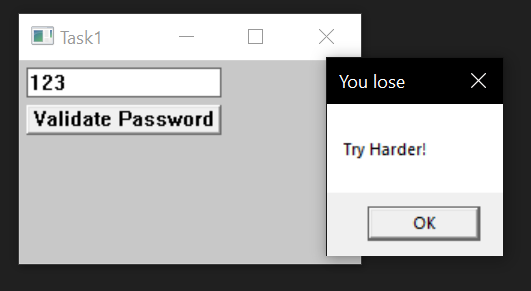
Lab Session 0x04

**4.1. Task: debugging in Windows**

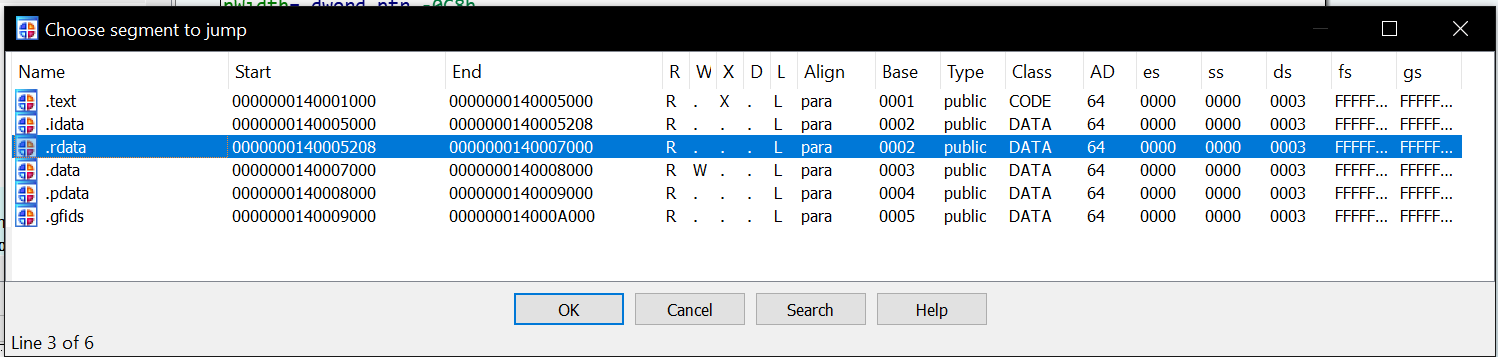
* *Open the binary in IDA and identify the password checking function (same procedure as in lab session 0x03) and the final* ***if*** *condition that verifies whether the password is good or not. Also, figure out which function is* ***sprintf****; (2p)*

Let’s start the .exe to see if we can gather some strings:

