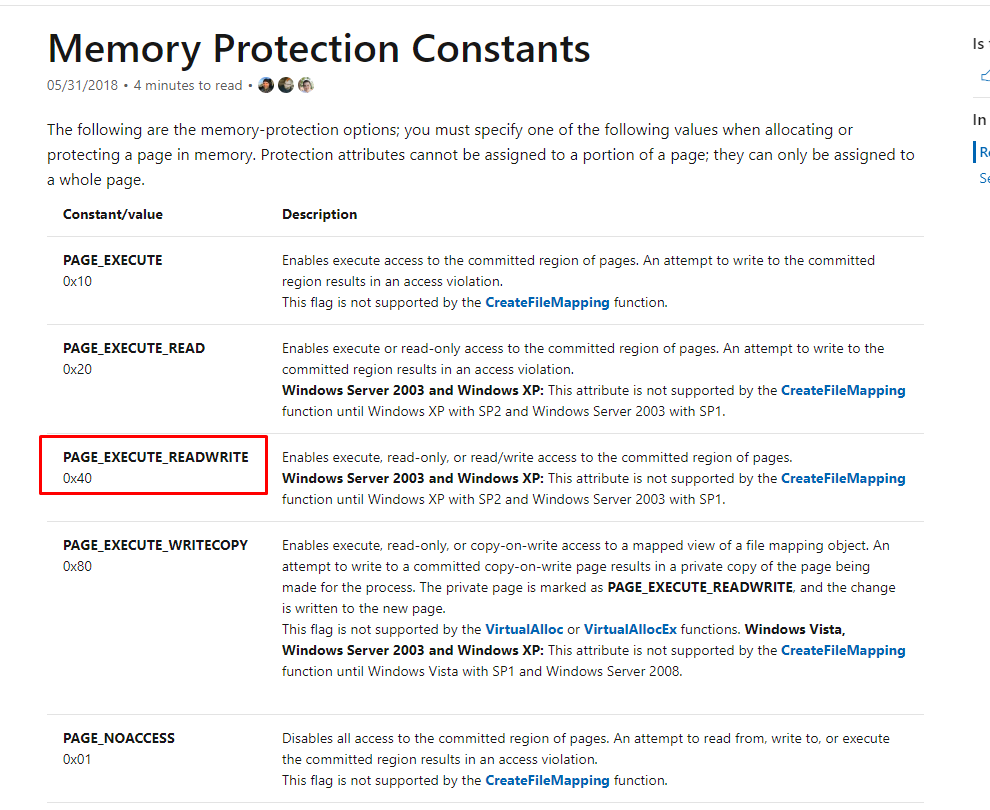


In the case that we use VirtualAlloc in an existing section, as in the stack, we must use only 0x1000, since it does not need to RESERVE, only COMMIT is necessary.

The fourth and final argument is flProtect.



Since we need the region to be executable, let's look at the constants for it.



It is clear that we must pass 0x40 to give it RXW permission.

We see that the last three arguments in our case can come directly in our data, we can send them and there will be no problems, since 0x0 is not an invalid character.

The 4 arguments for VirtualAlloc unprotecting the stack would be

lpAddress =?

dwSize= 1

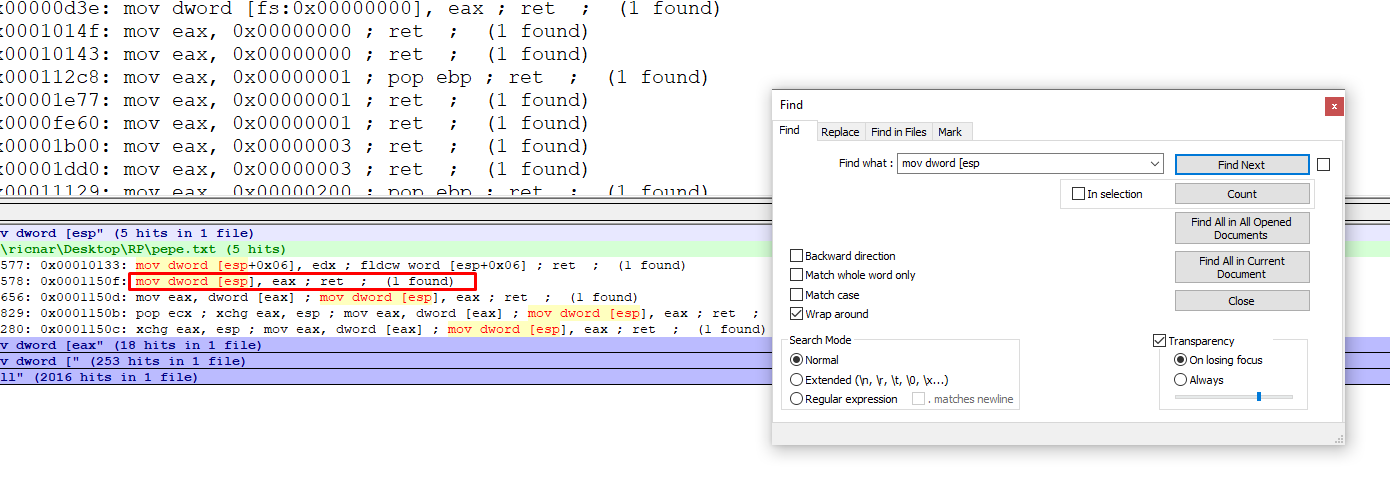
flAllocationType = 0x1000

flProtect = 0x40

We see that the only value we do not know is the first of the arguments (lpAddress), since we have no problem sending 0x1, 0x1000 and 0x40, which are known and without invalid characters.

