# EXPLOITING AND REVERSING USING FREE TOOLS (PART 15)

As we saw in part 14, this rop will not be as simple as the ones we have seen before.

We will continue using radare in visual mode to practice.

# REVERSING EXERCISE 2–32 BITS WITH RADARE

OPEN STORED PROJECT (PO)

I open my stored project, the explanation of each command is in part 14.

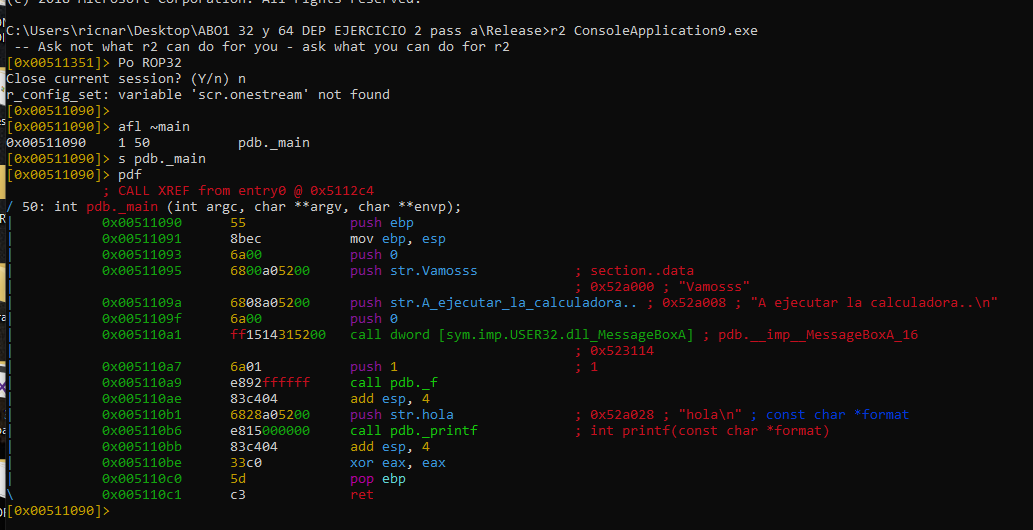
**r2 ConsoleApplication9.exe**

**Po ROP32 afl**

**~ main**

**s pdb.\_main**

**pdf**

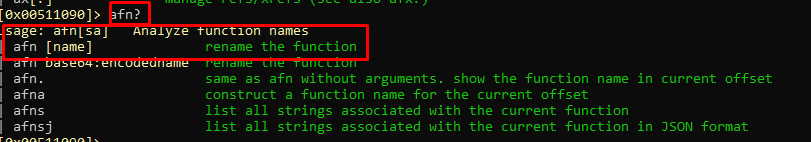


RENAMING FUNCTIONS (AFN)

I’m already in the **main**, I will rename **pdb.\_main** and **pdb.\_f** to **main** and **f** respectively, using **AFN**.

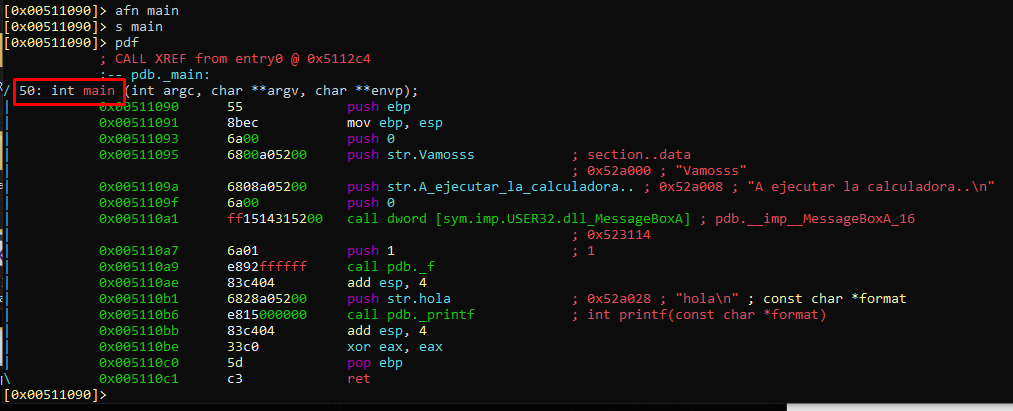
QUESTIONS ABOUT A COMMAND(?)

If I have any questions about the use of a command, writing **"?"** after it, I will have the help.

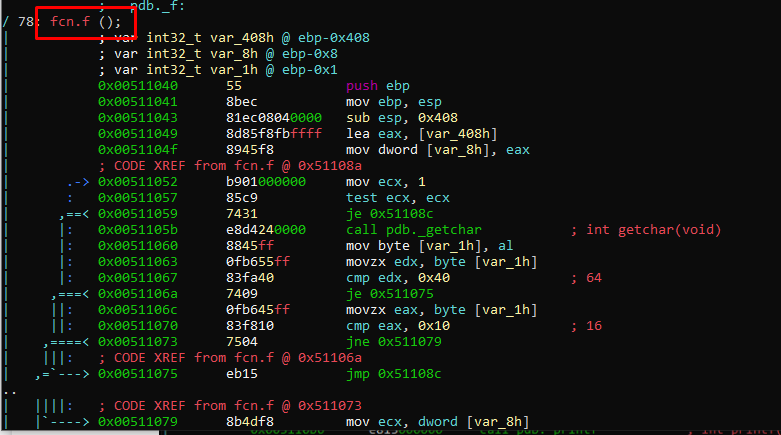


With the **afn** command, I will change the name of the function in which I am located.

**afn main**



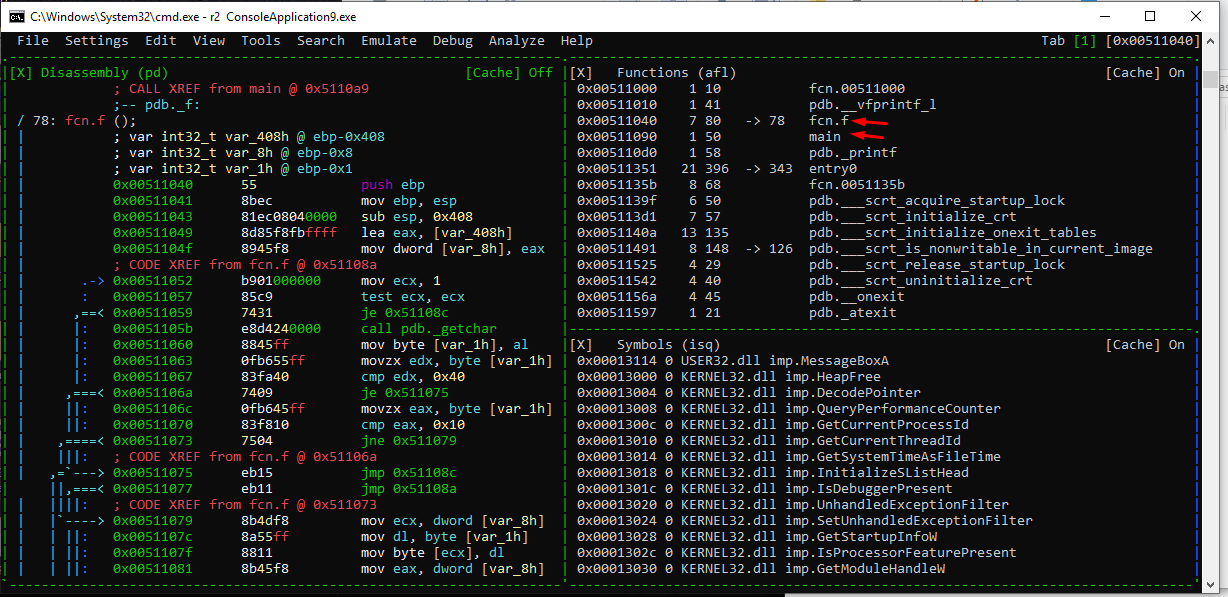
Then I will go to the function **pdb.\_f** with the command **s pdb.\_f**,and rename it with **afn f.**

****

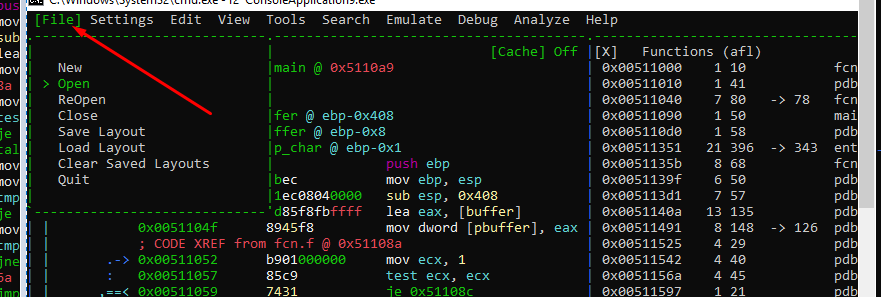
I see that radare forces me to use it as **fcn.f** at least.

VISUAL MODE (v)

I enter visual mode with **v**.

****

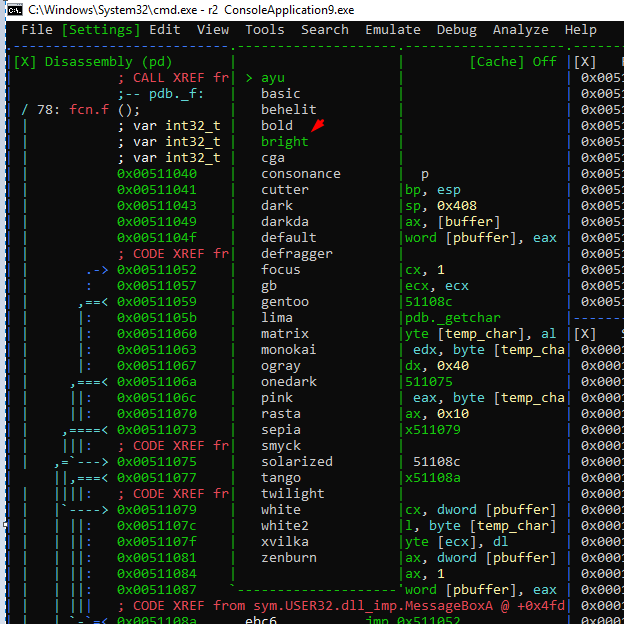
I see the functions that I have renamed, I save the project by pressing the key **“:”** and then **Ps ROP32**

****

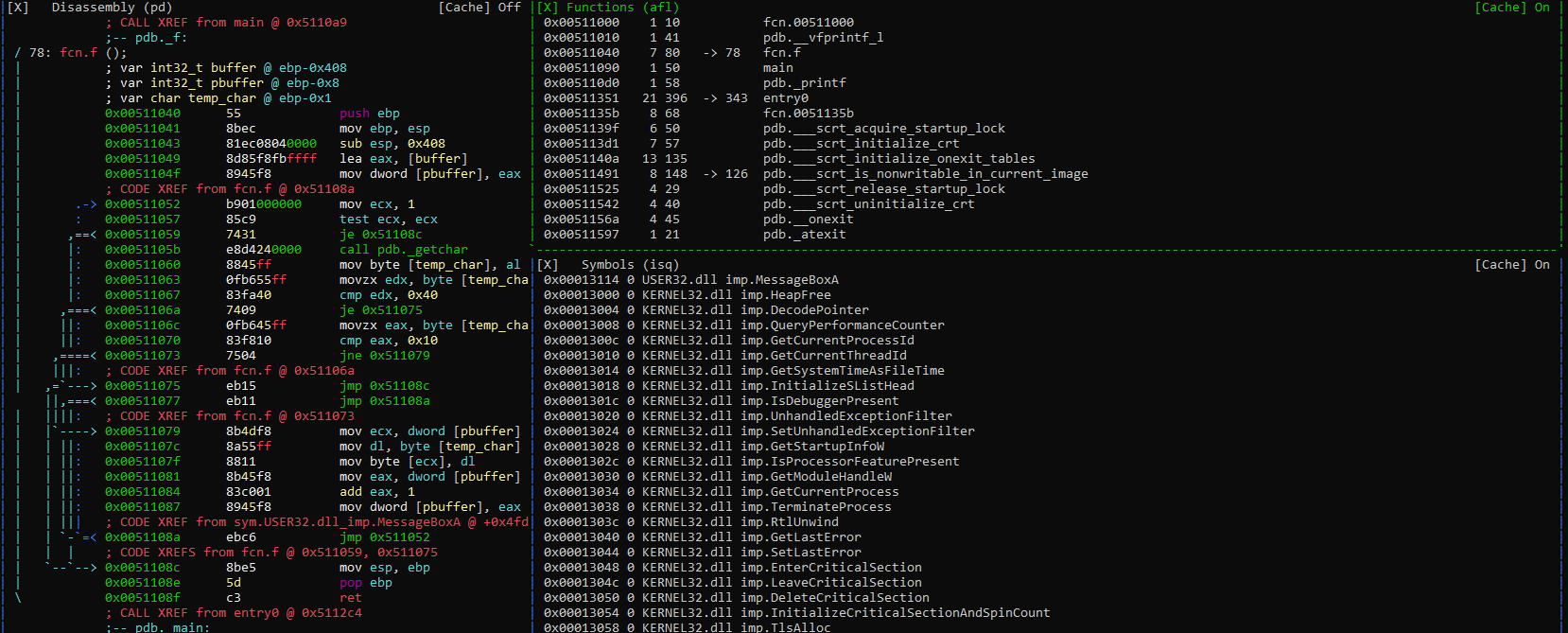
NAVIGATING MENUS (M)

I see that with the **M** key and the direction arrows, I can navigate the menus.

