# EXPLOITING AND REVERSING USING FREE TOOLS (PART 14)

As we have seen in the previous tutorials, creating a rop can be easy or very difficult depending on how is the vulnerability, and according to the 4 questions we have seen that indicates the level of difficulty.

According to the 4 questions, look at the example from the previous part:

1) Do you have modules that do not have ASLR? - **YES**

2) Do you have imported VirtualAlloc or VirtualProtect function in any module that does not have ASLR? -**YES**

3) Is the data already on the stack to start ROPING? **-YES**

4) Can I pass any character, that is, there are no invalid characters? **NOT, BUT AFFECTS LITTLE**

We can put a score from 0 to 4: initially adding one for each **NO.**

Of course, a level 0 ROP would be easier at first glance and a level 4 may be very difficult or impossible depending on the case, although it may change a little later in practice.

In the previous one we saw that using this scale, the rop would be level 1, although in practice the invalid characters affect so little, so in the case already done we could, after completing it, change the NO from 1 to only 0.5, which would finally give us a difficulty score of 0.5, according to the ricnar scale hehe.

These are the following two 32-bit and 64-bit examples, supposedly a little more difficult than the previous ones. First, we'll look at the difficulty we might have, before starting to build the ROP.

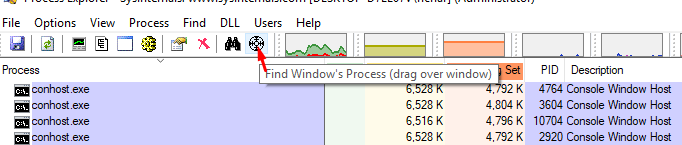
<https://drive.google.com/open?id=1IMiq0AqErEzWOiq78l0hA4Mb9S80RGXb>

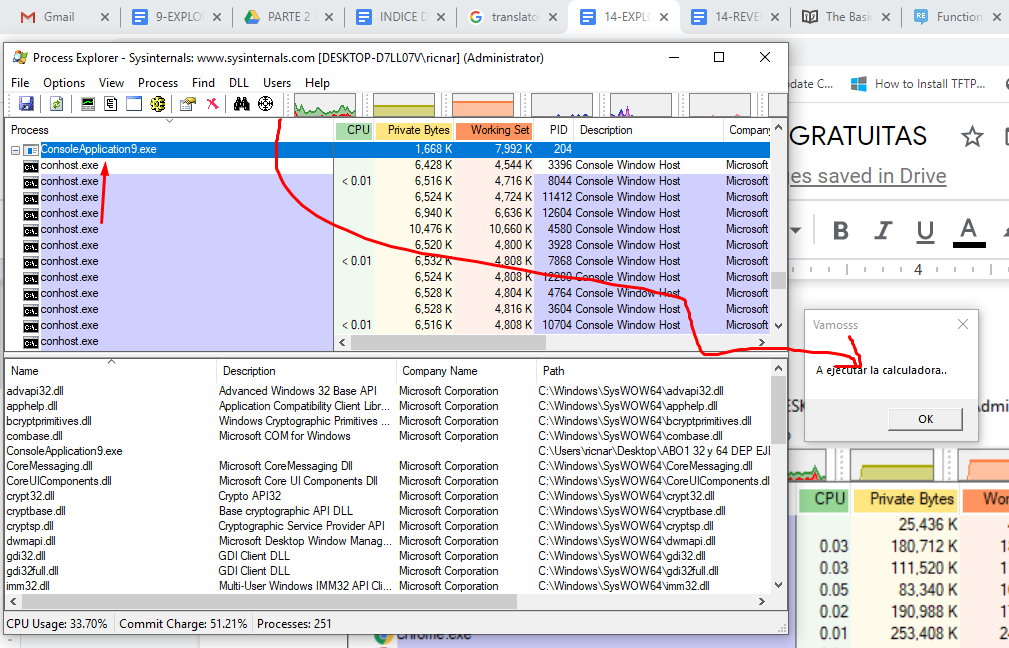
Analyzing The Difficulty Of The Second Exercise Of 32 Bits

Let's go answering the questions:

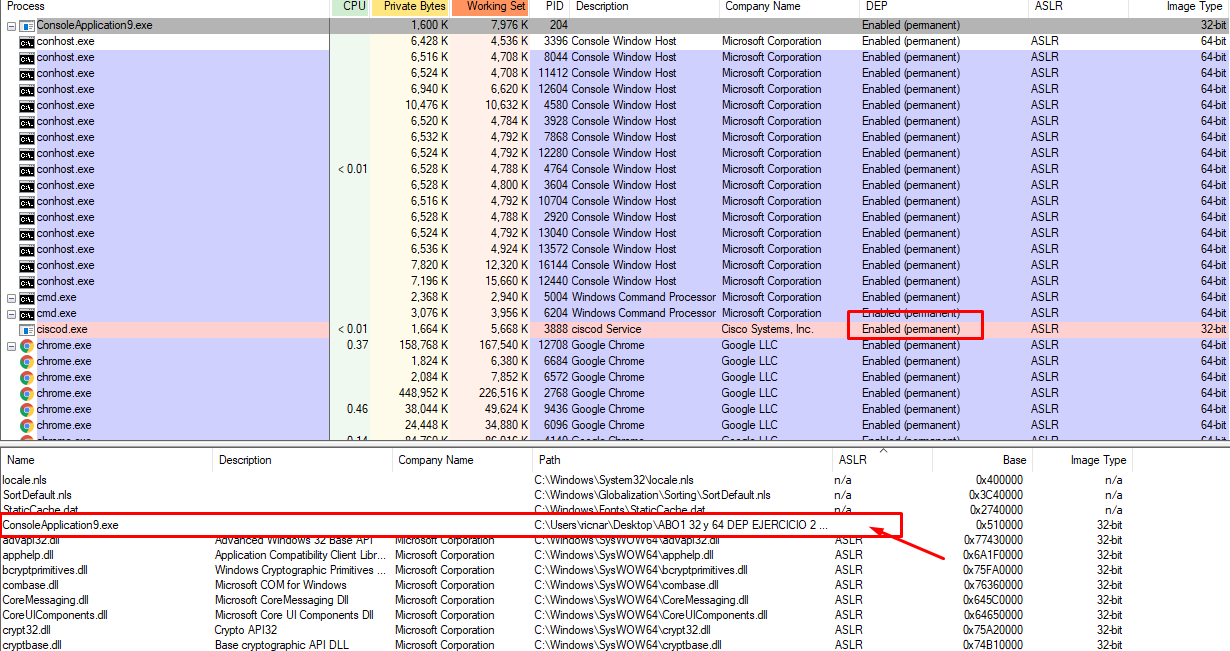
1) Does the program have modules that do not have ASLR?

Let's run it and when the MessageBoxA comes out, let's look at Process Explorer if there are modules without ASLR.





We click and drag to the MessageBoxA window, where we release and it will show us which process it corresponds to.

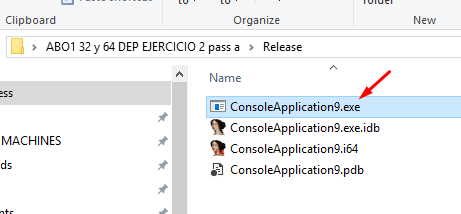


It has DEP enabled with which we will have to build a ROP to resolve, and it has a module without ASLR (the same executable), so the answer to the first question is **YES**, so we do not add anything (remember that we only add 1 for each **NO**).

**PARTIAL SCORE = 0**

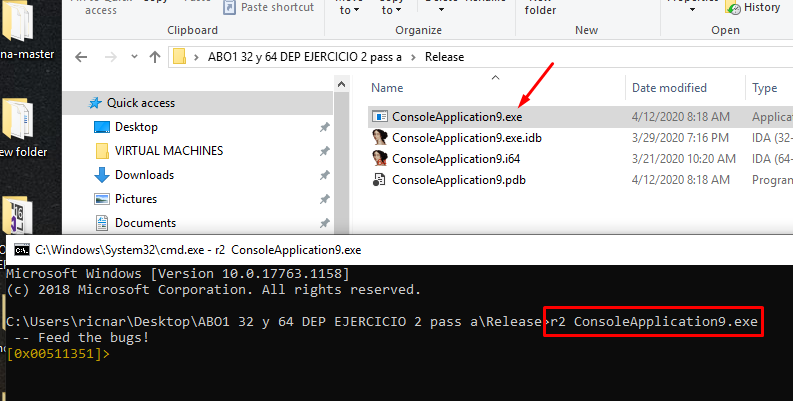
2) Do you have imported VirtualAlloc or VirtualProtect function in any module that does not have ASLR?

We will use RADARE to open the 32-bit example first.