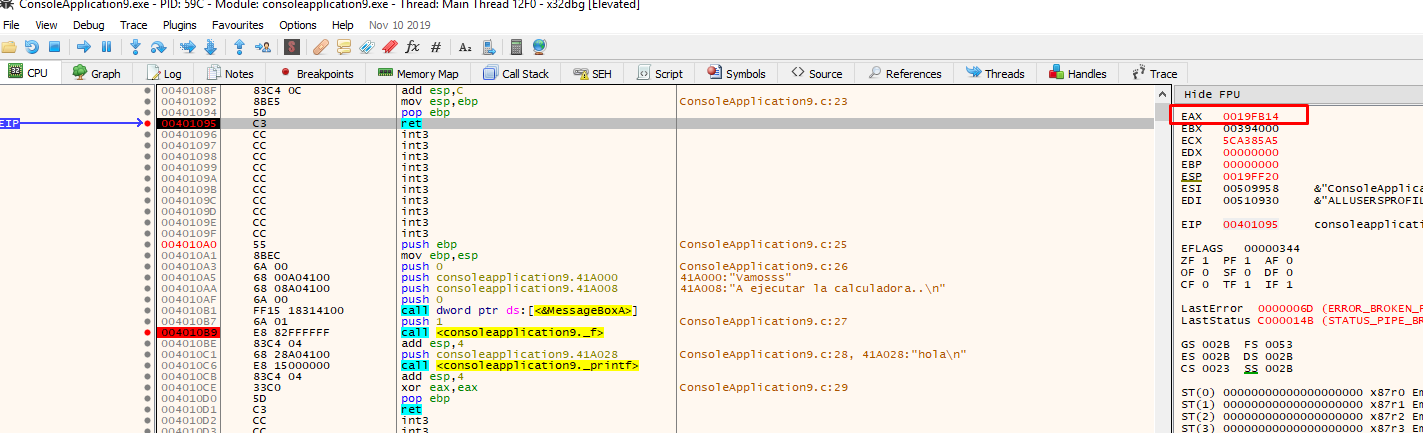
The only problem is putting the unprotected address first on the stack as we don't know its address and it may not be fixed.

Since I have to put the lpAddress argument on the stack, and I want that value to be an address from the same stack so I can unprotect from that address, I will look for a GADGET that writes values ​​in ESP or ESP + XXXX.



This gadget is very convenient since in the ESP content, we can write EAX. When we get to the RET before jumping into the GADGET, EAX is left with an address from the stack.

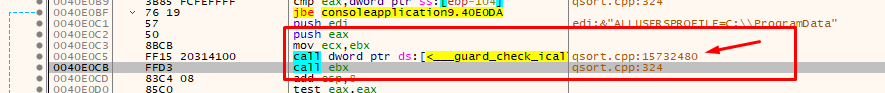
This could work, although it would be better if the gadget wrote in ESP + XXX a little lower on the stack.

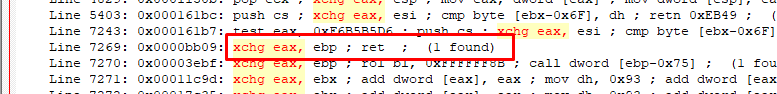


If we put that value as lpAddress and below the three arguments that we send, we would have the 4 armed arguments, we have to see if we can take advantage of this.

Since EAX is used to save the lpAddress argument, I cannot use it on other gadgets to resolve the call to VirtualAlloc.

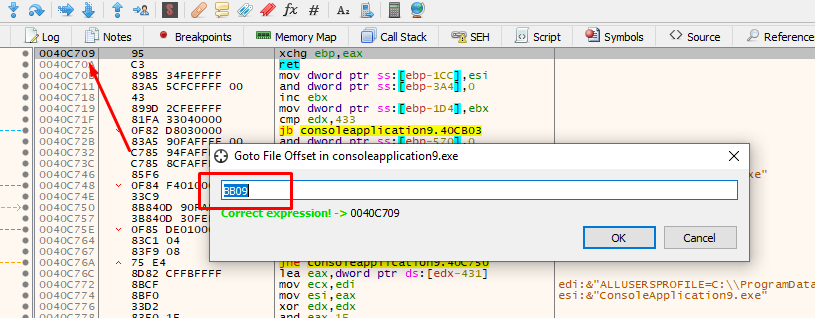
I can use EBX to call the function, if I can accommodate in EBX the value of VirtualAlloc and in EAX the value that it originally has from the stack, using this gadget that in the middle has a call that calls a ret, so it doesn't matter.





That will allow us to save the original value of EAX in EBP, build the remaining rop and then call the same GADGET again to restore it in EAX, and have it like at the beginning, great.

Let's build our ROP, the first thing then, will be to preserve the value of EAX, so the first GADGET to call is in FILE-OFFSET 0xbb09 whose virtual address is if we go to GOTO-FILE OFFSET in x64dbg.

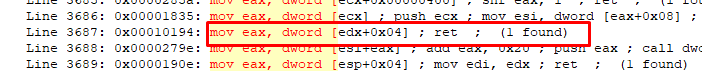


The address of our first gadget is 0x40c709.



Now we have EAX released to use it, since the value we were interested in is stored in EBP.

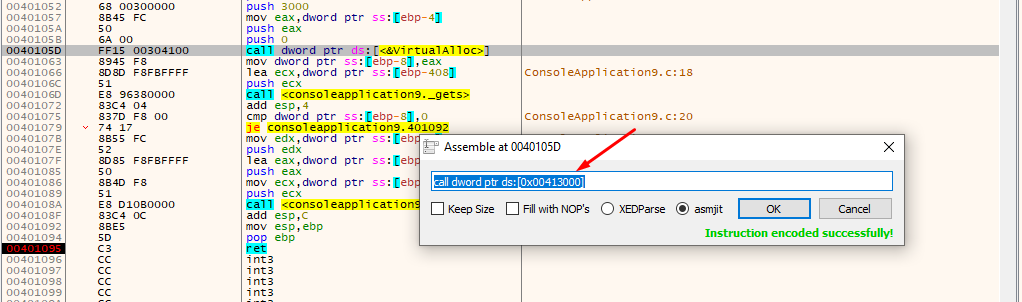
We are going to use this GADGET to move the address of VirtualAlloc



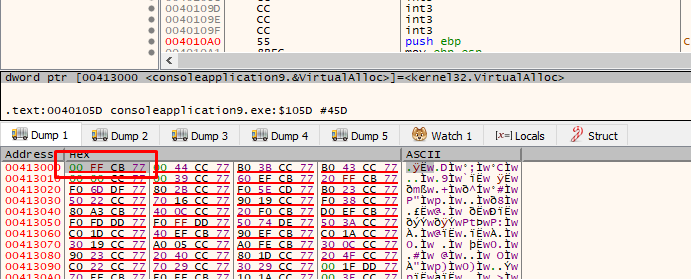
But first we must set EDX with the address in the IAT of VirtualAlloc, and subtract 4, so the gadget ends moving to EAX the address of the function.