In the **Colors** menu, the one I chose is checked, but there are many more options. (not all color options work)



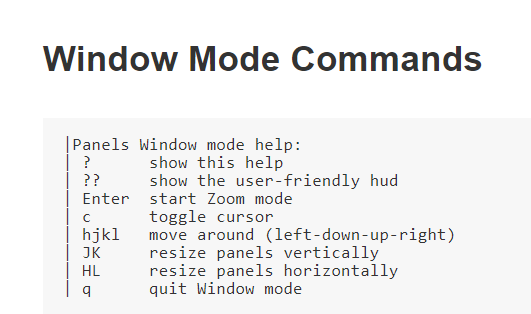
I can configure in VISUAL MODE, which panels I want to see, and what size they will have.

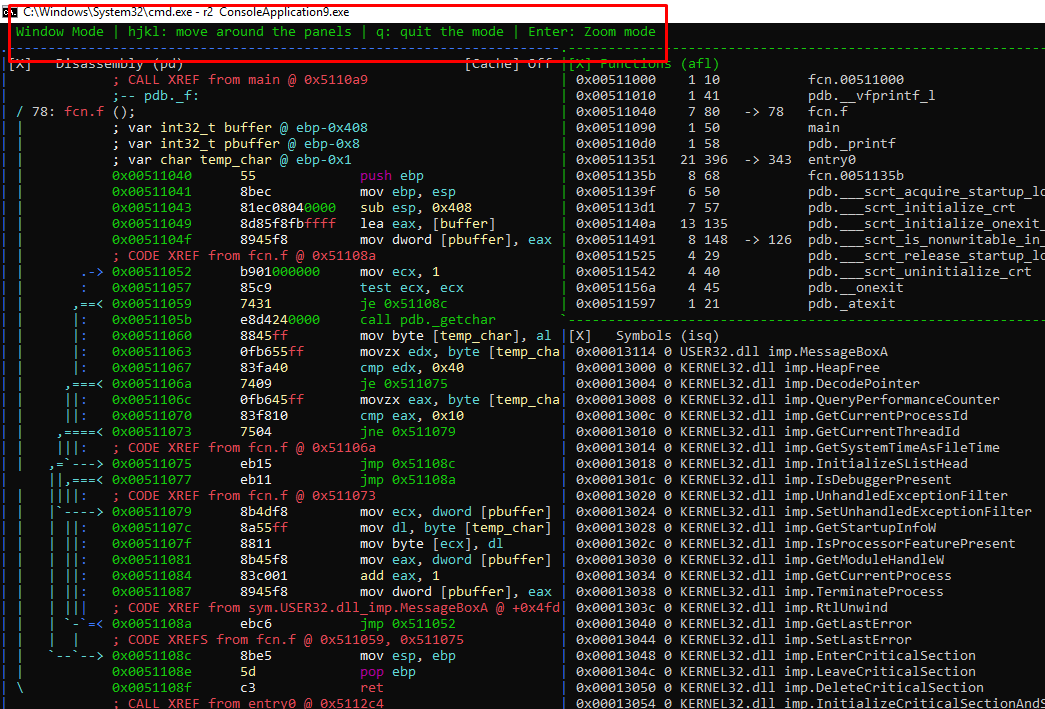


I see in my case, that the functions panel and the symbols panel are too big, I can enlarge the ones I want, eliminate the ones that are not used, and save the LAYOUT.

WINDOW MODE(W)

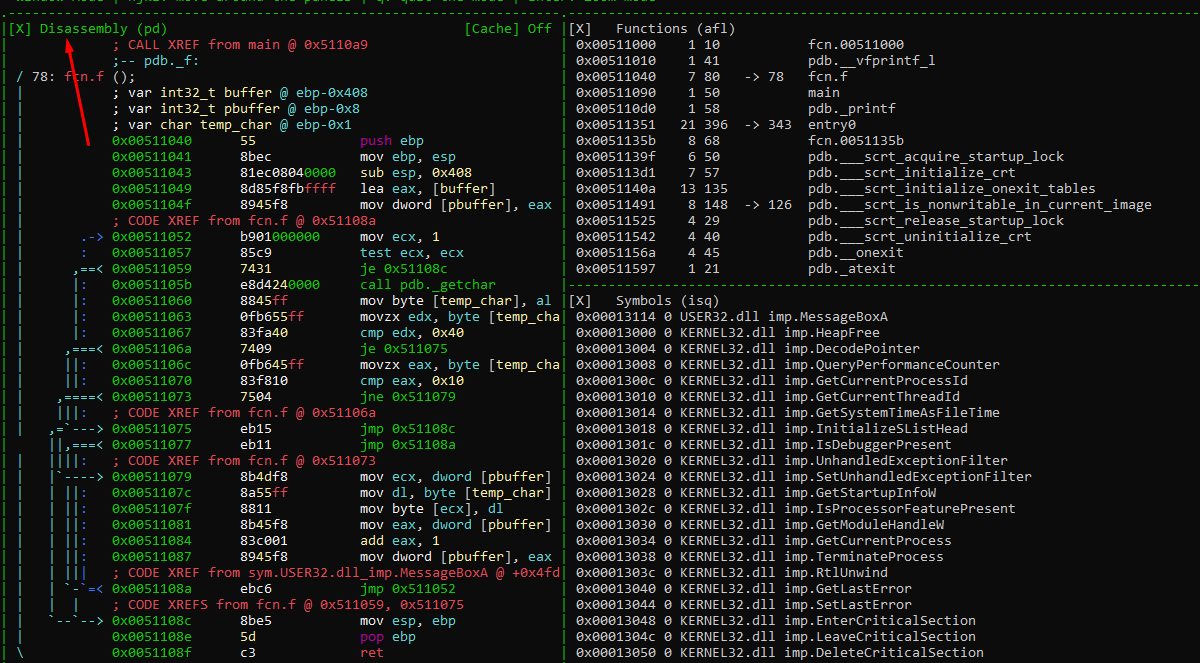
To expand the disassembly panel, for example, I enter in the window mode with the **w** key.



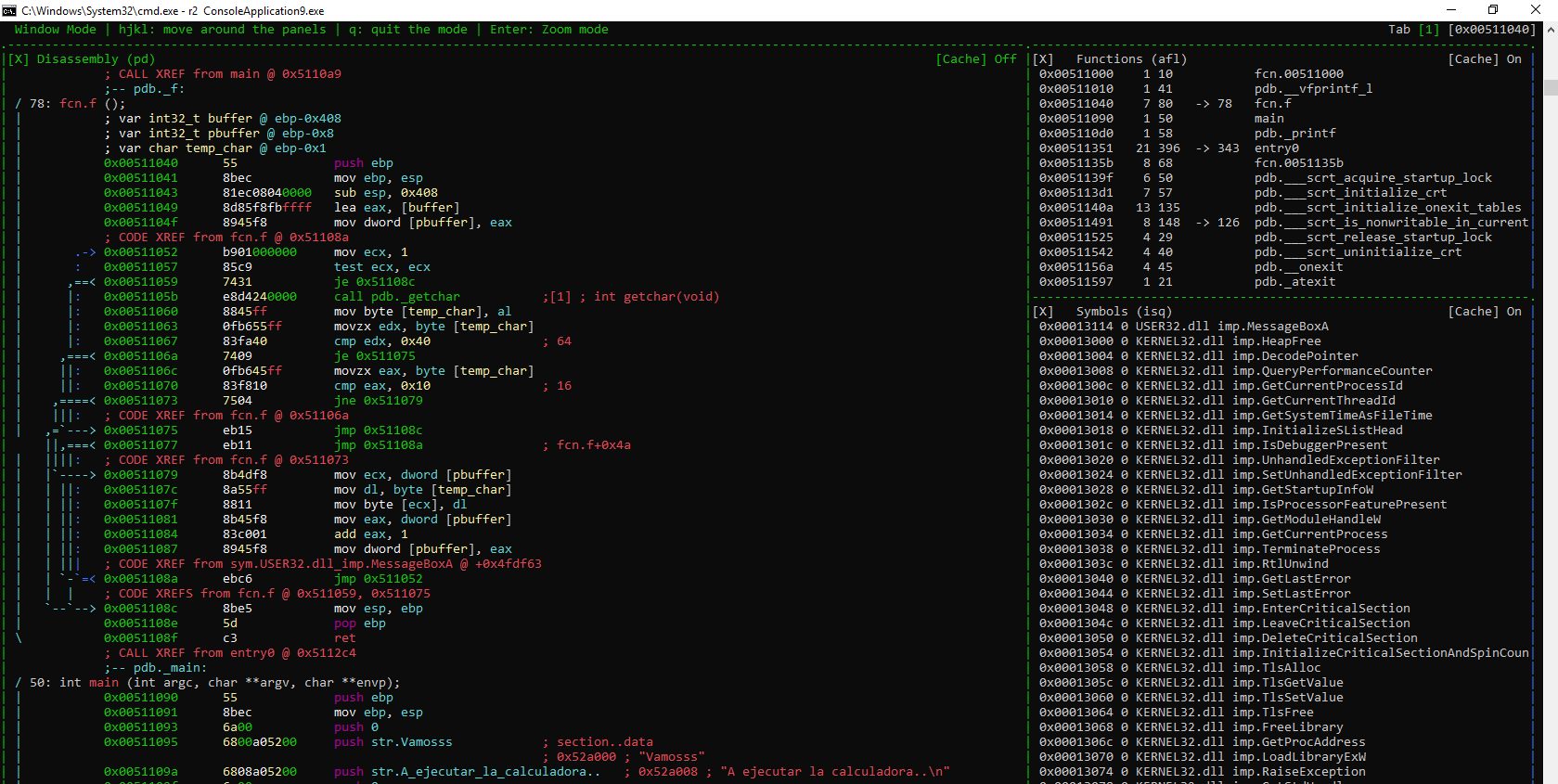


I can see at the top, that I am in WINDOW MODE.

With the arrows I move between panels, to highlight the panel that I want to be active.

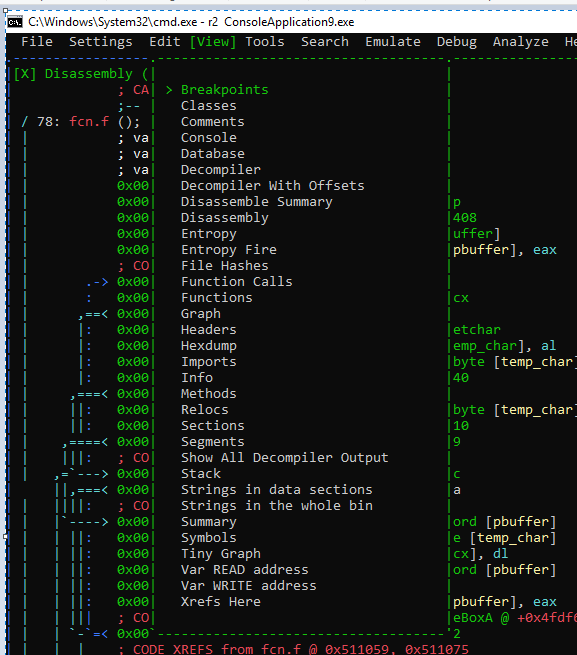


Then, with the **SHIFT + H, J, K, or L** key combination, I can change its size. In my case, I want to enlarge the disassemble panel to the right, so I will do it with SHIFT + L repeatedly.