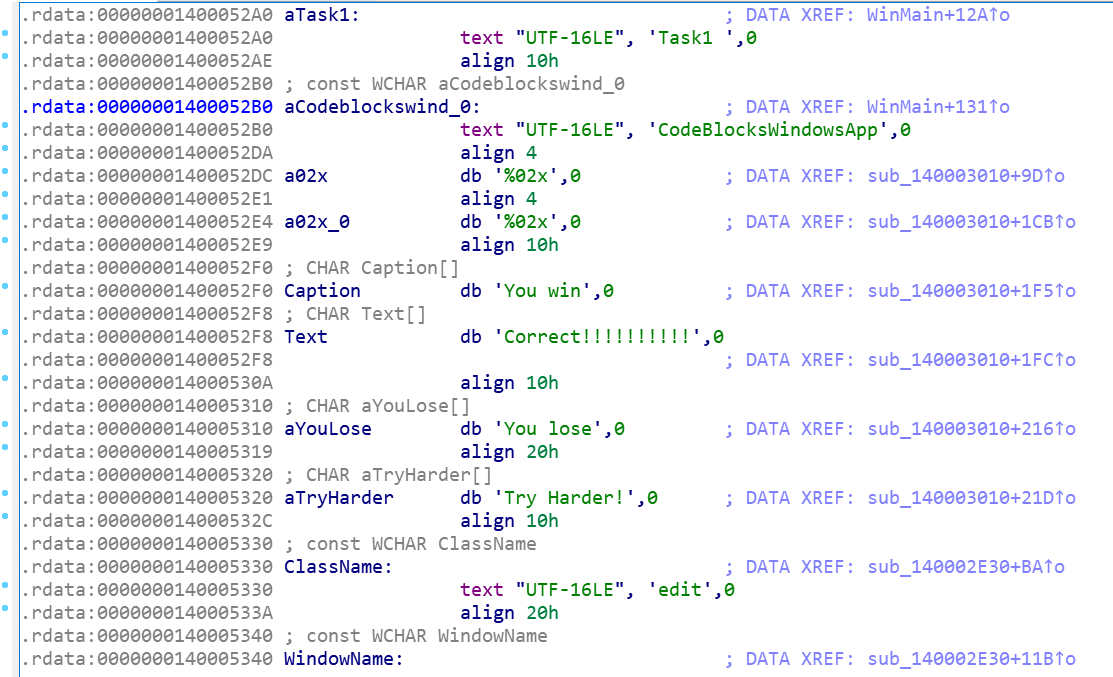


We can notice a few strings: „*Validate Password*”, „*Try Harder!*”, „*OK*”, „*You lose*”.

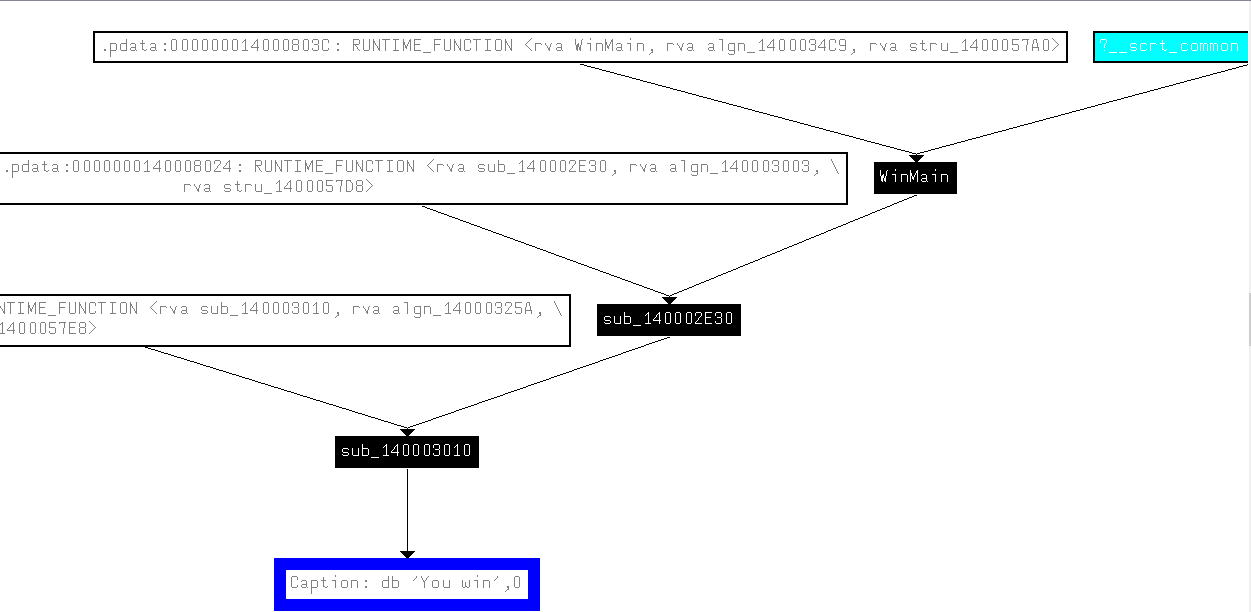
We open *task1.exe* with ***IDA → CTRL+S → .rdata*** and we start searching for the strings.