To see what is the address of the IAT VirtualAlloc where the address of the function is stored, with looking at some call we will see it easily in x64dbg.

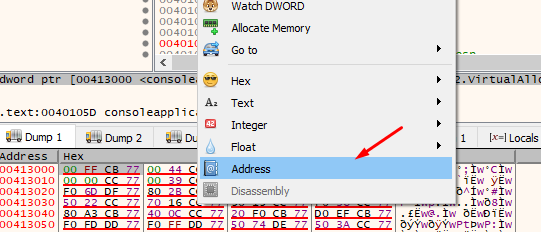
We see that when it jumps to VirtualAlloc it reads the address value of 0x413000.



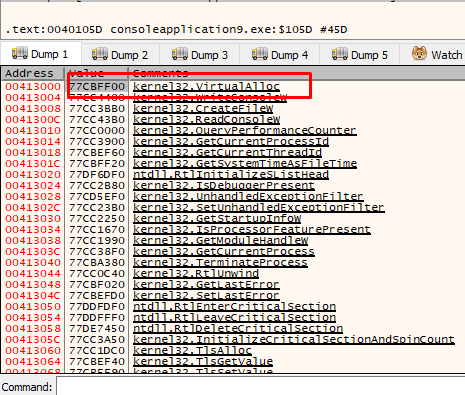
If we see it in the DUMP.



We can make it show the DUMP as addresses.

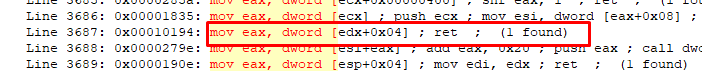


Now it will look better.



So 0x413000 on all machines, as this module does not have ASLR it will save the address of VirtualAlloc, which can change but the place where it saves it will not, so reading from 0x413000 we will have the address of the VirtualAlloc function for my process, for any machine.

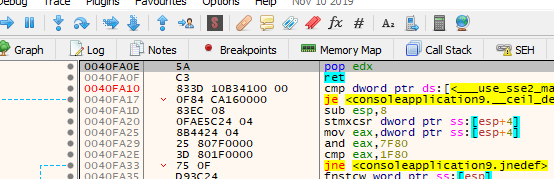
So since we will use EDX we will have to set EDX to 0x413000-4 first because it adds 4 inside the GADGET to compensate, and it ends reading from 0x413000 and move the address of the VirtualAlloc function to EAX.

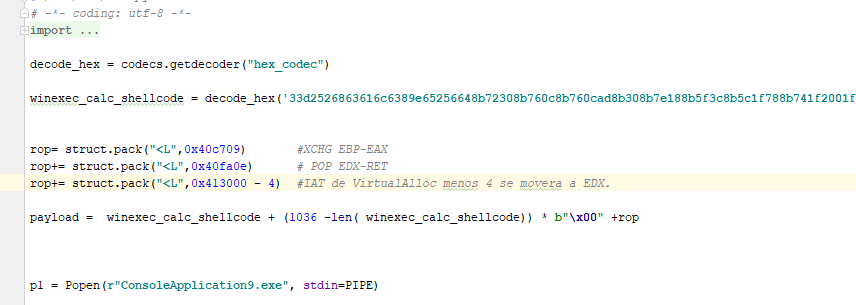


So our next gadget will be



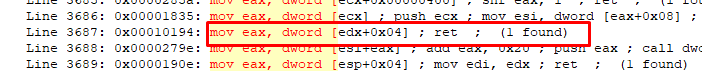
Whose virtual address is 0x40fa0e.



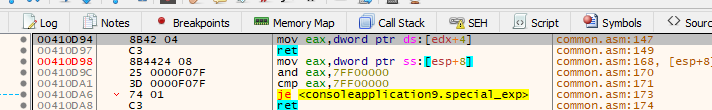


So we will move the value of the VirtualAlloc IAT entry minus 4 to EDX.

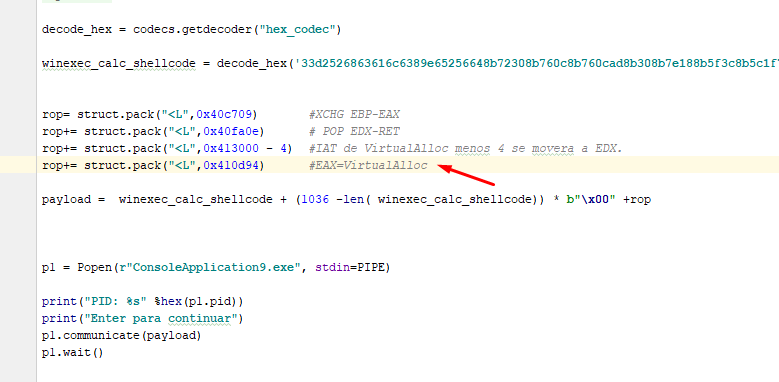
Then comes the gadget that moves the address of VirtualAlloc to EAX.



Its virtual address is 0x410d94

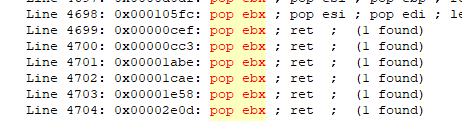


We have 0xD in the address but we already saw that it is not an invalid character, so we continue.



With this gadget, we can pass the address of VirtualAlloc to EBX if EBX is 0.



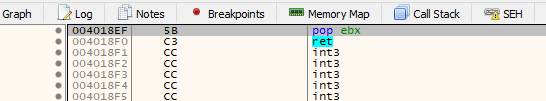


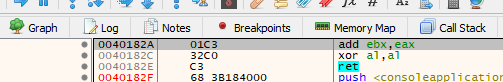
And with this GADGET we put zero in EBX.

One important rule is that we should always try to avoid GADGETS with LEAVE since the stack may break, although RP ++ doesn't show them to us, but if we search by hand, we shouldn't use them, unless we can accommodate EBP so nothing breaks.

So we can add these two gadgets, and we already know how to find their virtual addresses.