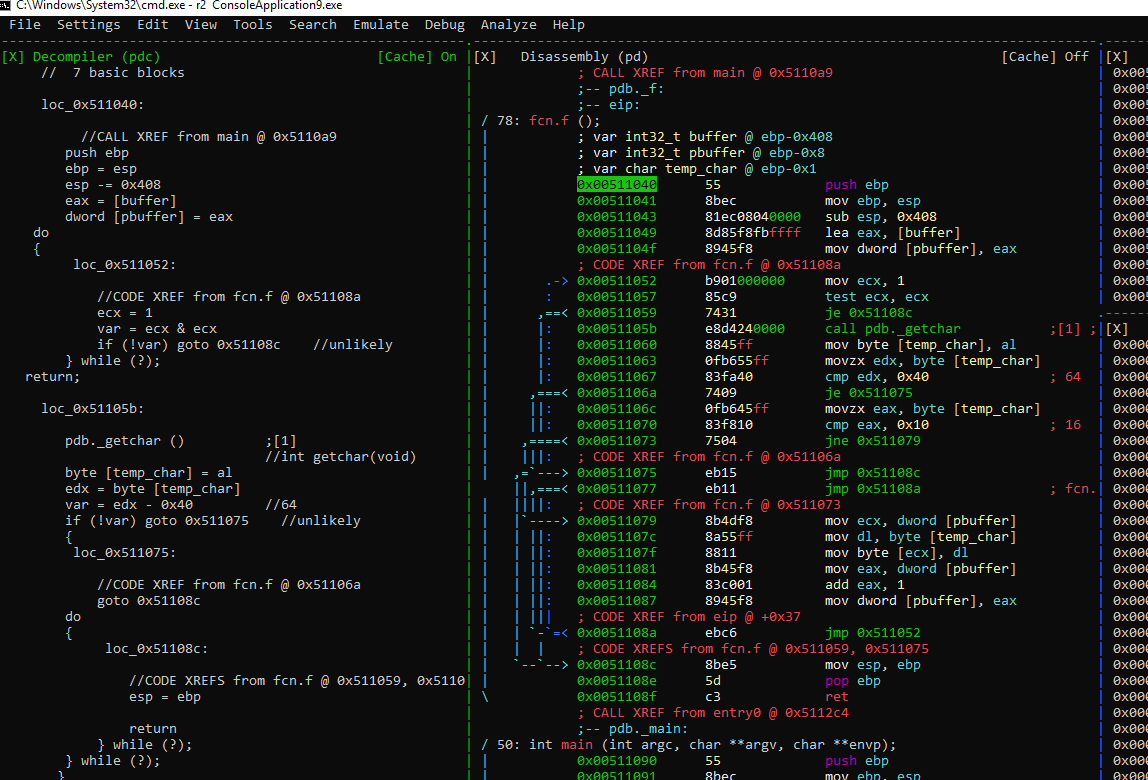


So it is more visible, I can also see what more windows I can add.

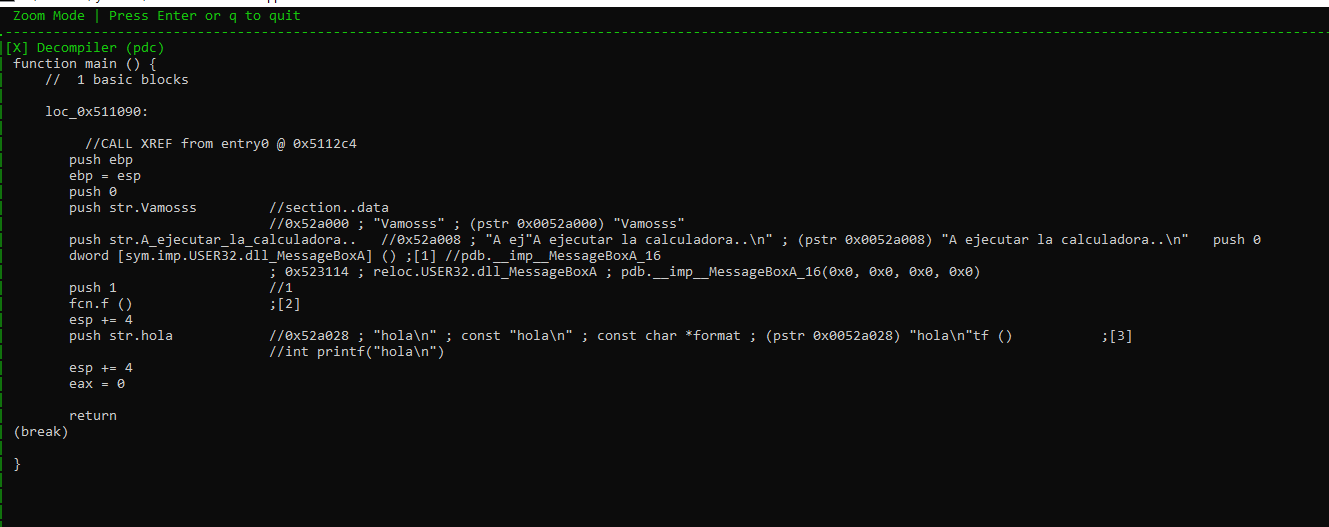
I exit from WINDOW MODE with **w**, and enter the menu with **M** and go to **VIEW**.



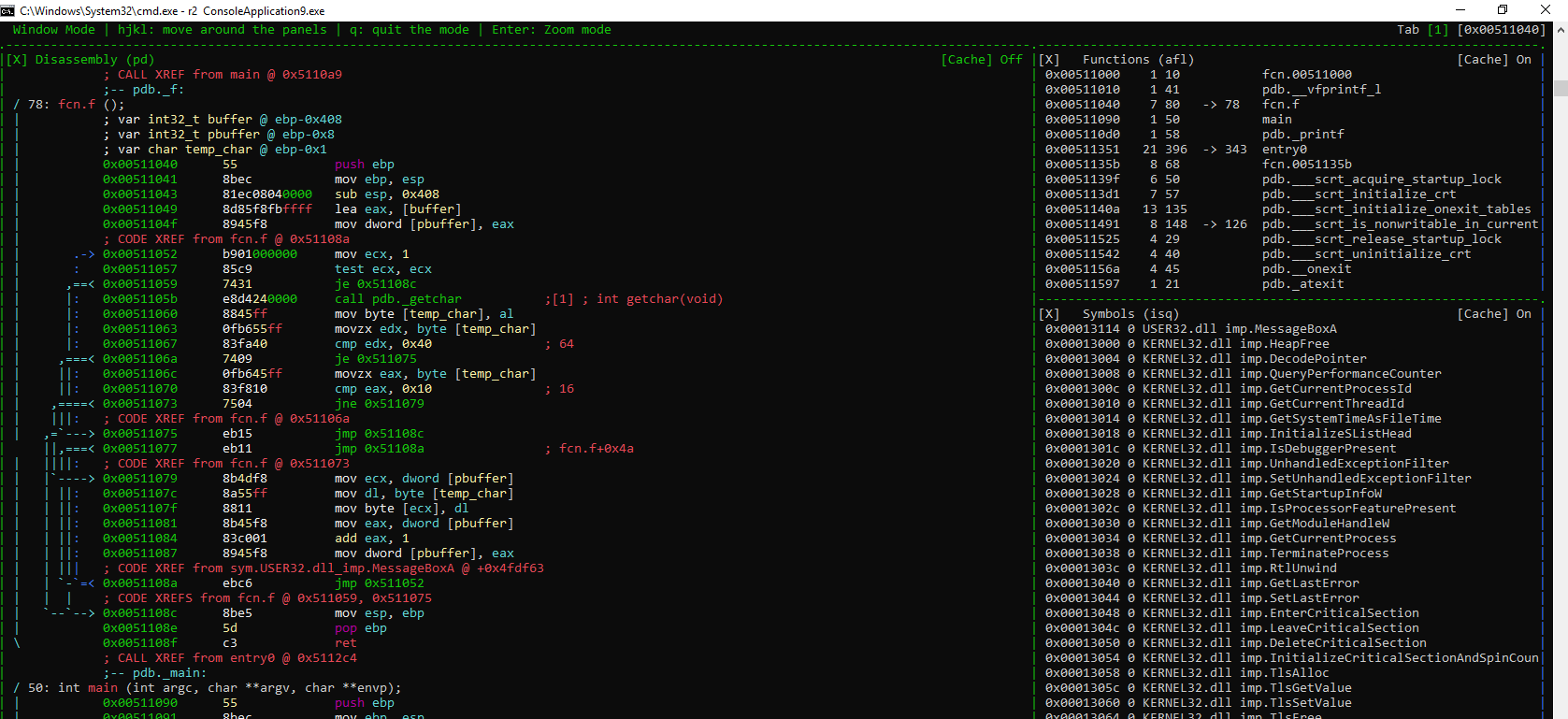
I can add a decompile window.



I can focus on that panel, and I can enter in ZOOM mode pressing the ENTER key.

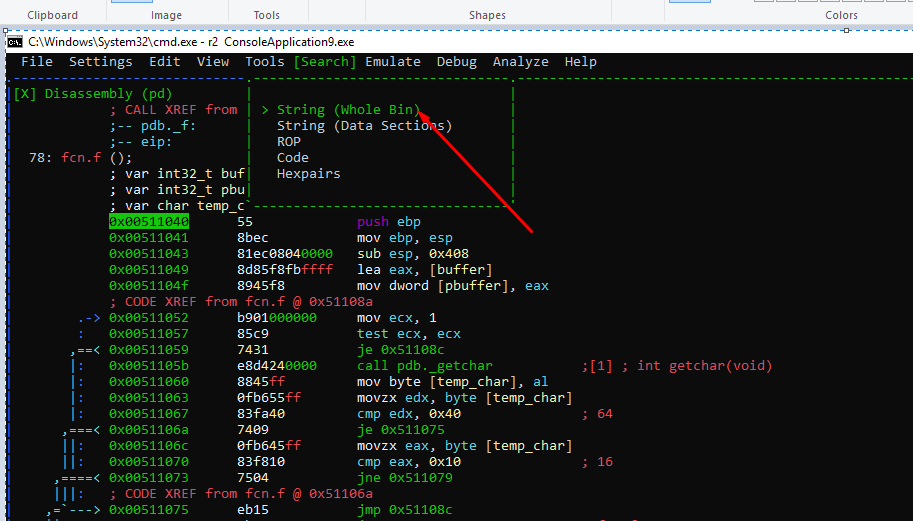


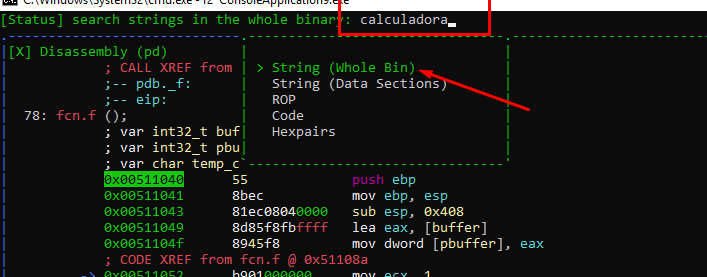
If I want to delete a panel, I go into WINDOW MODE with **w,** highlight it and press **SHIFT + X**.



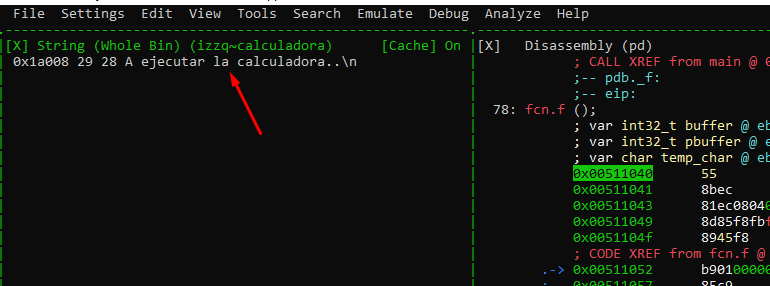
SEARCHING

If I return to the menu, I see in it, the option of SEARCH - STRINGS.





Let's see what it finds.

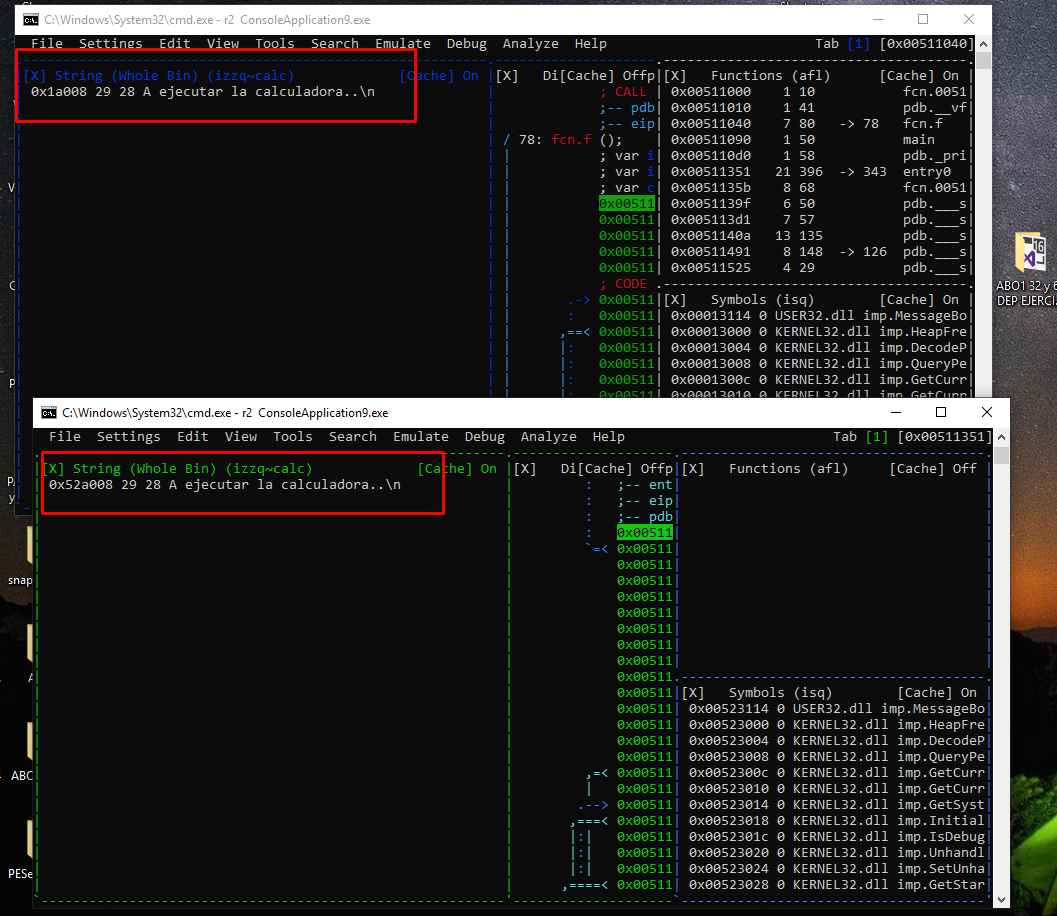


Here I found a bug in radare. When I reopen a project, it doesn't look exactly the same as if I ran it the first time. I see the address shown to which I must add the IMAGE BASE to get to the string address.