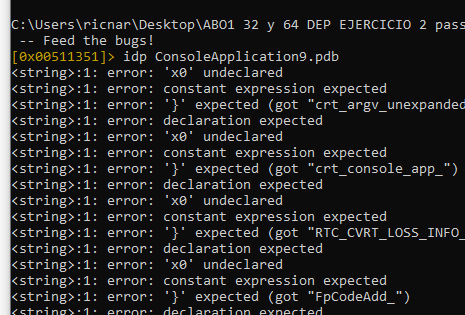
I open a **cmd** in the folder where the executable is located, and I type.

**r2 ConsoleApplication9.exe**

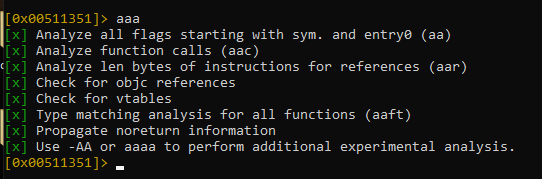


I load the symbols with idp command.

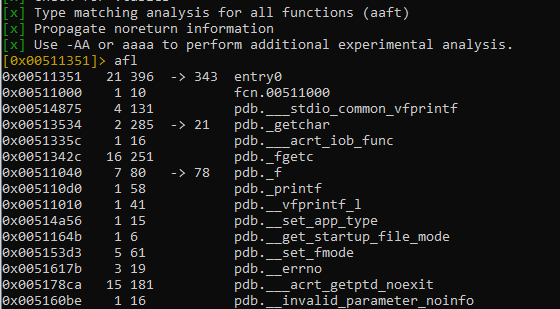
**idp ConsoleApplication9.pdb**



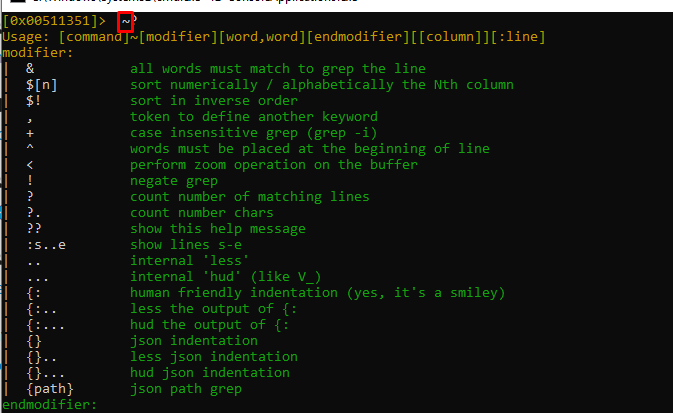
And then, using the **aaa** command, I analyze the program.



Using the **afl** command I can list all the functions.

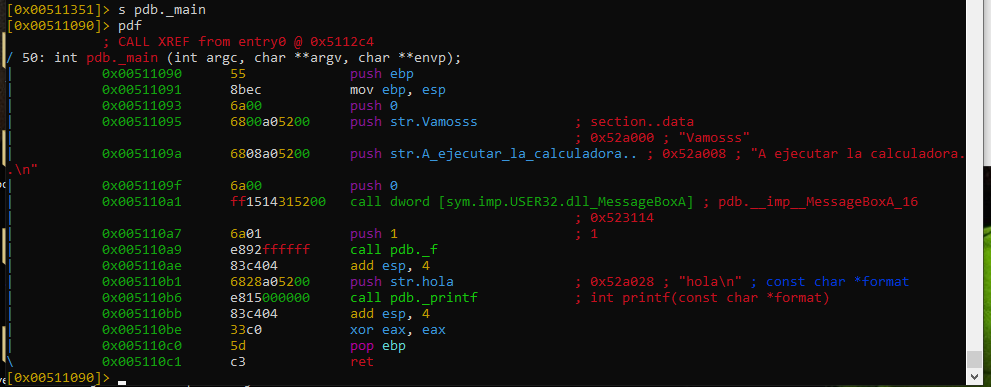


If we want to filter the results, radare has an inbuilt grep included using the **~** symbol.