**0x4018ef** = POP EBX-RET and **0x40182a**= ADD EBX, EAX -xxx -RET

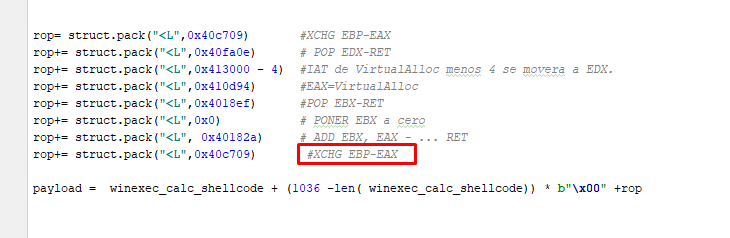




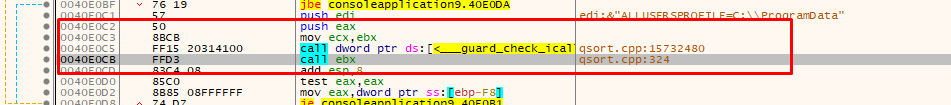


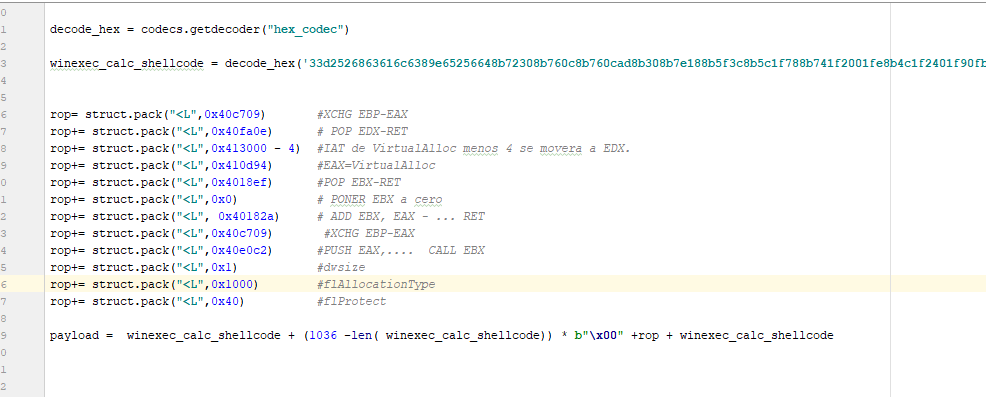
With that we would already have the address of VirtualAlloc on EBX.

We must return to EAX the value it had originally with another XCHG EAX, EBX.



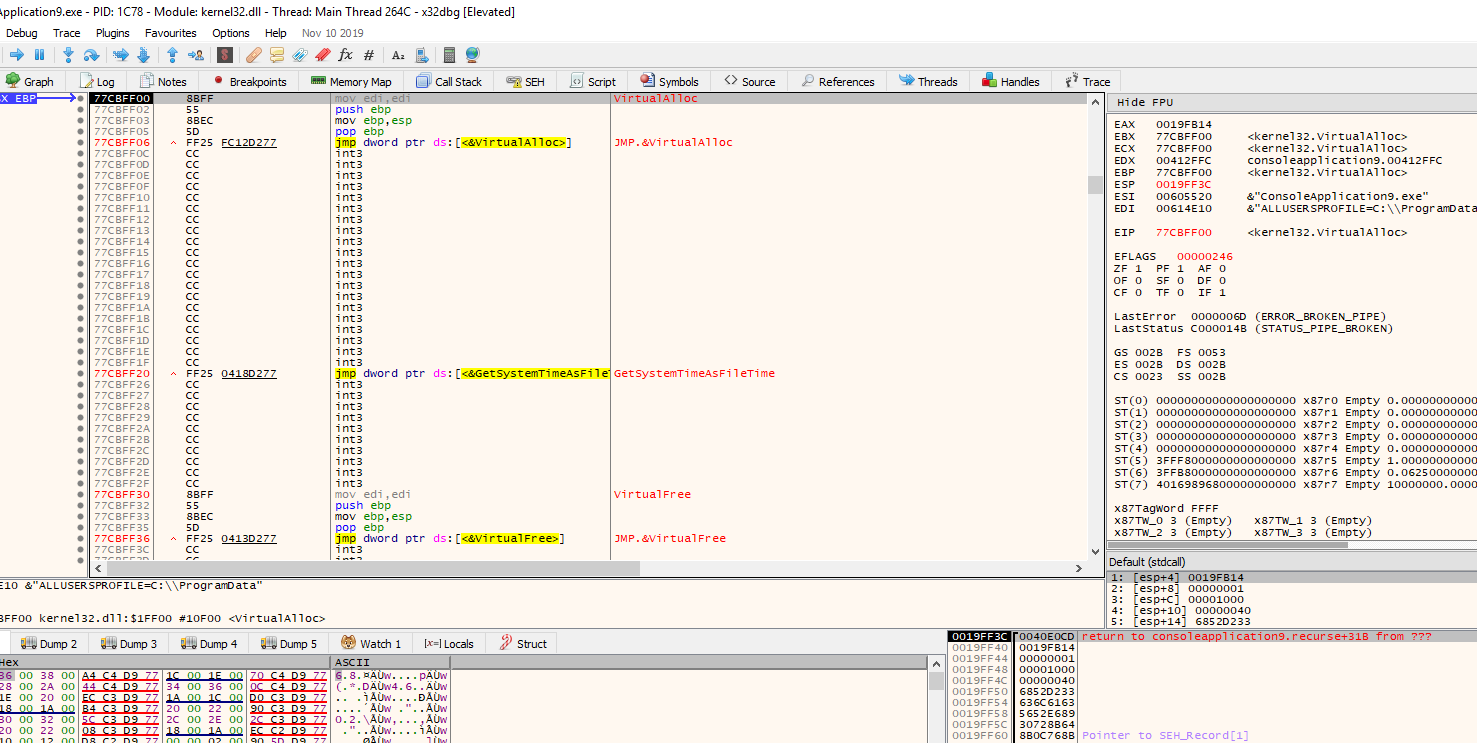
And then would come the call to function.





We see that if I run the script like this there is still a problem.

It reaches the RET and I trace with F7 until I reach VirtualAlloc.