Python> hex (0x510000 + 0x1a008)

'0x52a008'

Whereas if I open it on another console without loading the project, I load the symbols, parse them, and then go to the same menu to search for the strings, the result is different.



There I see the comparison of both.

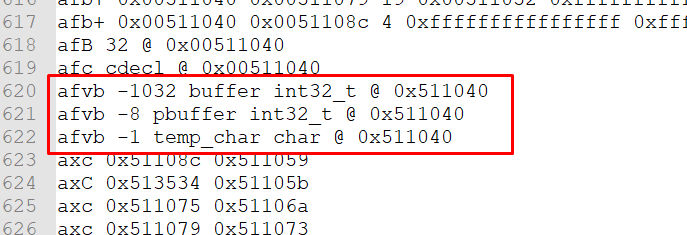
The result is not the same and the references will fail me, and maybe some other things.

I open the executable again on another console.

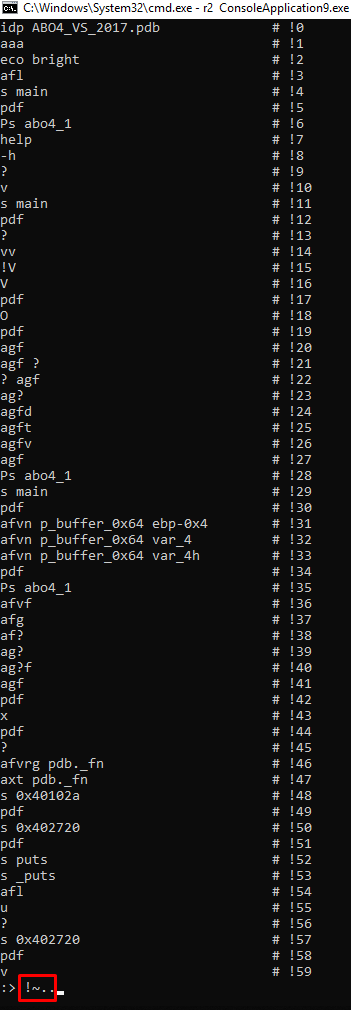
I can copy and paste all the commands that I had used all together and they will run at once

I can see the commands used in the project file with the ".rc" extension.

C : \ Users \**XXXXX**\ .local \ share \ radare2 \ projects \\ ROP32



There is also a command to view the history of the commands I have run. (**“!”**)



I copy the important commands to a text file, and copy-paste them all together on the console.

If I put them in the same order that I run them at the time, it will work without asking anything.

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

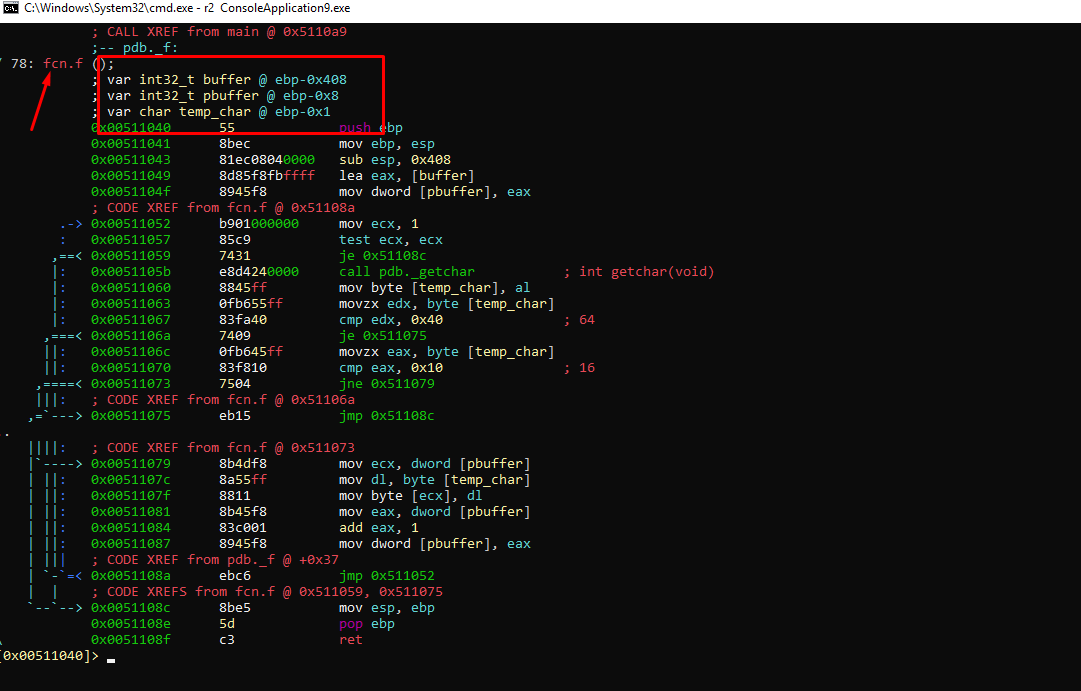
afn main

s pdb.\_f

afn f

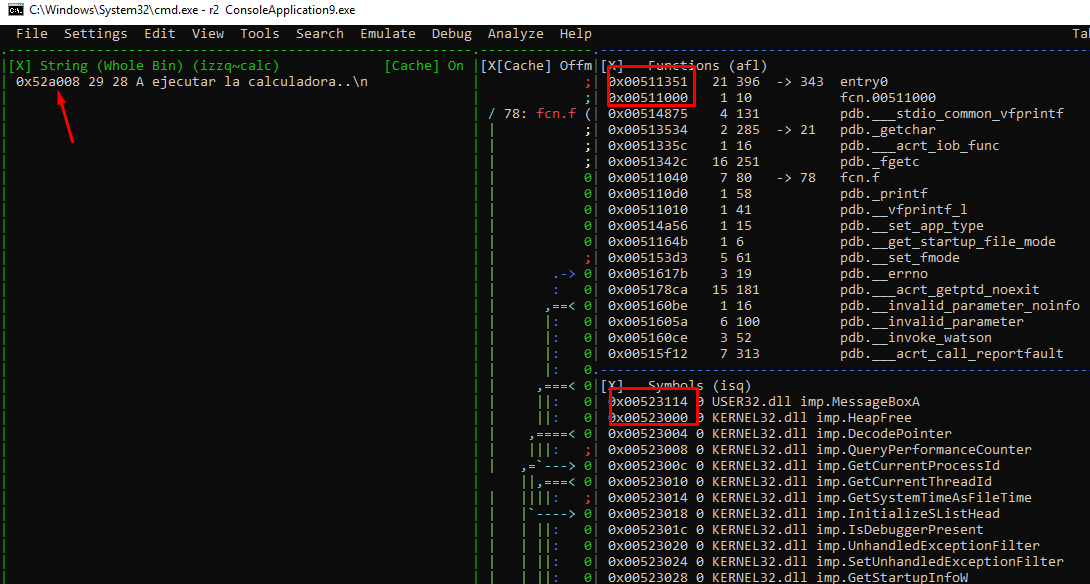
eco bright

pdf



I see that I am back to the state where I was previously, now I can continue.

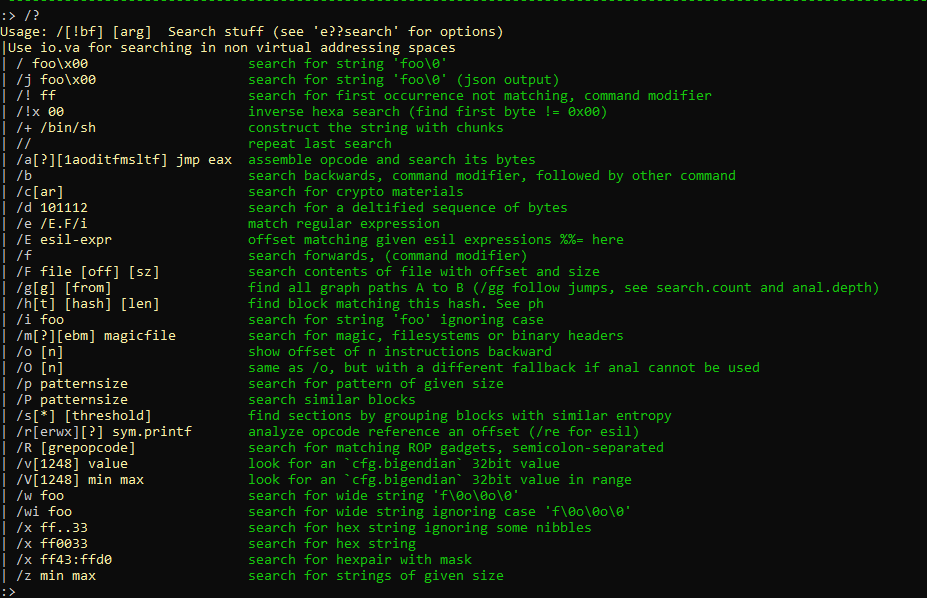
I enter to the visual mode with **v**.



Now everything coincides.

In the menu I can also search for ROP GADGETS, HEX CHAINS, etc.

Searching from the console gives me many different possibilities.

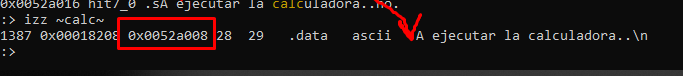


I see all kinds of searches. To search for string references, I can use the **“/”** command, if I type "**/ calc**" I will find them one result.

****

I see that it shows the address where the word **"calc"** is found, which is in the middle of the entire chain.

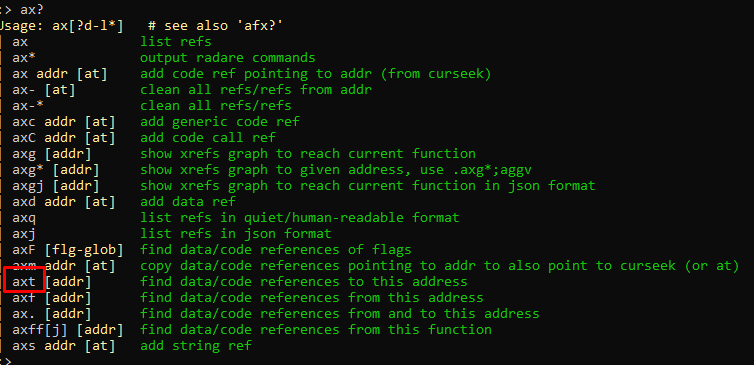
The problem is that there will be no reference in the program to the string calc, there will only be references to the starting address of the entire string. So I will use the **"izz"** command.

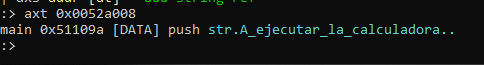


I see that although I have searched for the **calc** word, the entire string is found and returns the initial address of it (0x52a008), from where I can easily find the references in the code with the **"axt"** command.

FINDING REFERENCES

There I see all the possibilities of finding the references that I have.



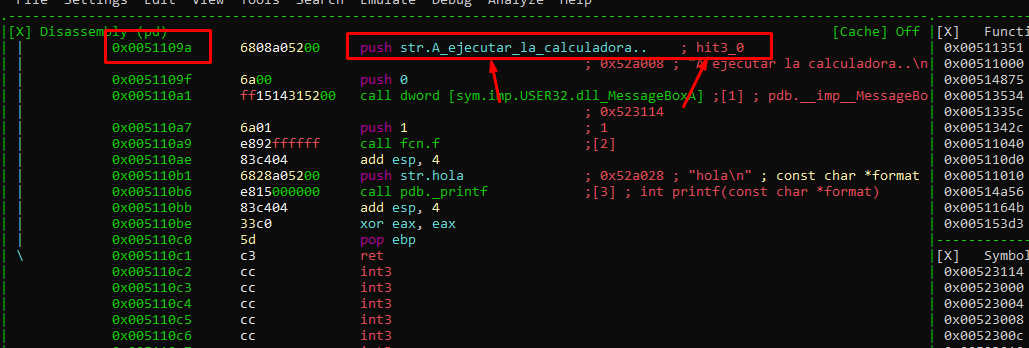


There is the address 0x51109a where the program uses the string.