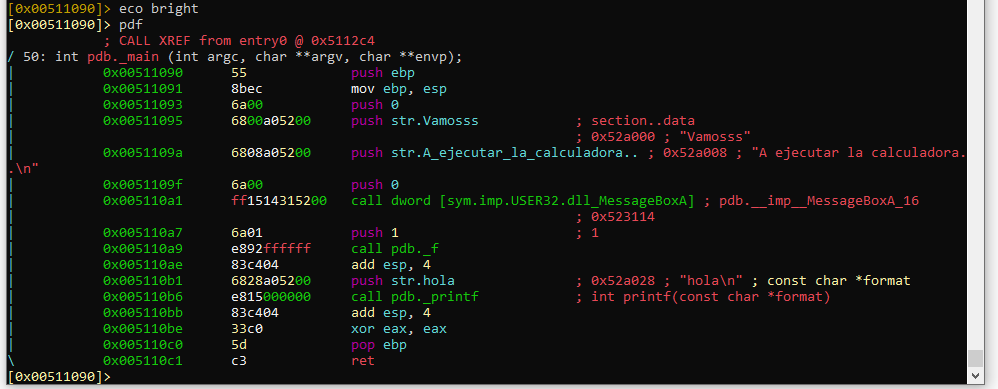
Let's go to main with the **s pdb.\_main** command.



I create a project with the **Ps ROP32** command.

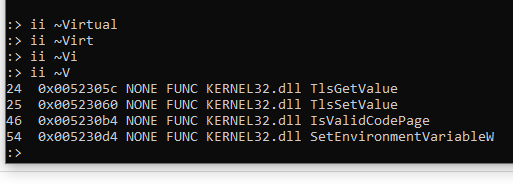
****

More visible with **echo bright**.

****

I see that the call to VirtualAlloc that we had in the previous part is not there, which made our work easier since this function was imported.

Let's make a list of the imported functions to see if we have them using the **li** command and filtering with **~**.



It’s not imported, the answer is NO.

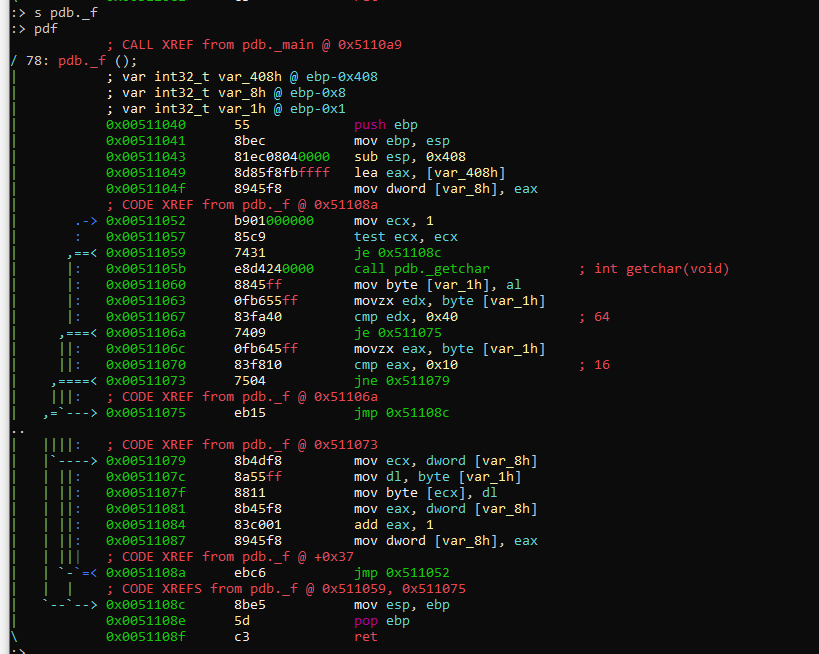
**PARTIAL SCORE = 1**

3) Is the data already on the stack to start ROPING?

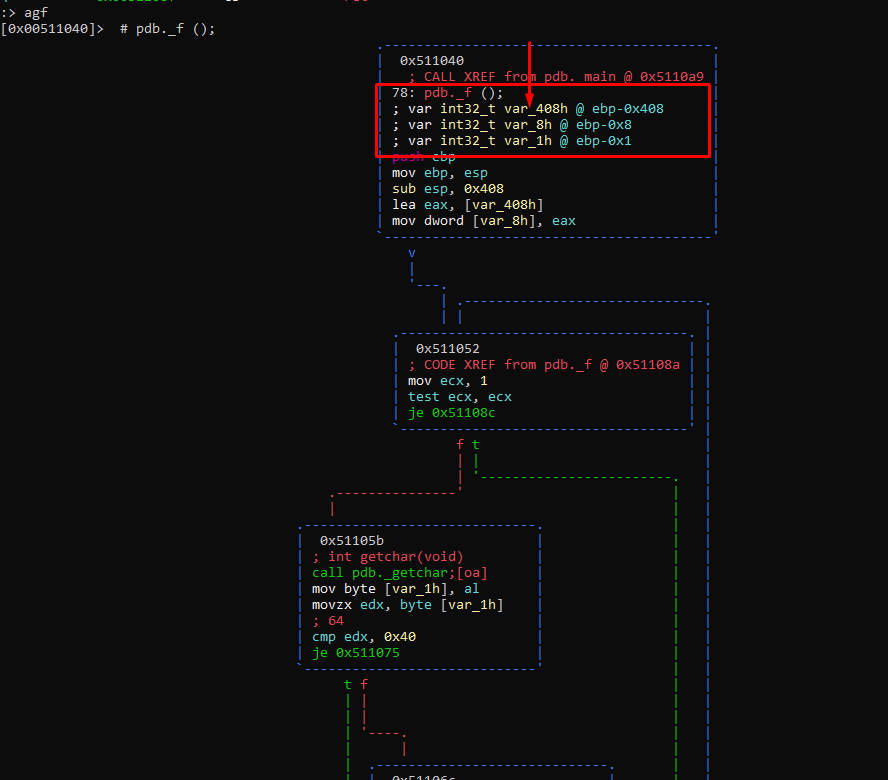
To see that, we must analyze the vulnerability of the **main** function. Nothing is seen, perhaps it is inside the function called **f.**