There are the correct arguments

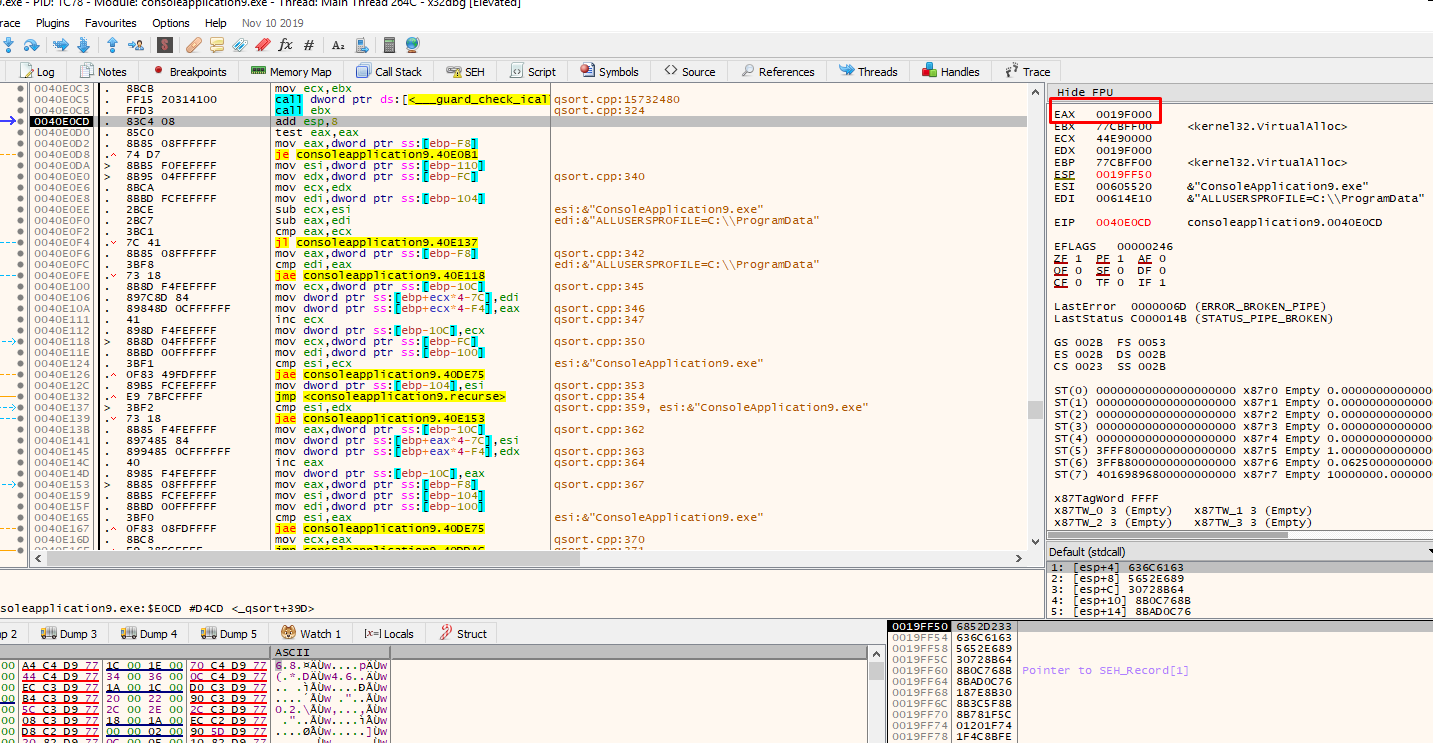
0019FF40 0019FB14 lpAddress

0019FF44 00000001 dwsize

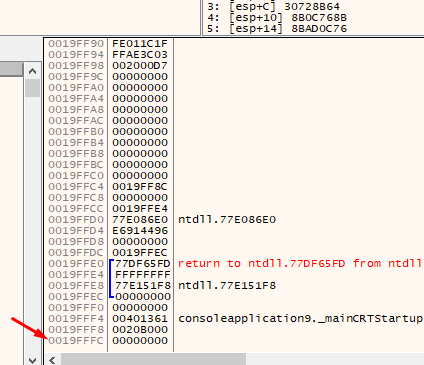
0019FF48 00001000 flAllocationType

0019FF4C 00000040 flProtect

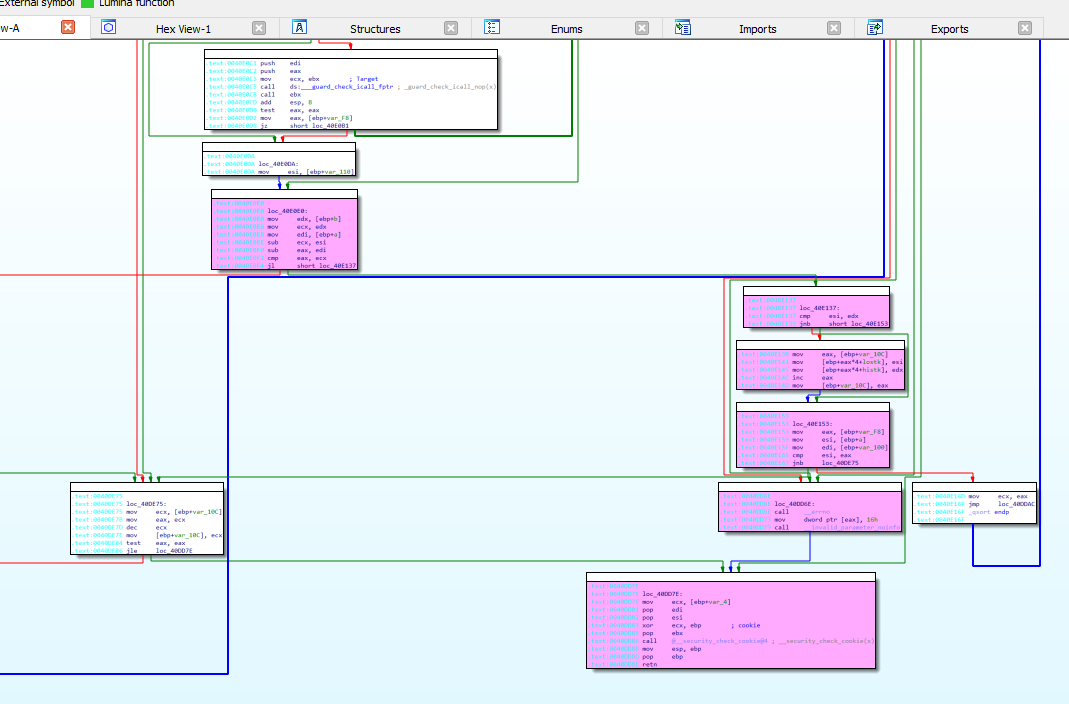
If we return from the VirtualAlloc function, we see that the return value is correct.