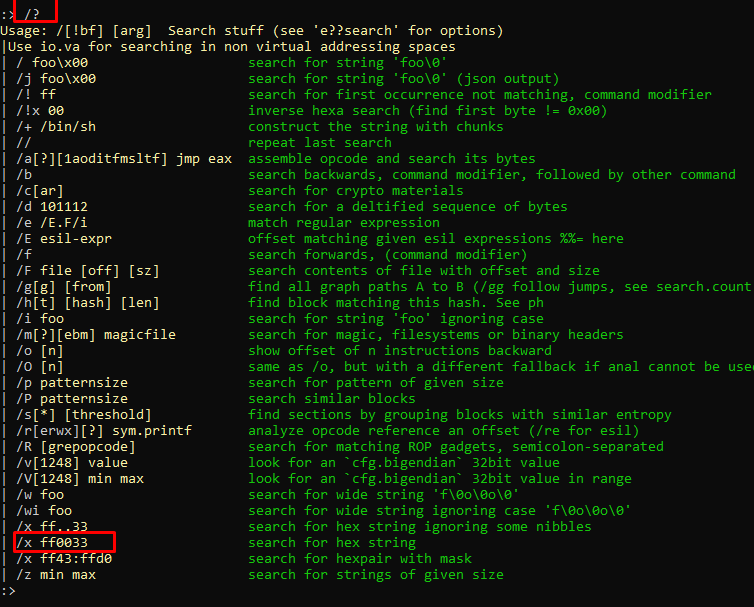
I can see it on the disassembly panel in visual mode with “**v”**, highlight this panel, hit the **"g"** key, and type the address **0x51109a**.

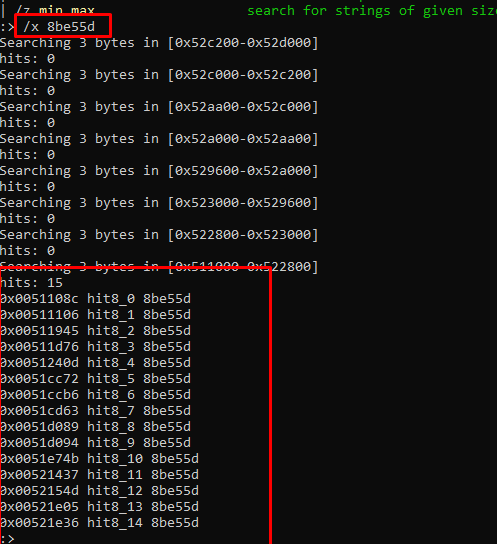


And it shows me the string in the list.

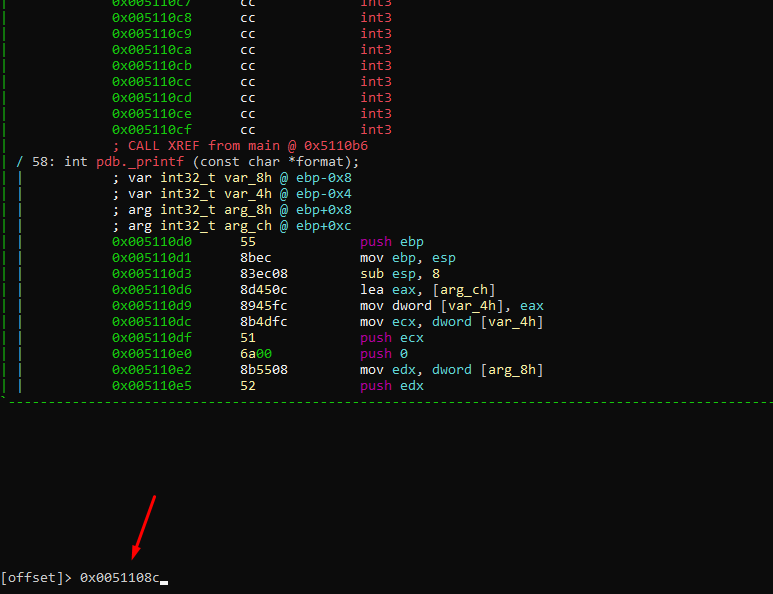


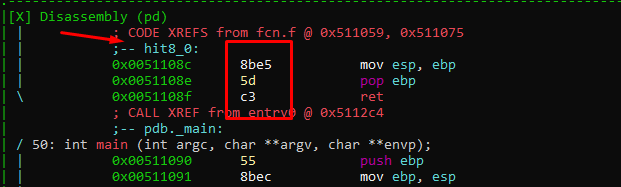
If I want to search for hexadecimal byte sequences, I press the "**:**" key and then type ”**/x**”.





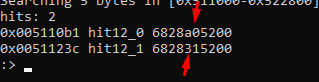
Here there is no need to search for references, the result is in the code, I go with the **v** key, then the **g** key and write the address I want to go to.



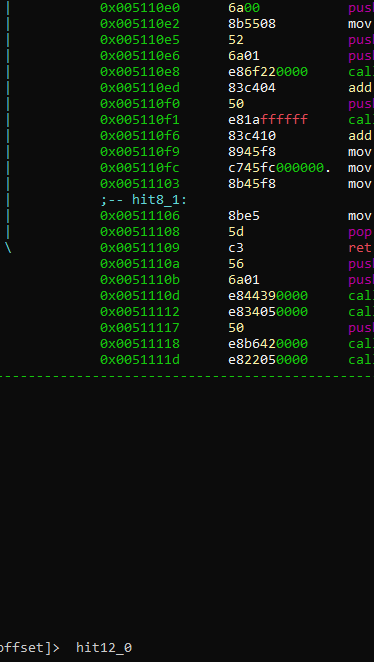


I can search using dots as wildcards.





I can go to the sequence using the address or with the tag.

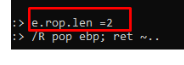




SEARCHING GADGETS

I can search gadgets to build a ROP.

I change the maximum gadget length to 2.

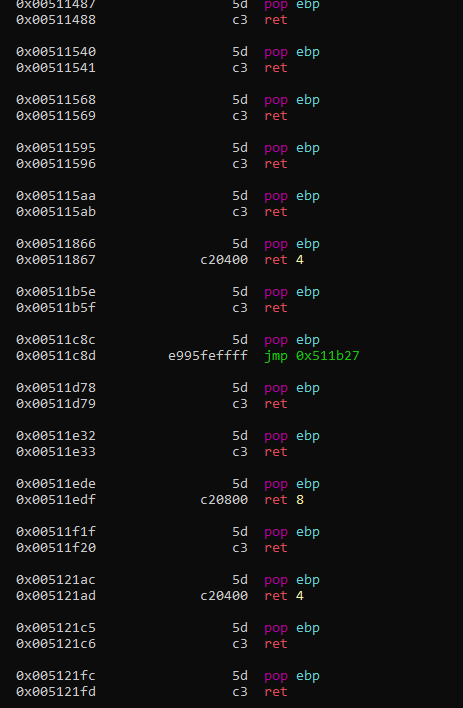


**e.rop.len =2**

And then with the command

**/R command1; command2**  ~ ..

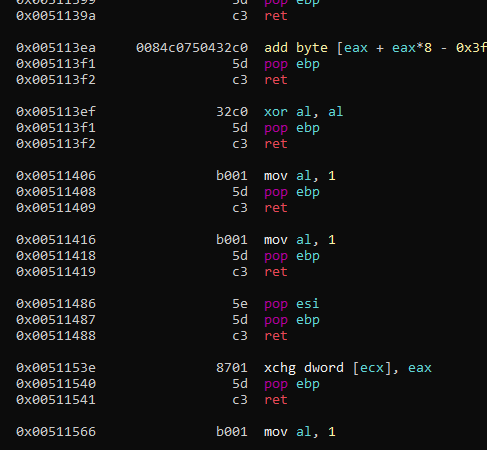
The modifier **~..** at the end of any search, allows you to move with the enter key and page up, page down, avoiding long unreadable lists.



I can continue to scroll down with ENTER or PG DOWN.

Changing the rop length to 3.

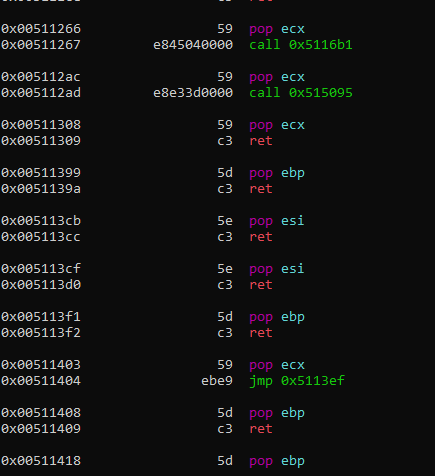
**e.rop.len = 3**

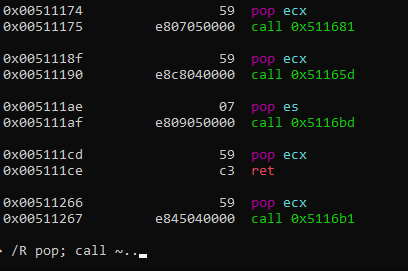


I can use regexp to search for gadgets with the command "/R/"

**/R/ pop e..; ret ~..**

It finds all gadgets with **POP EXX- RET**.

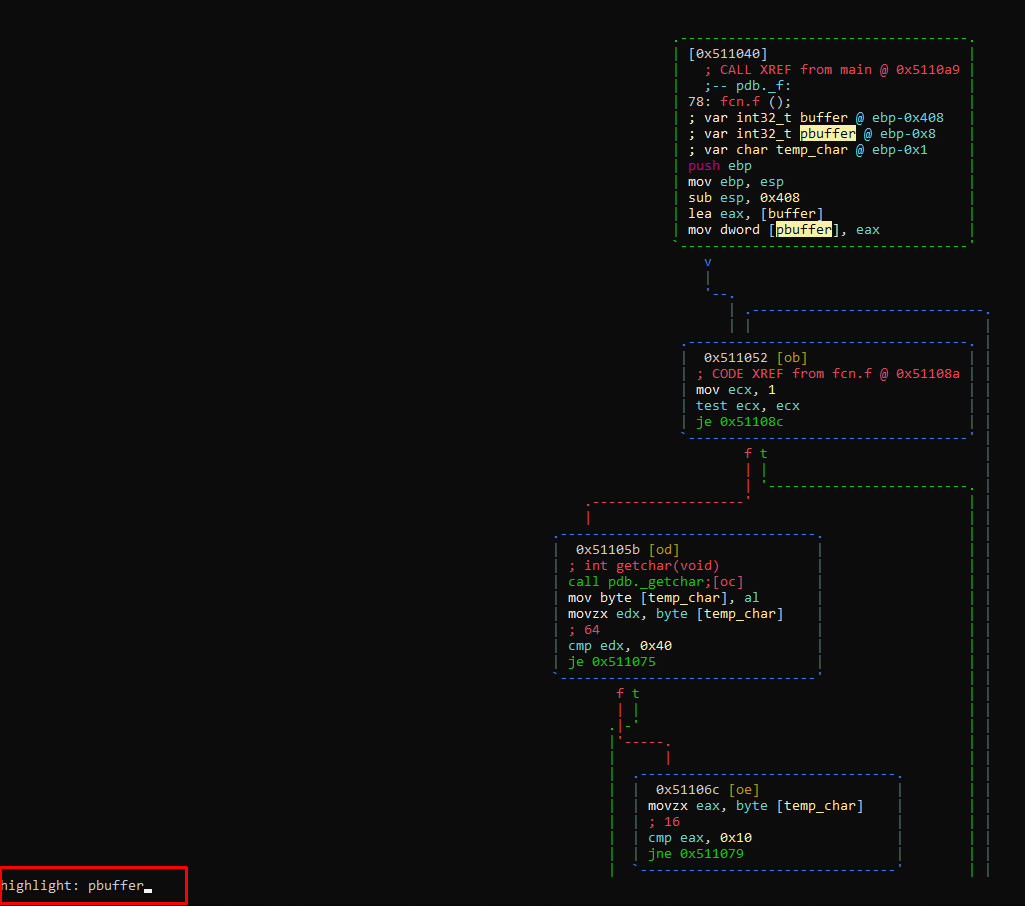




I can search for multiple combinations of instructions that, of course, I will use to solve this exercise later.

We already have an idea of ​​how we will search for gadgets, now I will try to create the base script.

In visual mode, I press the **“/”** key and type the name of the **pbuffer** variable, which is highlighted, so I can see where it is used.



Also in visual mode when I am in graphic mode by pressing the **p** key repeatedly I can change the way of visualization, the one I like the most is this.



It looks clearer.