We could see the function **f** with **s pdb.\_f**

But better we use the visual mode, entering it with **v,** pressing **c** we control the cursor and with the direction arrows we go down to the CALL to the function **f** and we press ENTER.



In visual mode, pressing **c** to exit cursor mode and pressing the space bar, we can see the blocks. ( press space bar again to return)



We can see three variables: the top one has considerable space with the next one, so it should be a buffer on the stack.

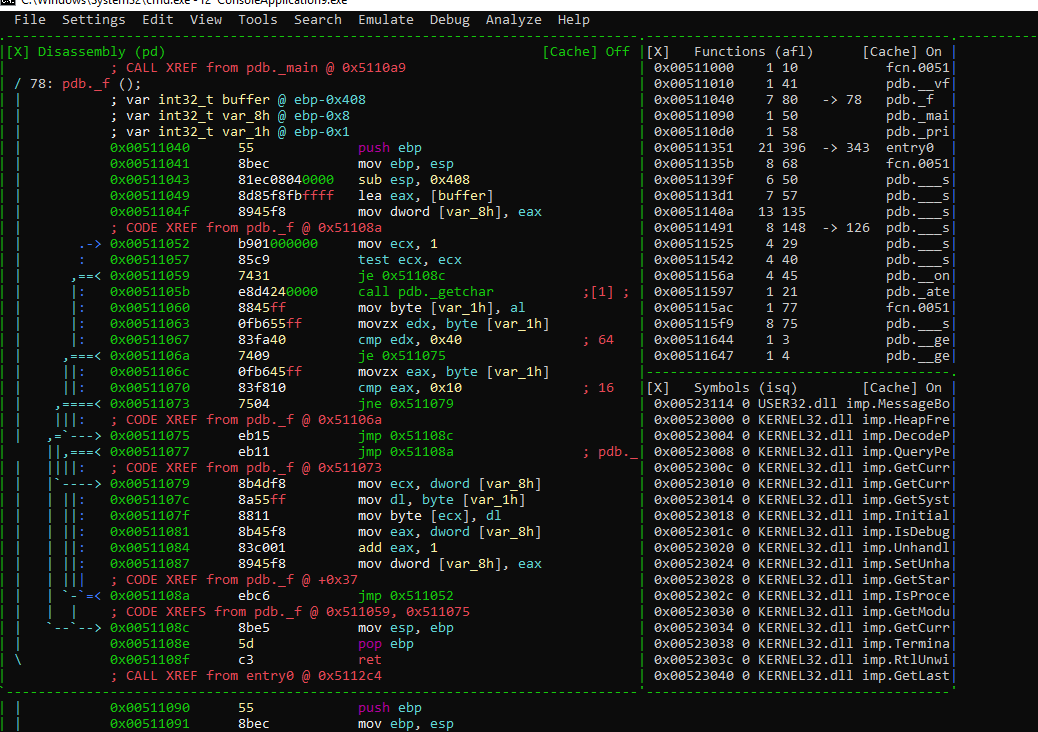
It is -0x408 with respect to ebp and the next variable is -0x8 with respect to ebp. There are 0x400 bytes difference between them, which is the length of the buffer.