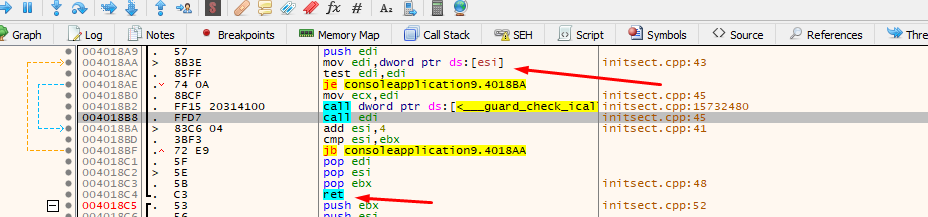
The function returns 0x19f000 that is the beginning of the page of the unprotected section, from 0x19f000 to 0x19f100.



We already have an executable stack but there is still a problem, can we return from this function without crashing the program to execute the shellcode?



We see in IDA how to get to the RET, and we see in the addresses light blue letters, which indicate that it is an embedded dlls code, so it adds the security cookie before the RET, which prevents us from reaching it. We must change the strategy.

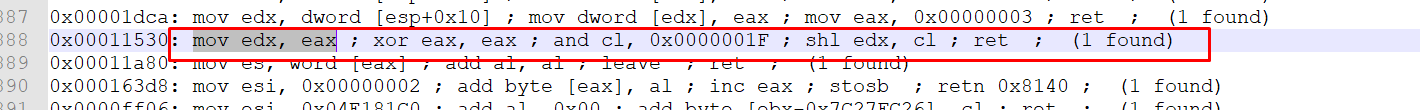


We see another GADGET to jump to VirtualAlloc.

EDI must have when jumping the address of VirtualAlloc and since EDI will have the ESI content, it solves us easily by putting the address of the IAT entry 0x413000 before in ESI.

In EDI at the beginning of this GADGET, the address of the stack that EAX originally had should be to be pushed, so it will have to be moved from EAX to EDI.

Here we see two gadgets, the first one if ECX is zero moves EAX to EDX (0x412130)

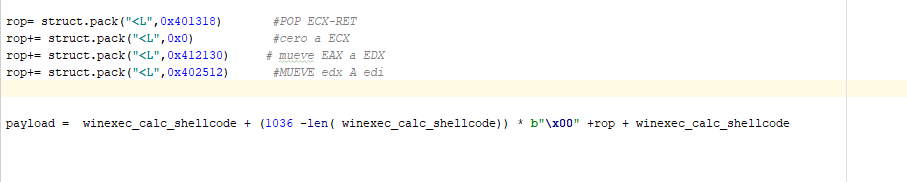


and we move EDX to EDI (0x402512)



With those two, if ECX is 0 at the beginning we already have EDI with the stack value, let's build.

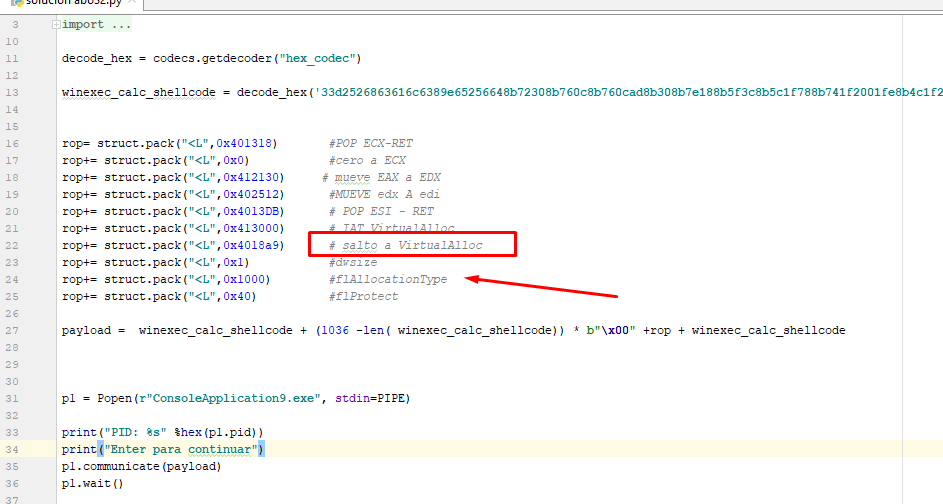
With this we set ECX to zero (0x401318)



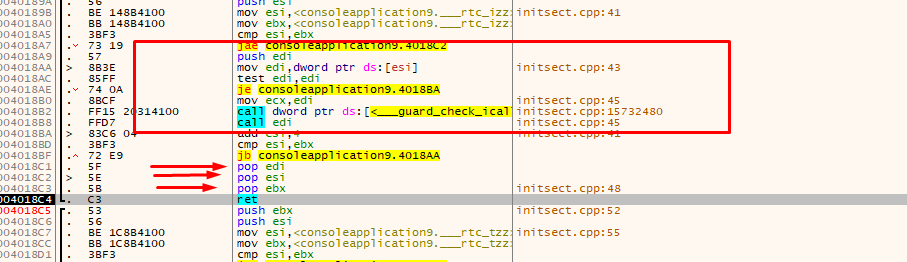
With that gadgets, we already have in EDI the value of the stack that it is going to push, we only need to put in ESI the address of the IAT of VirtualAlloc, it is easy. (0x4013DB)