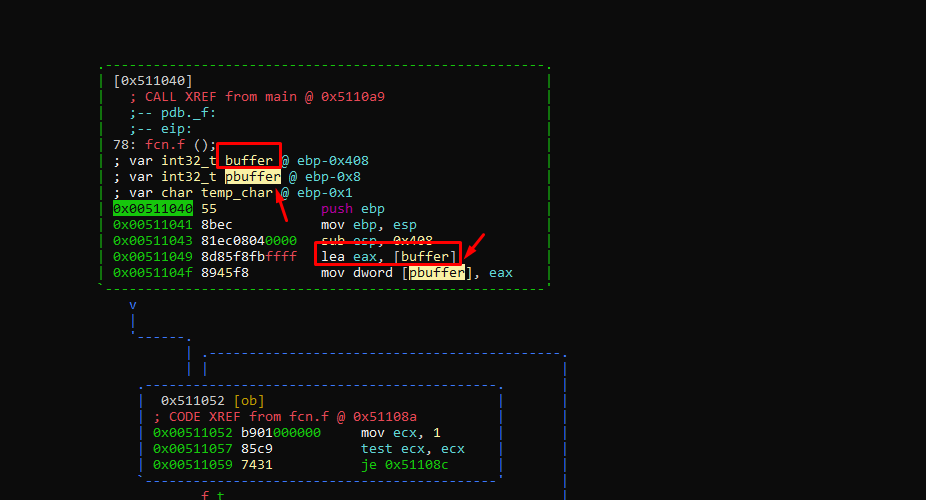
There are other display modes that can be accessed by pressing **p** repeatedly.

There is a block graph that marks only the CALL instructions. (Maybe it can be useful in certain circumstances).

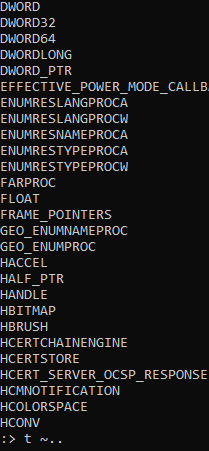


REVERSING STATICALLY

As we had analyzed, below the **buffer** there is the variable **pbuffer**, which stores the address of the **buffer**, it is a pointer-type variable.



I can see the types accepted by radare with the command **t** (I add "**~..**" at the end to be able to scroll through the result).



The **pbuffer** type is **"char \***", I can change it with the command.

**afvt pbuffer char \***

Or if I like PCHAR more

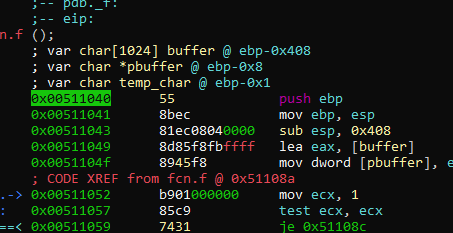
**afvt pbuffer PCHAR**

The **buffer** length is 0x400, since subtracting the offsets from the variable **buffer** and the next variable **pbuffer** gives me that value.

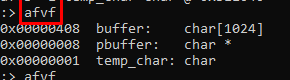
I can change the type too.



**afvt buffer char [1024]**

****

Now the variables were more accurate.



The **“afvf”** command shows me the static positions of the stack.

After the last variable shown in the list, there is the **saved ebp** (4 bytes) and then the **return address** (4 bytes).

**-0x00000408 buffer: char [1024]**

**-0x00000008 pbuffer: char \***

**-0x00000001 temp\_char: char**

**0 stored ebp**

**+4 return address**

Therefore I can calculate the distance from the **buffer** to **return address**.

I add 1024 bytes of the **buffer** variable, then a **pbuffer** variable of 4 bytes, 3 **empty bytes**, 1 byte of **temp\_char** and 4 of **stored ebp**.

1024 (buffer) + 4 (pbuffer) + 3 (empty) +1 (temp\_char) + 4 (stored ebp) = **1036**



That would be what I should send to overflow the buffer, until just before the return address, so my script should be like this.

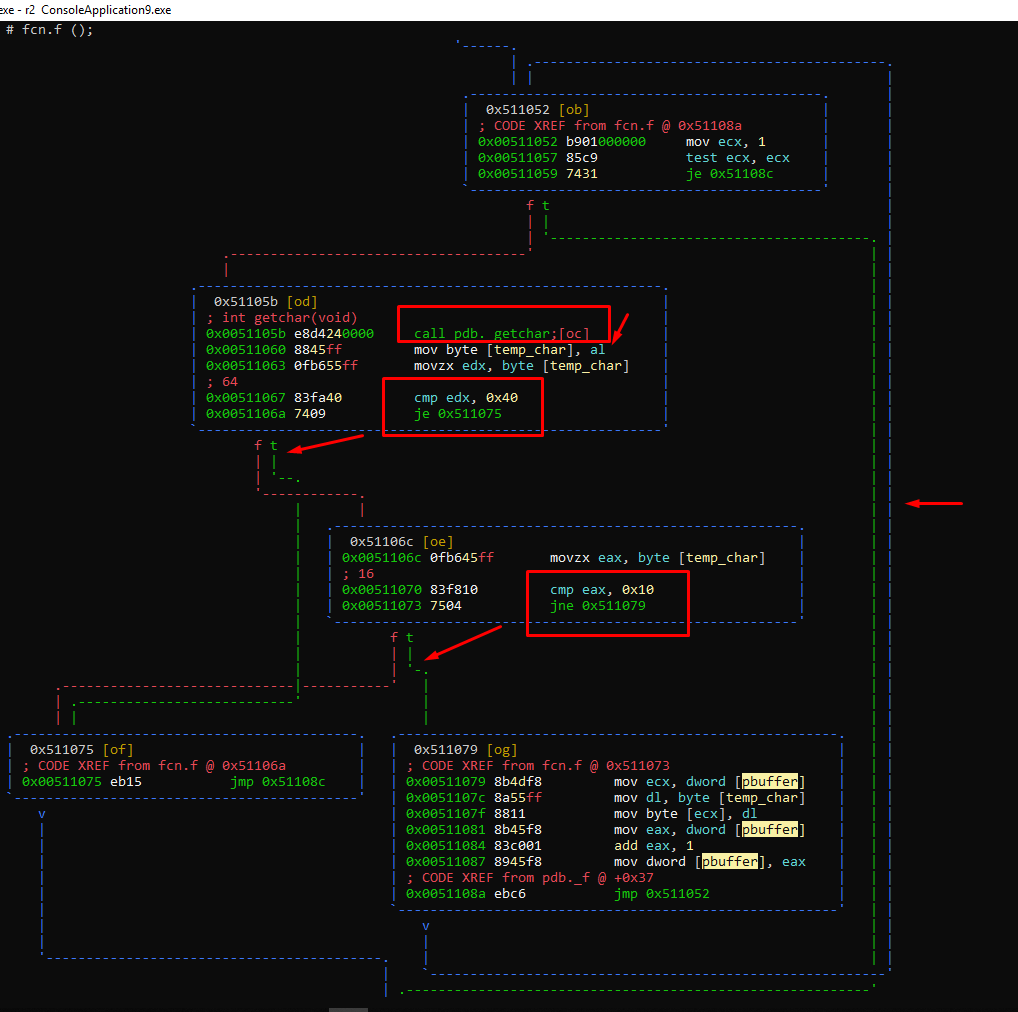


But let's analyze a little more, the questions are

1) Do I smash the return address with this script?

2) I can reach the retn instruction without breaking anything?

Well, I see that there is an infinite loop that has some way to get out.



Inside it, with the function **getchar** () it reads a character, stores it in **temp\_char**, compares it with **0x40**, if it is the same, it follows the green arrow of true and leaves the loop, if it is not equal to **0x40**, it continues and compares with **0x10**, if it is not the same, it continues inside the cycle, which is the same as saying that if it is equal to **0x40** or **0x10**, it leaves the cycle.

Inside the loop, It saves the character in the ECX content, since ECX is **pbuffer**, and its content is **buffer**, will be saved in the first position of the **buffer**.

After saving it increment **pbuffer**, so in the next cycle can save the next character in the second position.

If everything is correct, if I add a char 0x40 after smashing the RETURN ADDRESS, it will exit the loop and I will reach the RET.



Remember that since I’m in Python 3, It’s needed to put the letter **b** in front of each chain of bytes, in this case, it would be **b "\x40"**

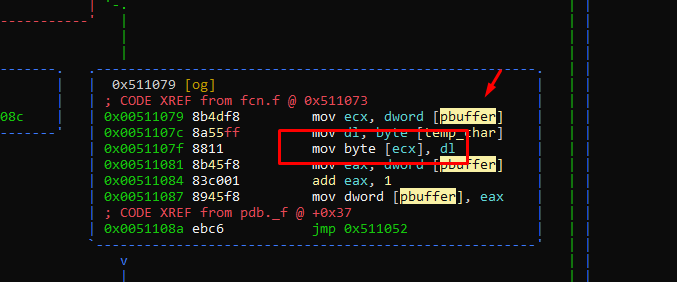
With this script theoretically I can smash the RETURN ADDRESS with 0x41424344, then put some pad bytes, and then exit the loop with the char **0x40**.

But, but ... not all are flowers, there are some problems.



Remember that below the **buffer**, there is the variable **pbuffer** that will be stepped on, before stepping on the return address.

That means that if you stepped on **pbuffer** with 0x41414141 for example, on the next cycle, it would try to write to [0x41414141]



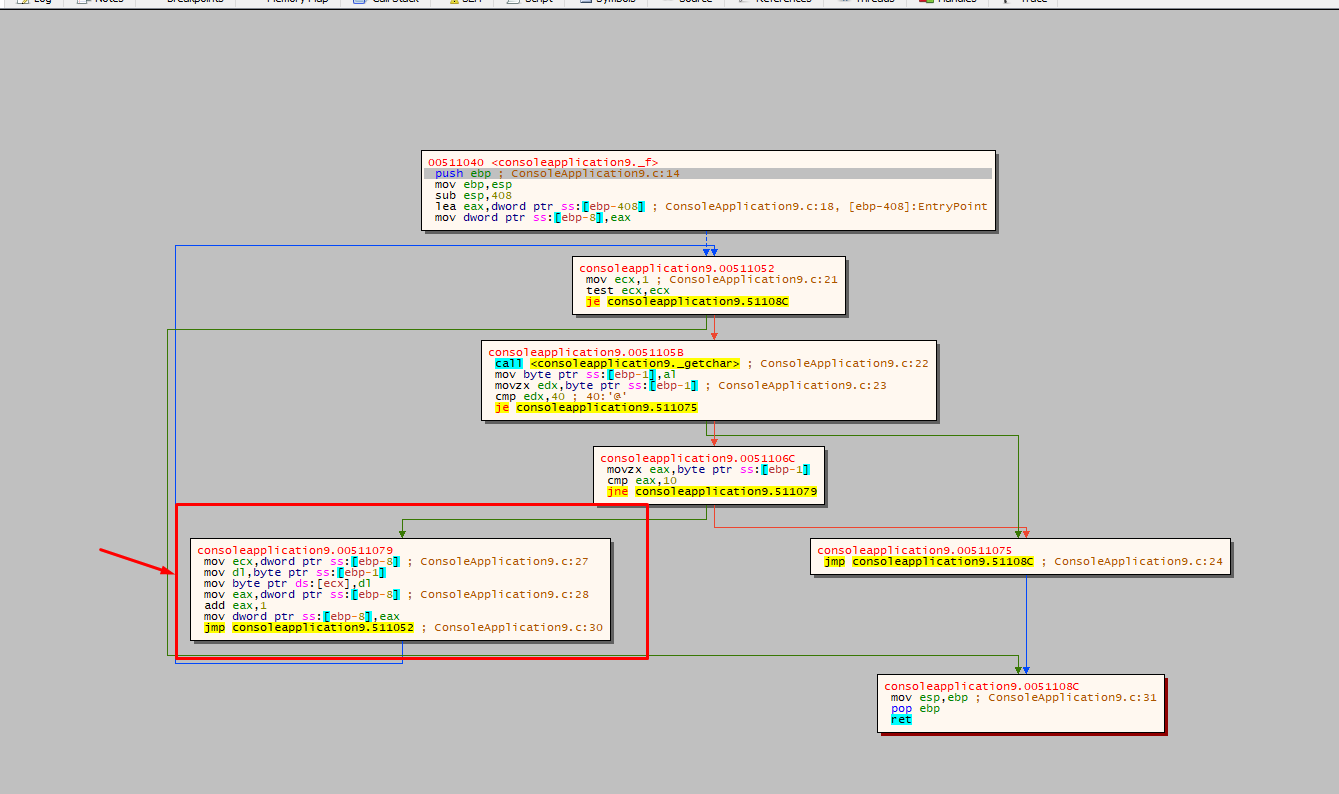
If I step **pbuffer** with the value 0x41414141, in the next cycle it would write in this address with **mov byte [ecx], dl**, and I could no longer step on the return address.

The reality is that **pbuffer** is smashed one byte in each cycle, so I can step on its lowest byte and skip the other three bytes of the address in the next cycle.

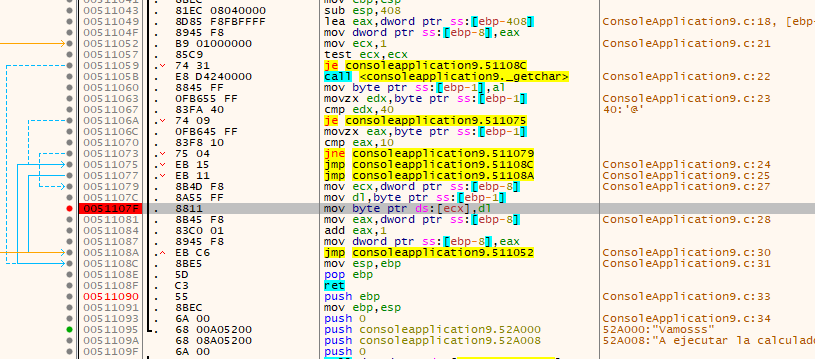
Let's run the script and attach it to view it on x64dbg.