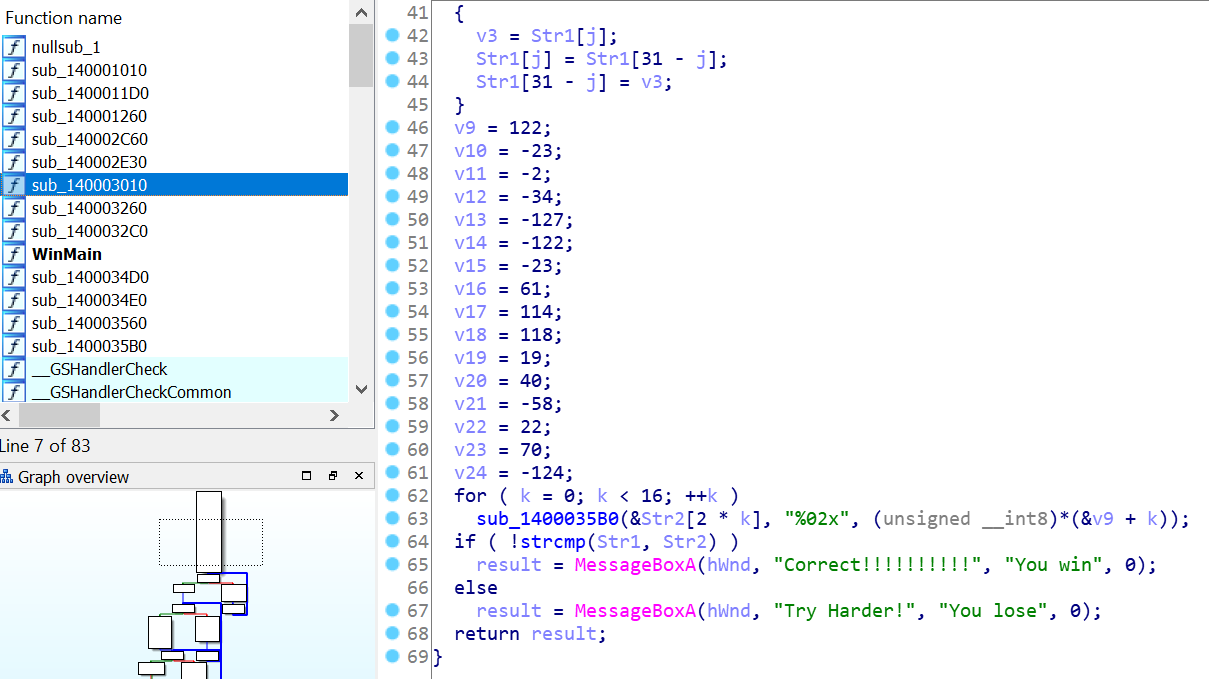
We found them:



We choose „*You win*” and ***RIGHT CLICK → Xrefs graph to***:

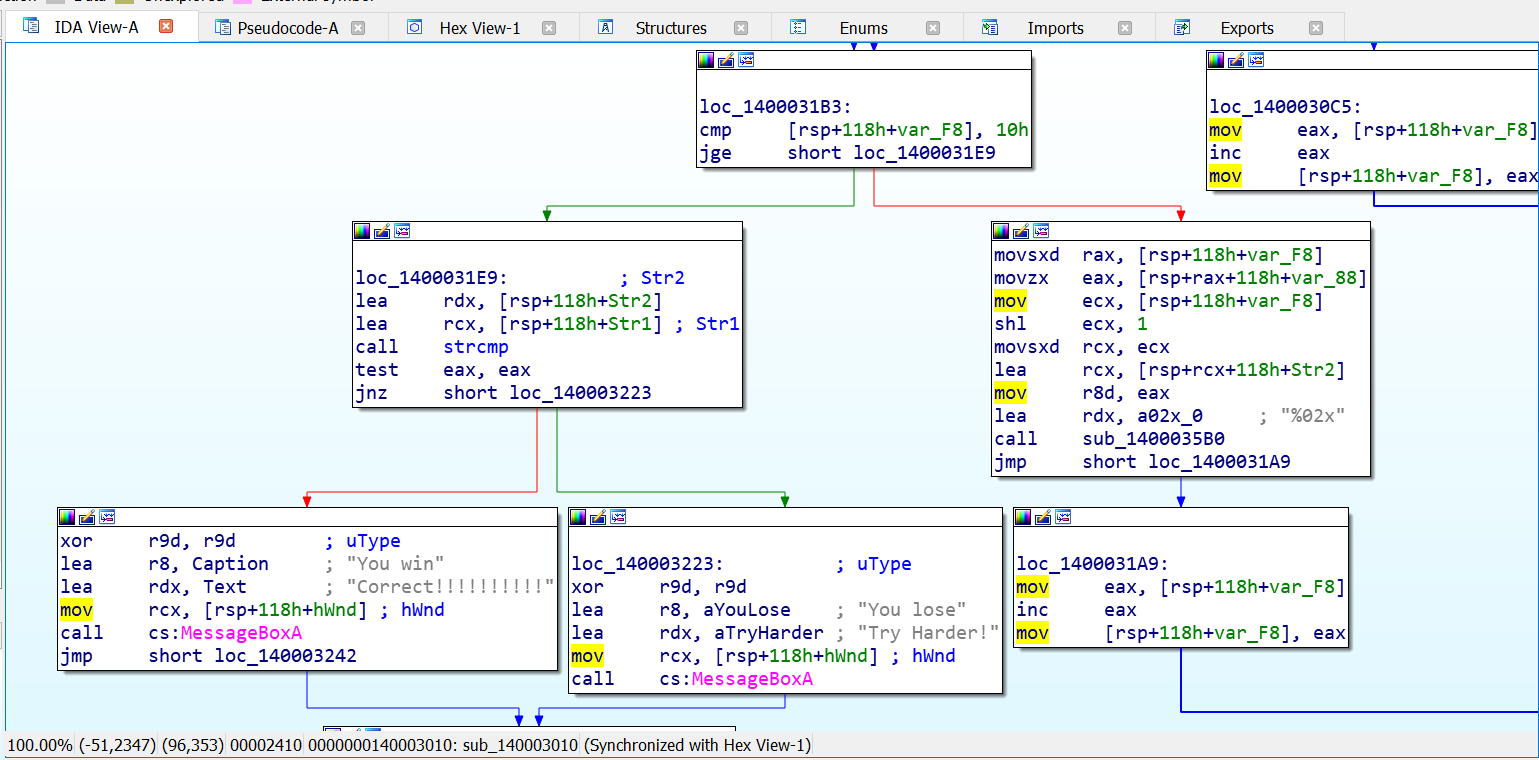


The function ***sub\_140003010()*** looks like it checks the password against the input:

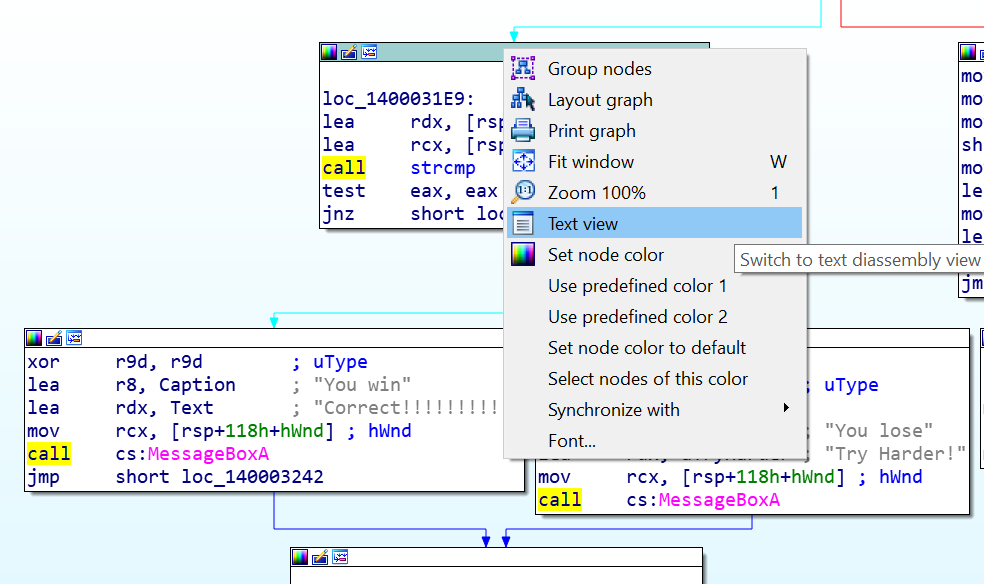


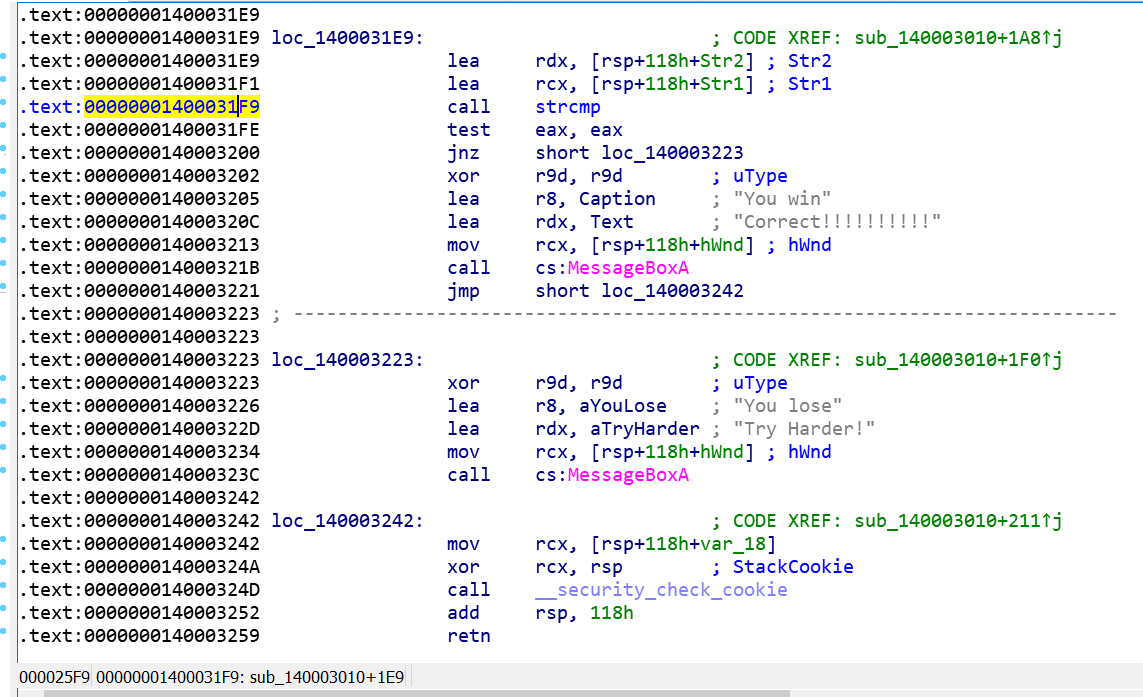
The final ***if*** condition that verifies whether the password is good or not can be noticed at line 64. ***MessageBoxA*** is the equivalent function for printf (we have a graphical interface).

* *Open the binary in x64dbg and set a breakpoint at the function call in the* ***if*** *condition*:
  + *Note that after starting, x64dbg will set some standard breakpoints which you probably do not need. Note the state of the program (Paused/Running) in the lower-left corner;*
  + *Also note that on Windows, the calling convention is different; see the call window on the right;*
  + *To do this, copy the address from IDA and navigate to it in x64dbg after the program has started. See* ***x64dbg basic commands*** *above;*
  + *Identify which parameter is the result from user input and what it is compared against. (1p)*

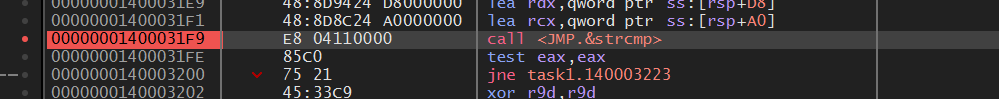


We are interested in the ***if*** that contains ***strcmp*** and breaks the program in the 2 cases:

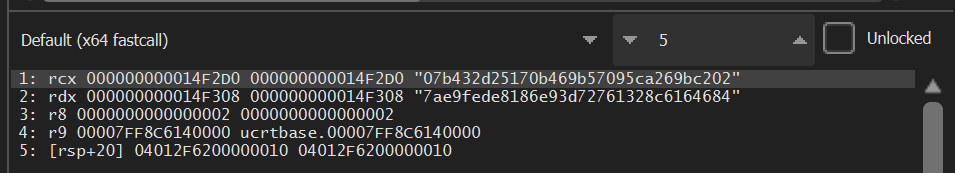
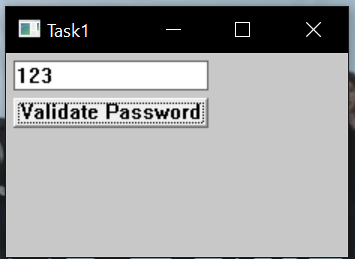