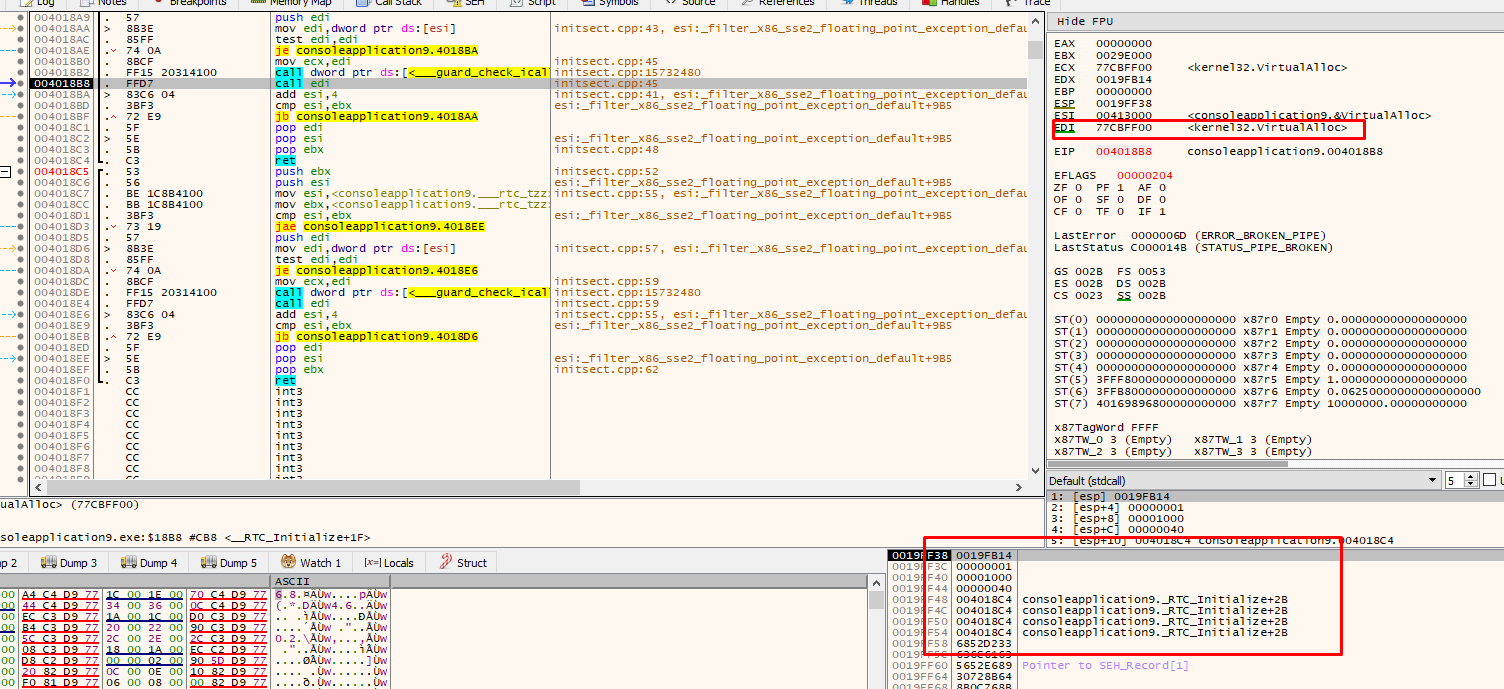
And we jump to execute the function, then we put the missing arguments underneath.



And I have to accommodate the return, since when returning from the call the program makes three POPs.

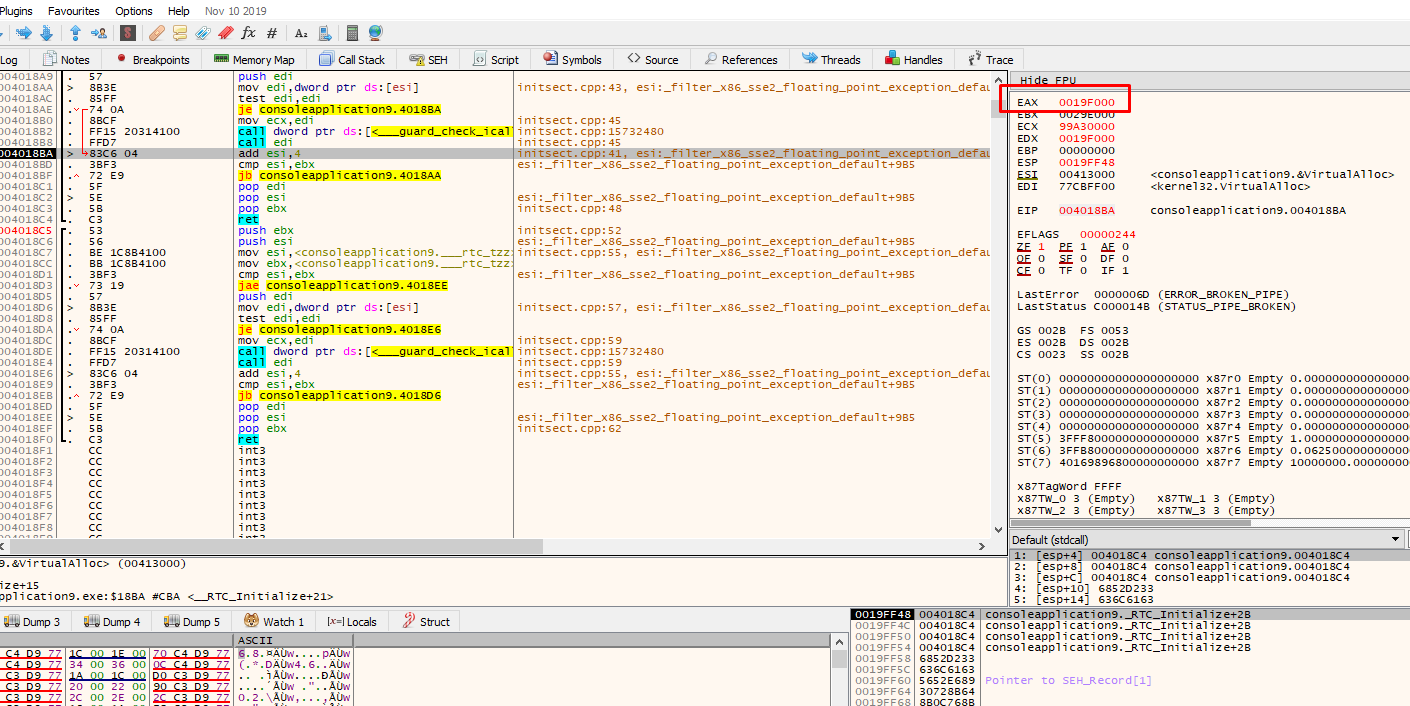


So I will put three RET pointers that are similar to NOP, while we are ROPING they do nothing just padding, in the image above I can see a RET at 0x4018c4.