I go to the function **f,** and I can see the “graph mode” with **g**.

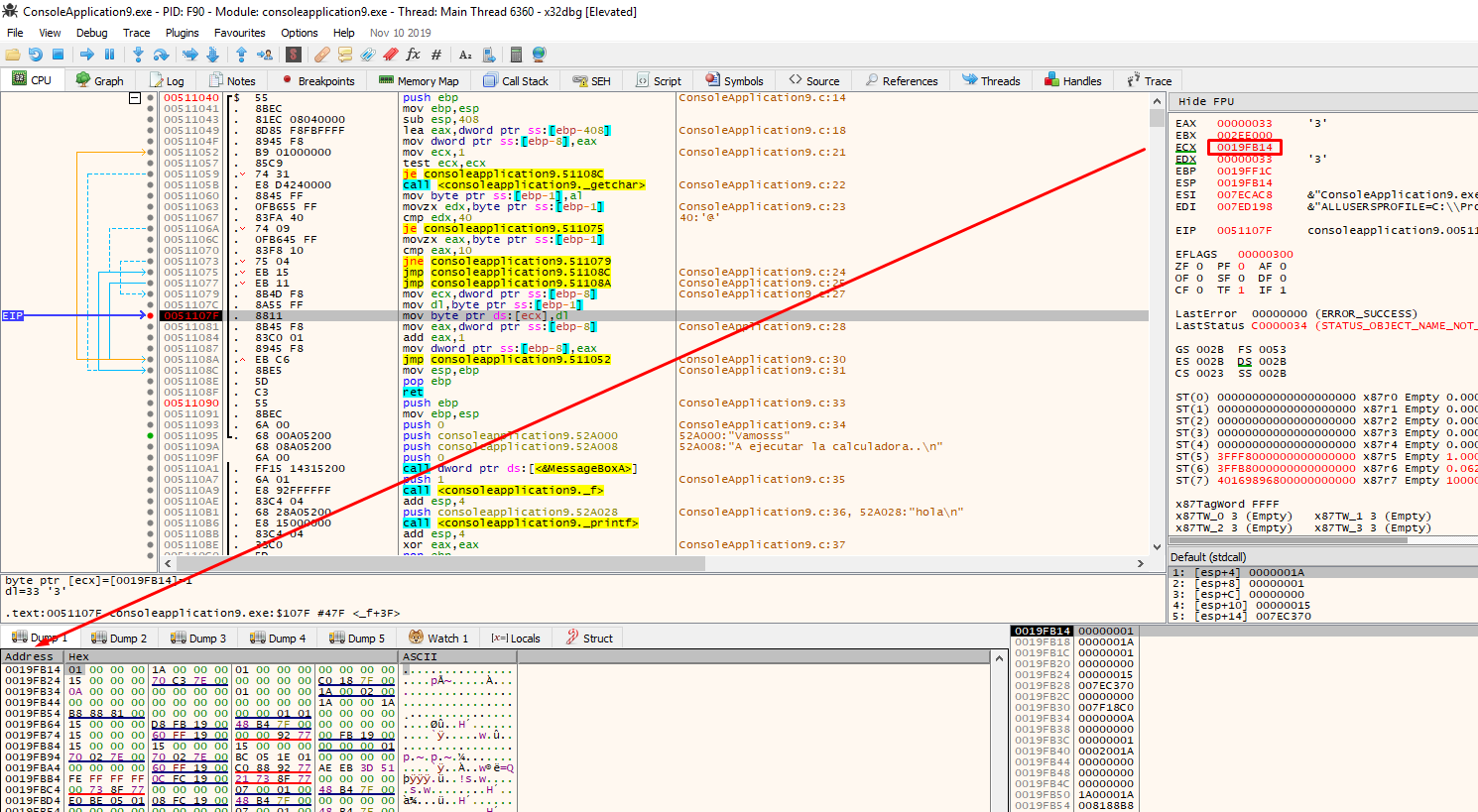


There I have the block where the pbuffer will be smashed. If I put a hardware breakpoint on it, the program will stop every time it is incremented, so it's not very useful.



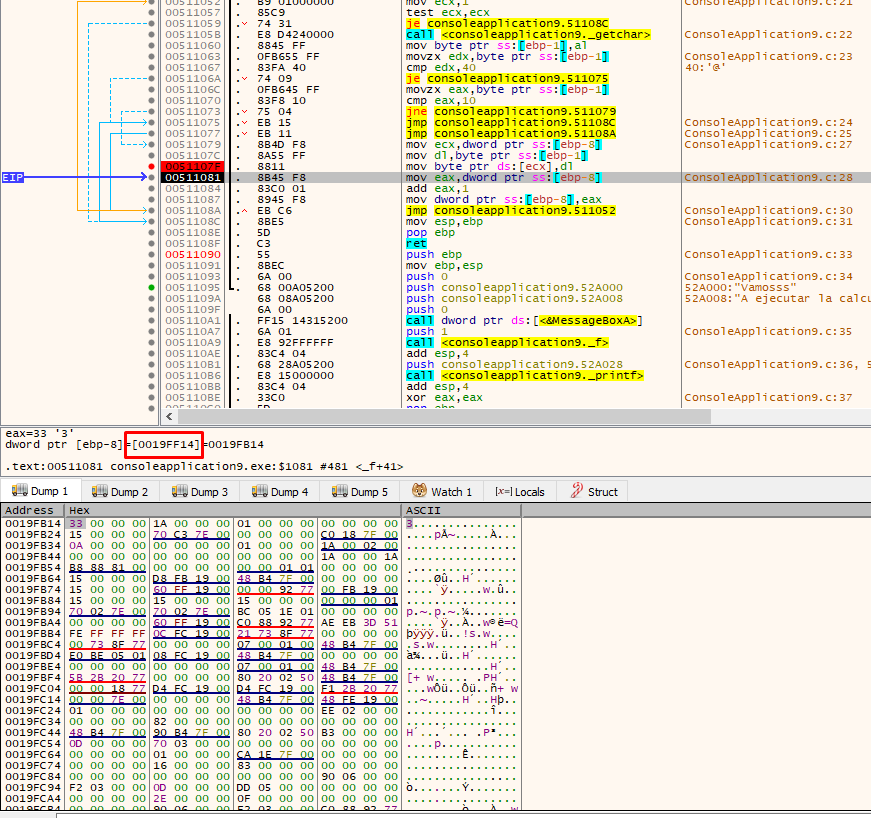
Let's run and accept the message box.

I can see how the buffer is filled in each cycle with ECX-FOLLOW IN DUMP.

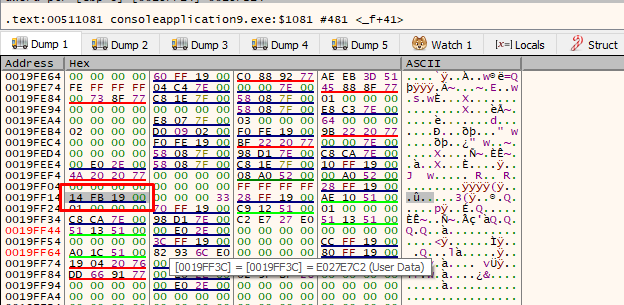


Let's find **pbuffer**.

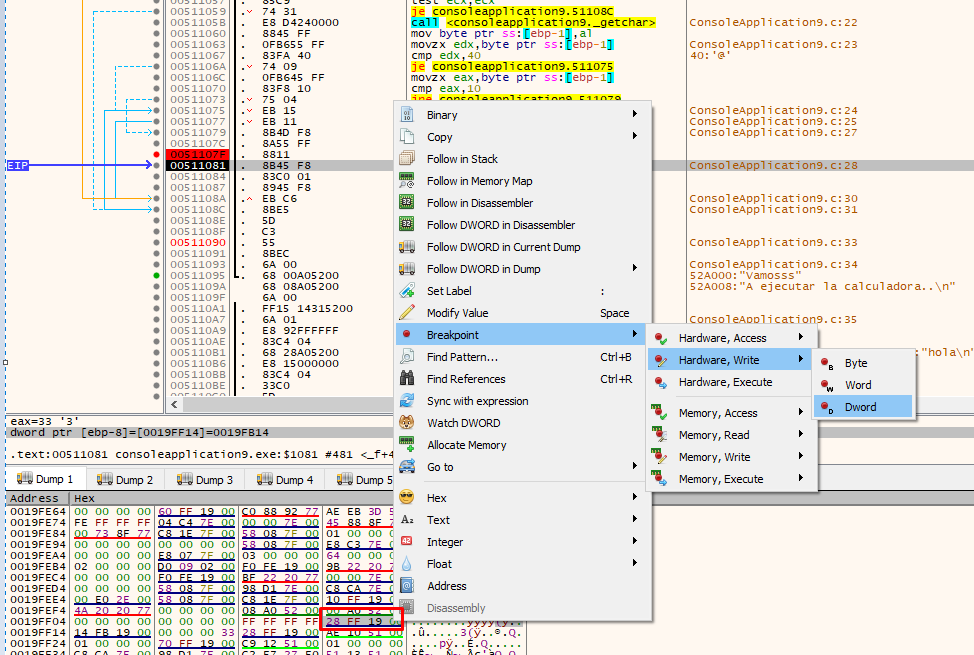
The next line reads the contents of 0x19ff14, and this is the **pbuffer** variable that stores the address of the **buffer**.



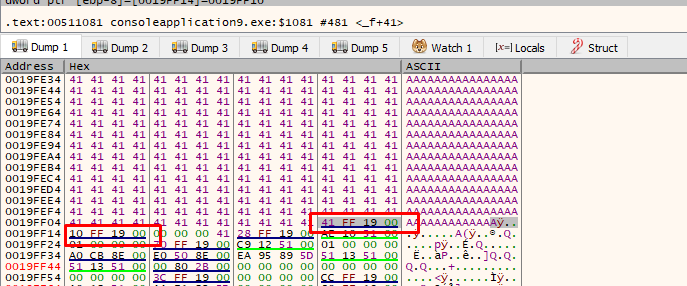
So when I step on **0x19ff14**, I will change the direction where it will write in the next cycle.



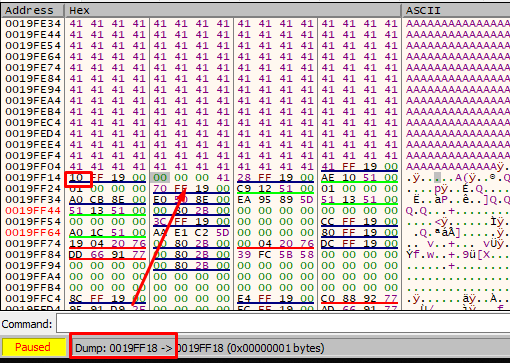
I can put a "hardware breakpoint when writing" in the address just above **pbuffer**, so that it doesn't stop with every loop.



Let's delete the breakpoint that I had initially set, and leave only the last one, so that the program stops when the pointer is about to step on, let's give RUN.

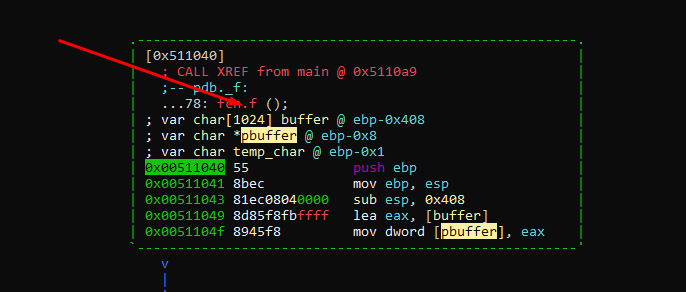


There it stopped, it will step on the previous 4 bytes, and then it will step on the lowest byte of **pbuffer**, if I make the byte to be 0x18, in the next cycle the program will write in 0x19ff18 and will skip the other three bytes of **pbuffer**.



How many bytes below the **buffer** is **pbuffer**?

The variable **pbuffer** is below the **buffer**, there are **1024** bytes from the start of the **buffer** to smash the lowest byte of **pbuffer**.



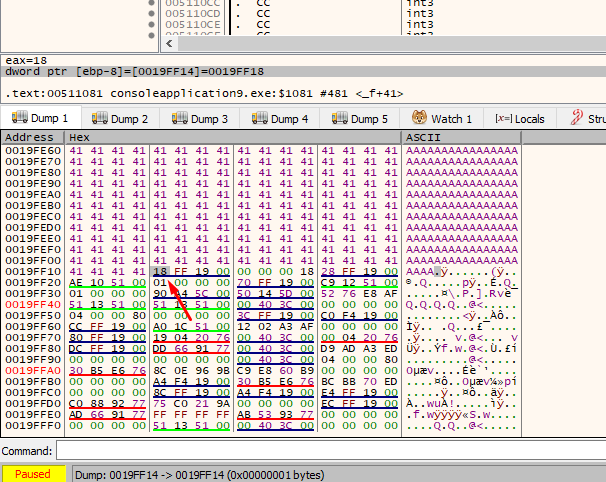
I must write 1024 bytes to fill the buffer, then a char 0x18, and then it will continue writing under **pbuffer**.