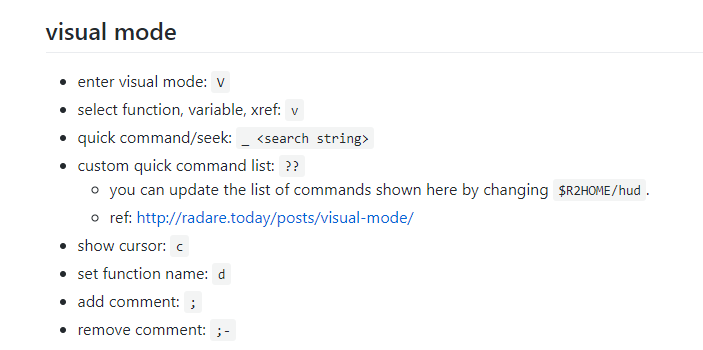
In visual mode I can also press the key **:** and then I rename using the command **afvn buffer var\_408h.**



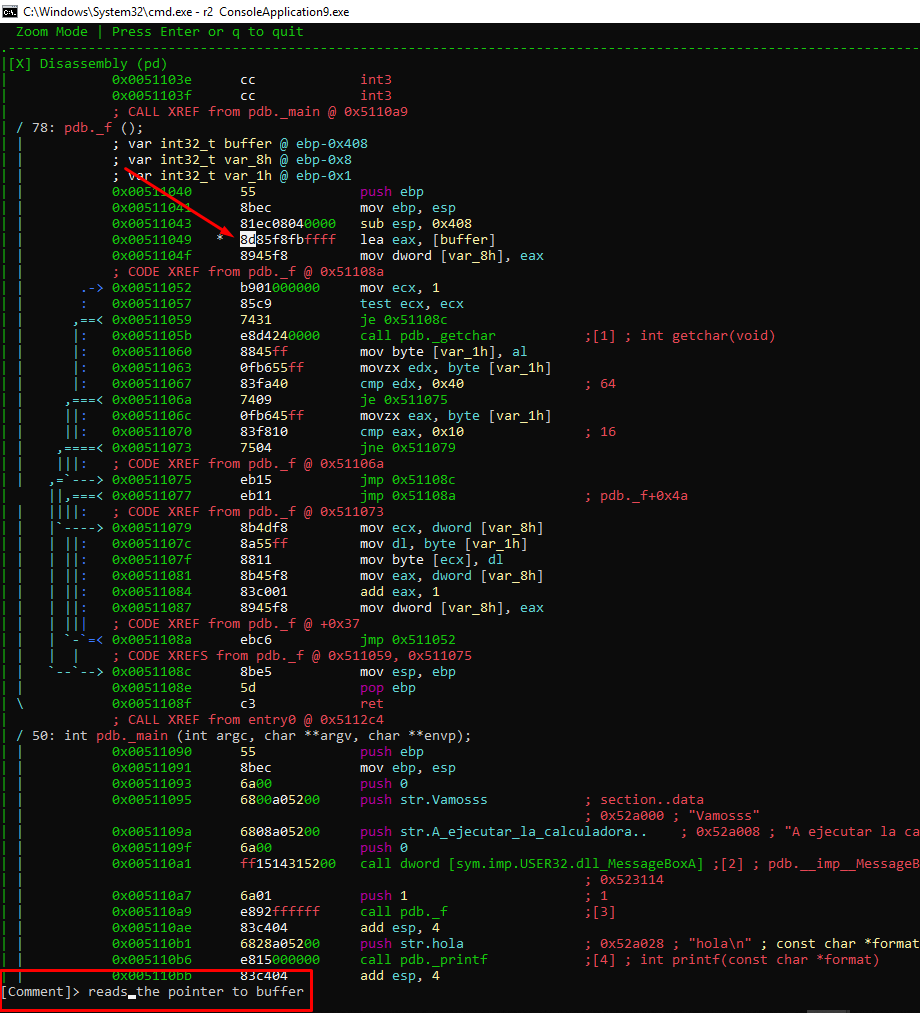
And then **v** again for refreshment.

Pressing ENTER can use the ZOOM MODE of the actual position of the cursor. (ENTER again to quit)



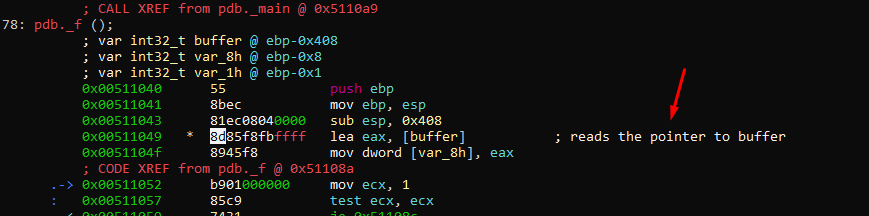


As we saw, pressing the **c** key we control the cursor with the direction arrows (c to exit cursor mode).