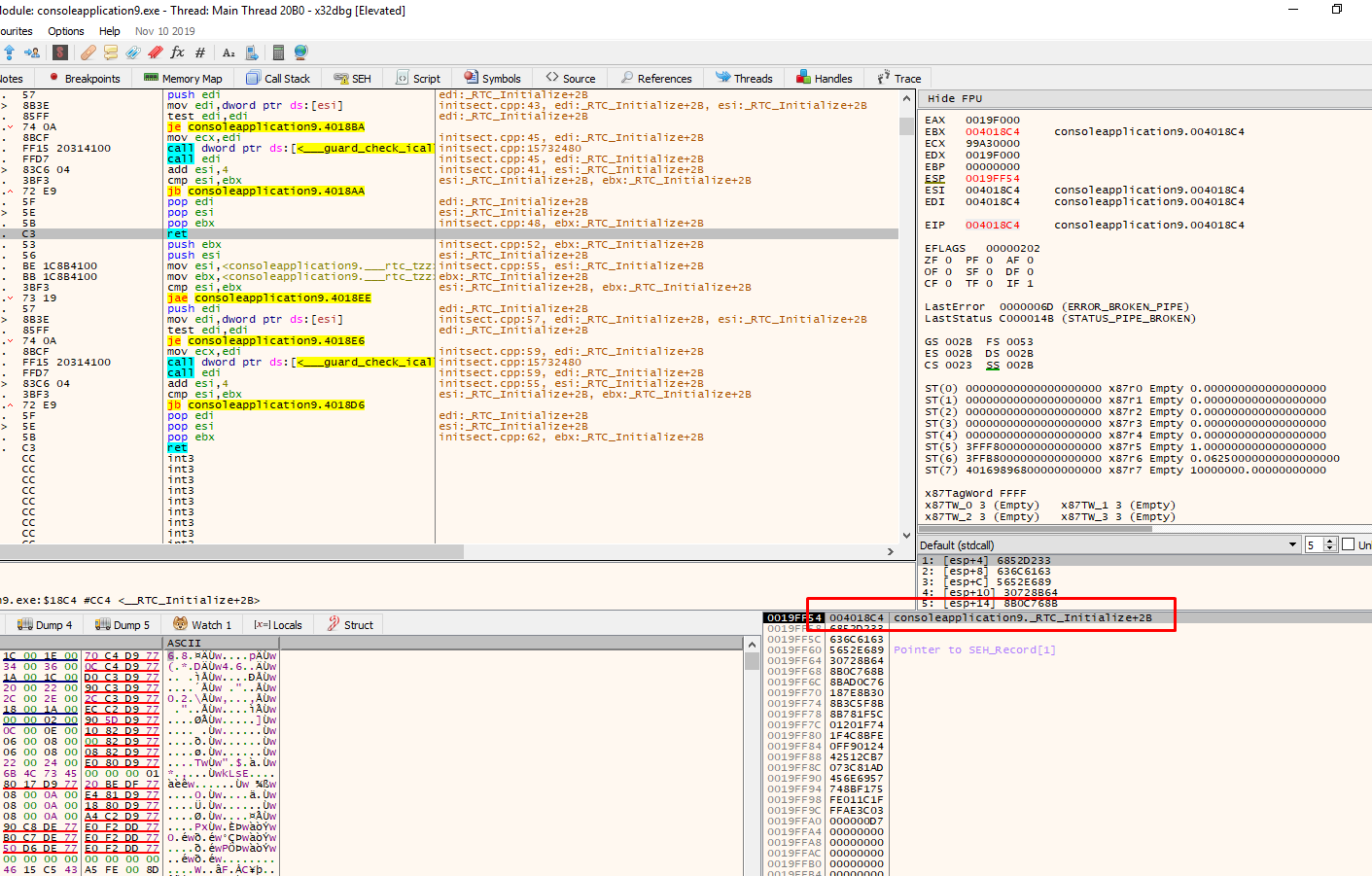


Now we reach the function with all the correct arguments and the possibility of handling the return.

We see that like the previous time it returned the address of the page to which we gave permission to execute.

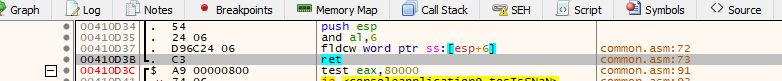


If I continue I will see that the pops are fine I only need one last gadget to jump to execute the stack where is my shellcode.

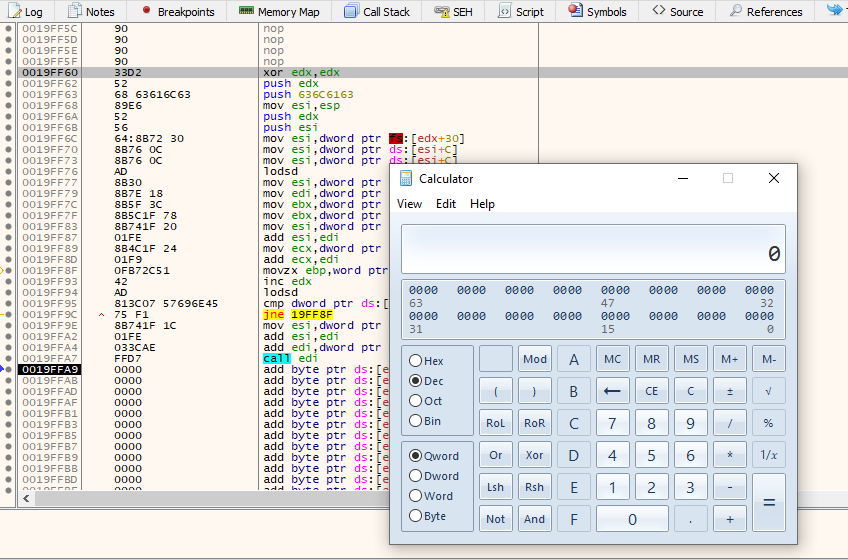


Right after the last RET we lack a CALL ESP or something similar to jump to execute the stack.

We will use a PUSH ESP-RET with a little garbage that does nothing in between.



With this we already jump to our shellcode and we can run without problems.



We have defeated DEP, in a simple case, hehe.

I did not want to eliminate the part we did and it did not end up working, because everything is practical, even the errors, and with that we also learn from them and how to repair them, as we do when we work on a day-by-day basis.

We will continue in part 12 with the following exercise, which is compiled in 64 bits and we will have to build the ROP as we did in this one.

Then later we will see more complex cases of ROPs and we will advance slowly.

Until part 12

Ricardo Narvaja

Forced to stay in a house due to the coronavirus pandemy,

Luckily we are healthy for now.

03/15/2020