I have left from here, to get to the return address, the 3 **empty bytes**, the **temp\_char** byte, and the 4 of the **stored ebp**.

3 +1+ 4 = 8 bytes

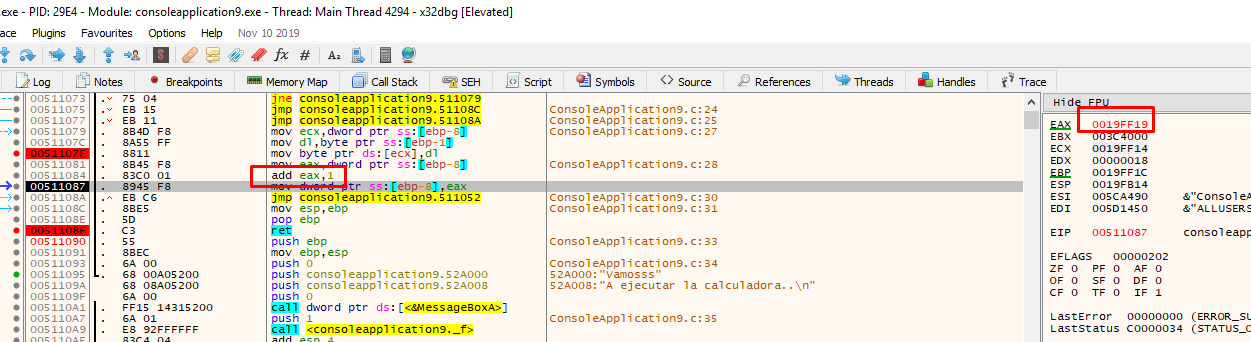


Let's check if this is the case.



I stepped on the lowest byte of the **pbuffer**, so in the next cycle I will write in 0x19ff18 and it wouldn't touch the remaining 3 bytes of **pbuffer**.

I will continue tracing with f8.

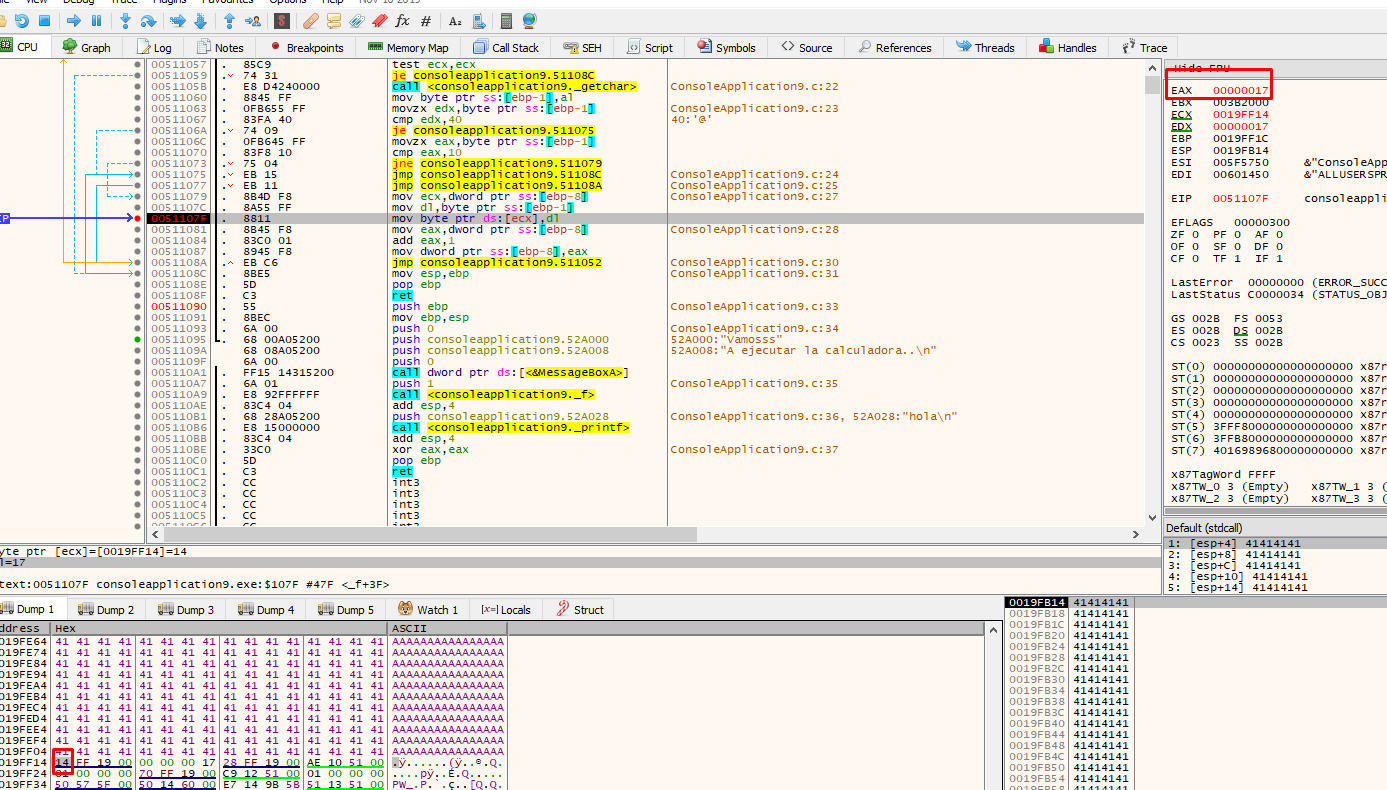


HOUSTON I HAVE A PROBLEM.

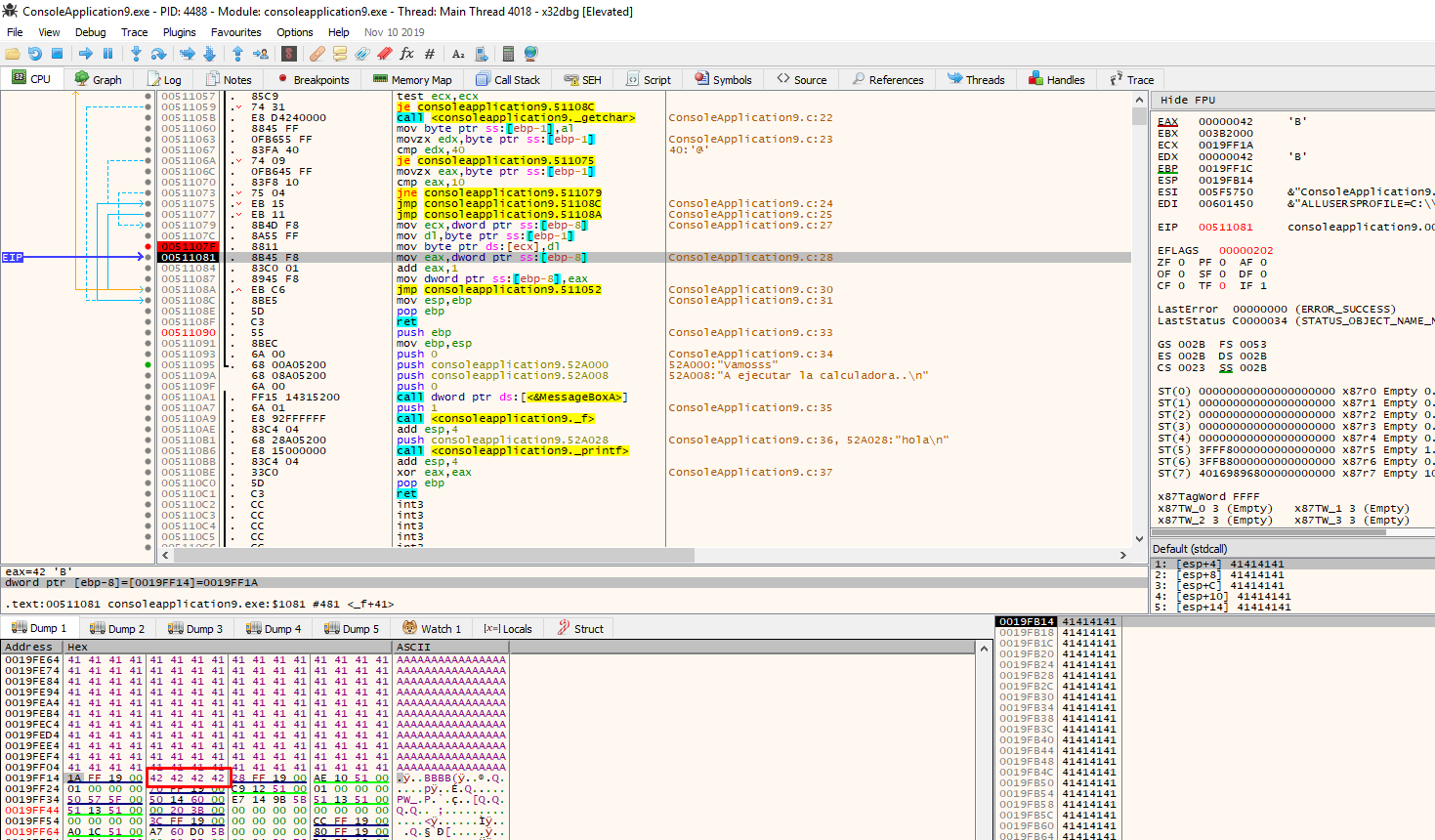
The program increases the pointer right after stepping on it, so I either step on it with 0x17, or I accommodate the entire payload, I better step on it with 0x17.



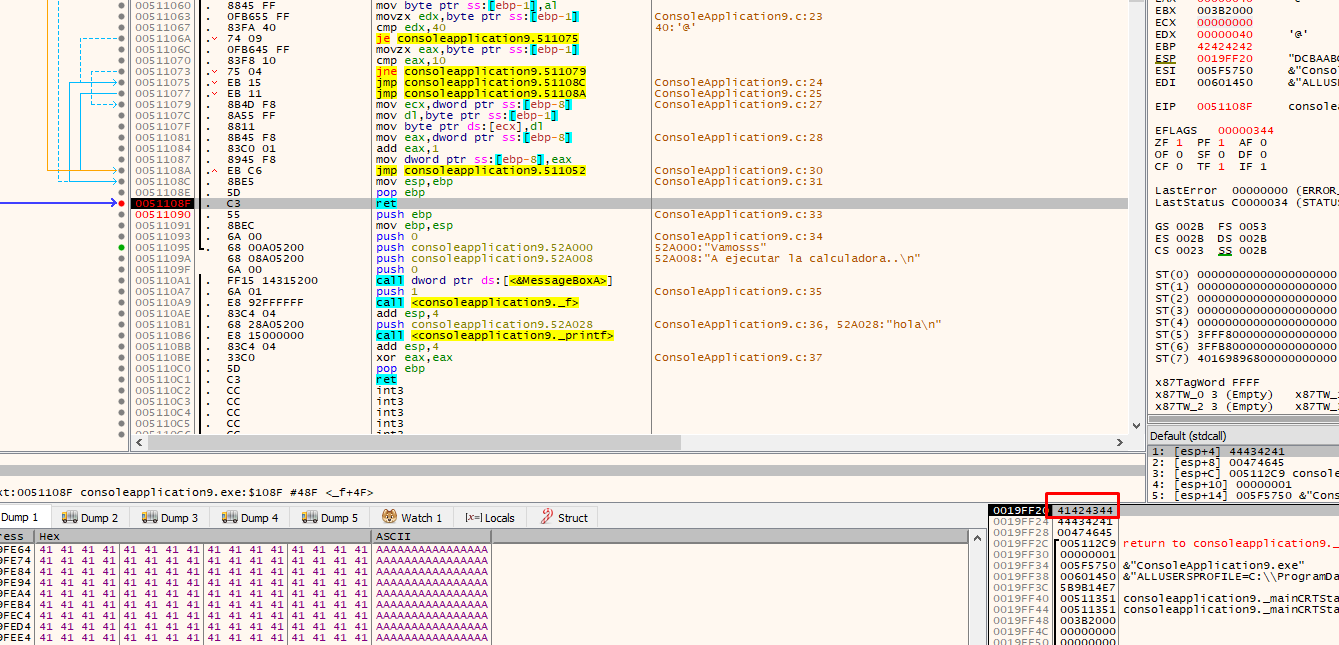
Let's try again.



It writes the char 0x17, let's continue tracing the loop with f8.



The program continues writing the 0x42, starting at 0x19ff18, I will get to the **retn** removing all the breakpoints and putting on breakpoint there.



BASIC SCRIPT TO START ROPING

*#!/usr/bin/env python*

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

import sys

from subprocess import Popen, PIPE

import struct

import sys

import codecs

import random

import string

payload = b"A" \* 1024 + b"\x17" + 8\* b"B"+ struct.pack(**"<L"**, 0x41424344) + **"ABCDEFG"** + b"\x40"

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

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

print(**"Enter para continuar"**)

p1.communicate(payload)

p1.wait()

Now I’m ready to ROP, which I will do in the next part.

Until part 16

04/26/2020

Ricardo Narvaja