****

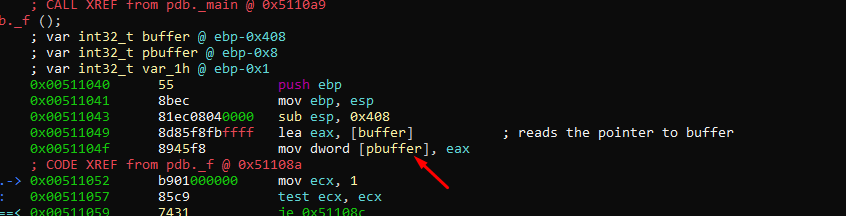
I go with the direction arrows to the line that I want to comment and press the key **“;”** and add the comment and hit ENTER.

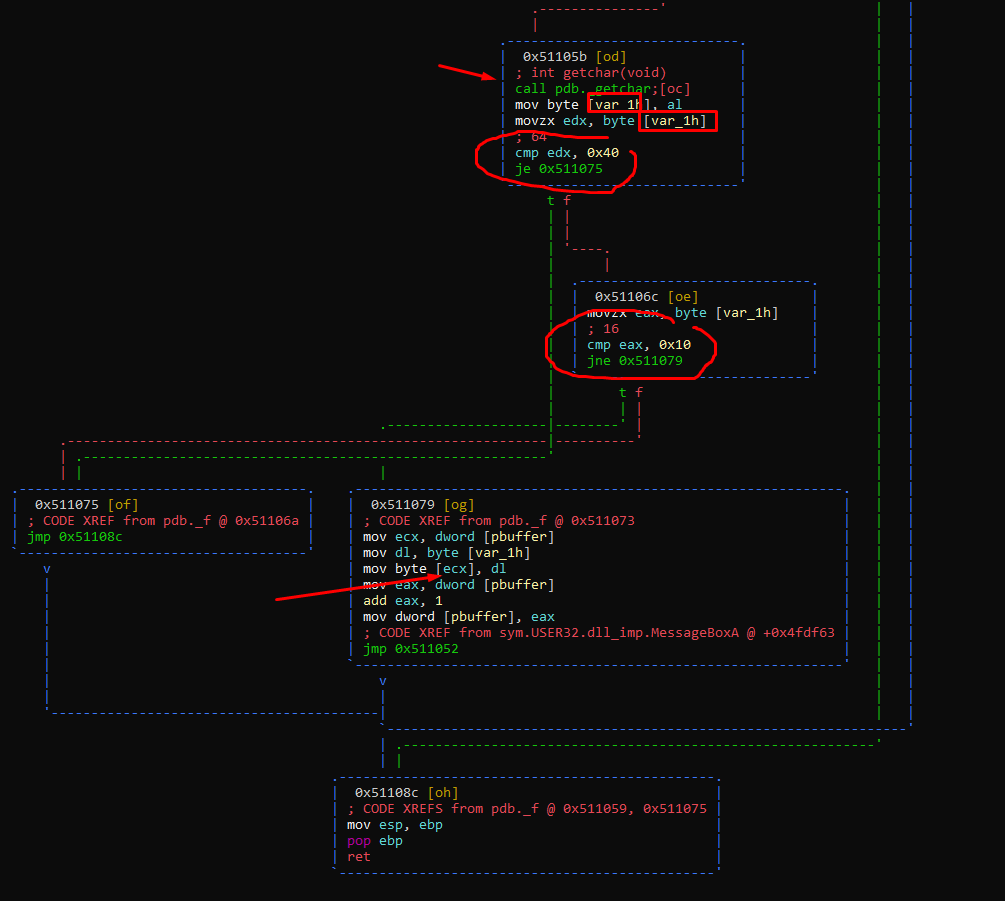
In these visual mode commands there is no need to refresh.



We see that it saves the pointer to buffer in **var\_8h**, I will rename that variable to **pbuffer**.

I press the key **:** next the command **afvn pbuffer var\_8h** and next **v** to refresh**.**





We see that there is a cycle where it reads a character every time it repeats with the function **getchar()**, saves it in the variable **var\_1h,** which is a single byte variable, and then compares them first with **0x40** and then with **0x10,** and exits the loop if it is equal to any of them.

If it is not one of those values, it saves it in the buffer since ECX is equal to pbuffer and then saves it in its content, that is, in the buffer, increases pbuffer and repeats the cycle.

We see that there are no protection cookies, so you can fill the buffer and step on the return address, until the program finds a byte 0x40 or 0x10.

The answer to the question is YES since my data will be on the stack and I will be able to ROP directly from there.

**PARTIAL SCORE = 1**

4) Can I pass any character, that is, there are no invalid characters?

As we saw, we can only pass characters that are **0x40** or **0x10** if we want to finish copying. Therefore, the data copied to the stack that formed the ROP cannot have **0x40** or **0x10**, the same as the **SHELLCODE** so the answer is **NOT**.

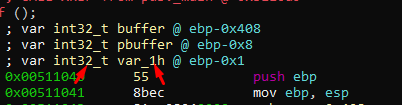
We will see if it affects a lot or not, for now, we add one to the score.

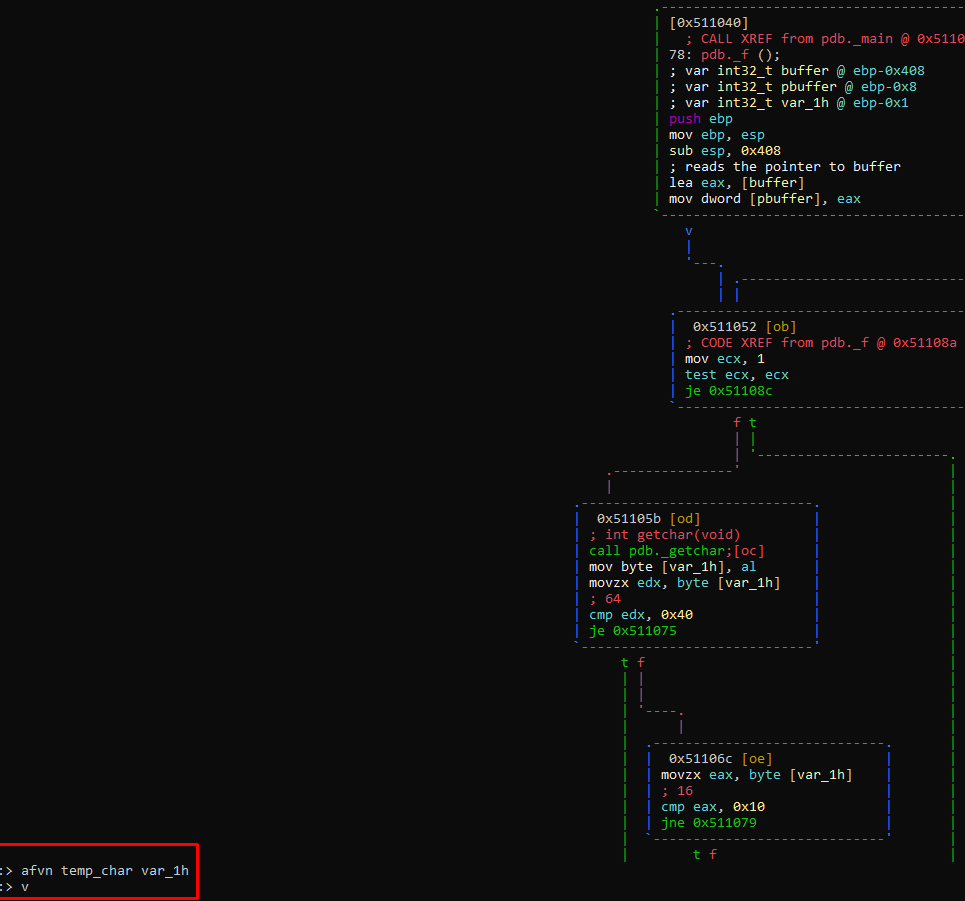
Final Score

**FINAL SCORE = 2**

As we see, this partial analysis tells me that this ROP with **SCORE = 2** should be more complicated than the one we saw earlier, which was **SCORE 0.5.**

We’ll rename the variable to **temp\_char** and we’ll correct the type using **afvn temp\_char var\_1h** and next **v.**

****

****

To change the type of **char** we will use the command **afvt temp\_char char** and next **v.**

I will save the project with **ps ROP32** to continue in the next part, and because I would like you to try to solve it, don't forget that a failure teaches us more than something solved easily.

A Tip To Help You To Solve.

**A TIP :> Be careful when copying to the stack with pbuffer that is below the buffer and it may crash the program and not allow you to get to the return address.**

04/18/2020

We are still in quarantine but we are still working.

See you in part 15.

Ricardo Narvaja