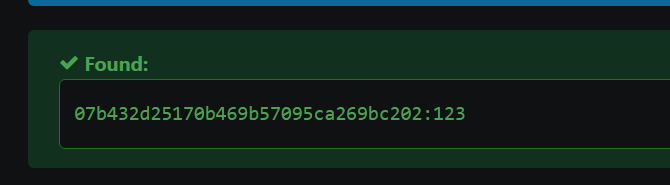
Setting the breakpoint in ***x64dbg*** using the adress from above (...***31F9***):



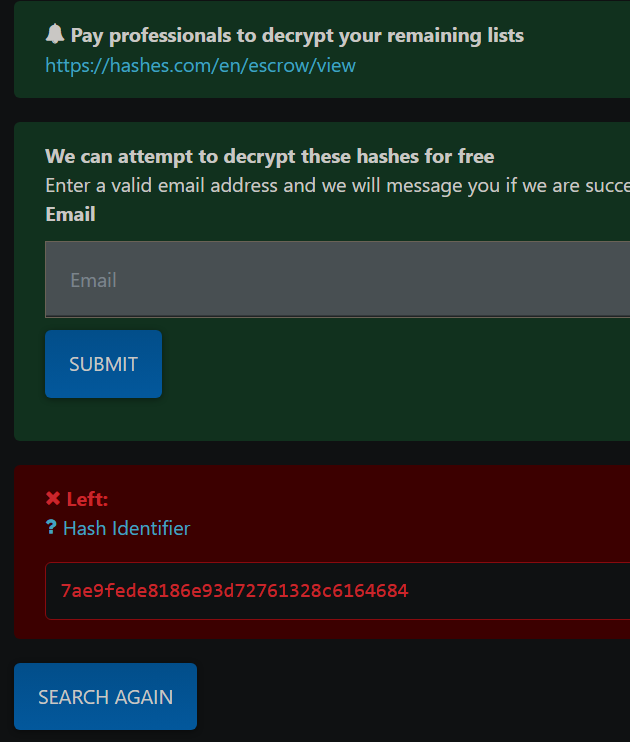
We run the program and input a password...

... and this is what we get:

It looks like we have some hash values. Let’s test this theory, using this [site](https://hashes.com/en/decrypt/hash) (decrypt hash):

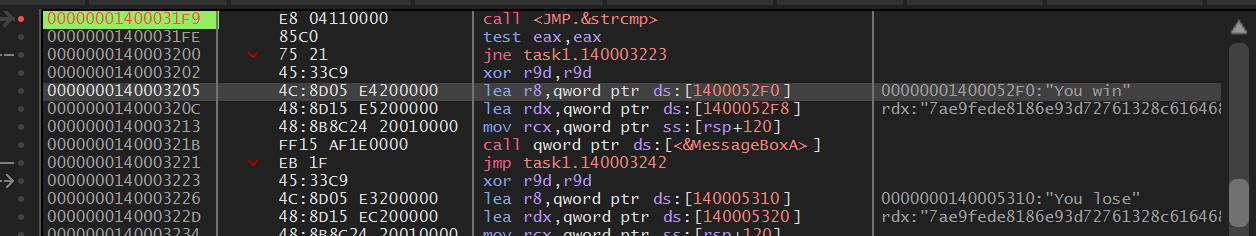


So we get the password that we inputed, meaning that ***rcx*** is the ***value from the user*** and ***rdx*** is the ***true password***. Let’s find the real password:

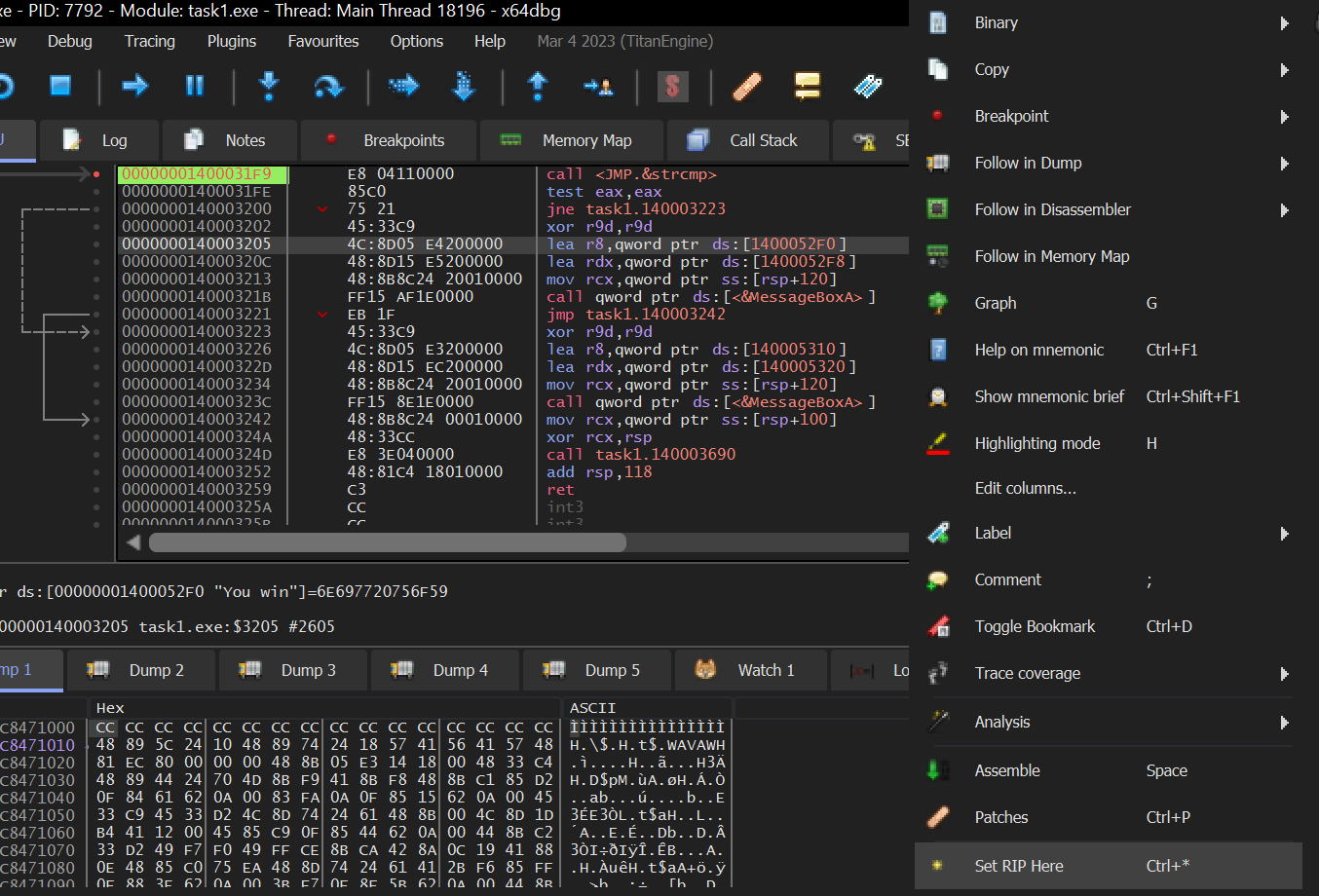


Eh, I tried =))).

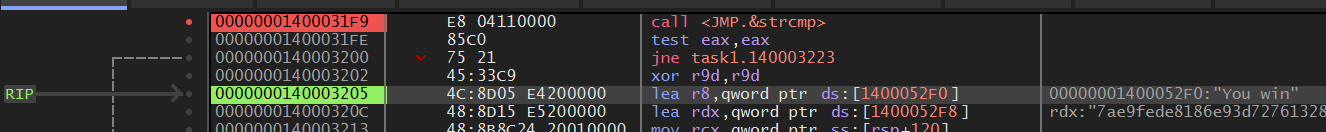
* *Using* ***Set New Origin here*** *or by modifying the corresponding CPU flag manually, make the program branch into the “Correct password” part; (2p)*

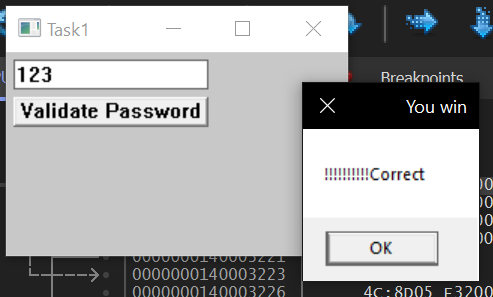


We will jump at the *...3205* instruction:



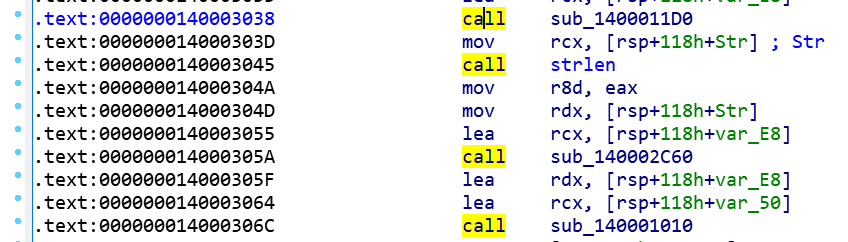
Result:

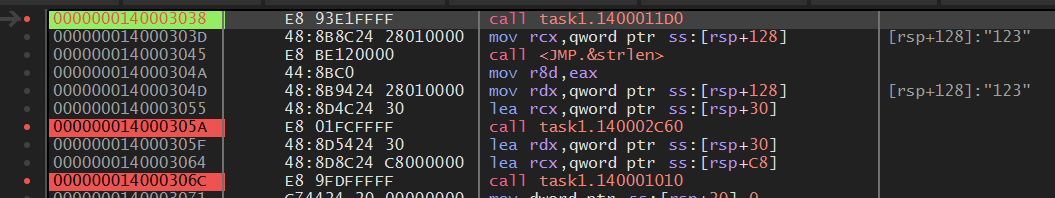




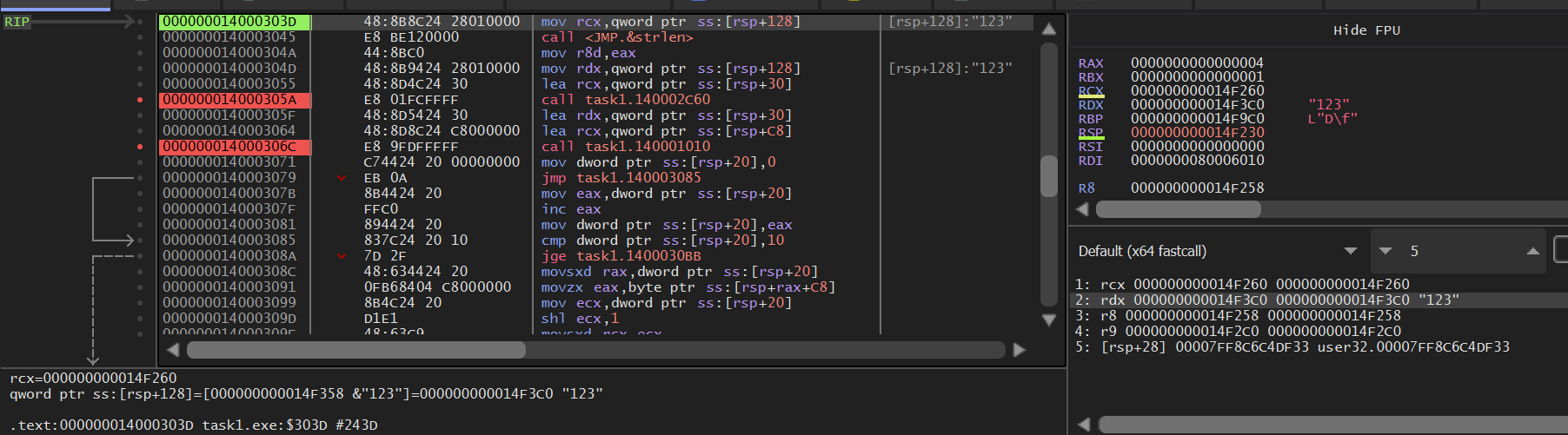
* *Find out what the three restricted functions mentioned above do by treating them as a black box. Use dynamic analysis and: (2p)*
  + *enter “password” into the text field (we want to find out what the transformation does by serving it a common input);*
  + *getting the function output from the debugger;*
  + *searching on the Internet for that hex string.*

The 3 restricted functions are: ***sub\_1400011D0, sub\_140002C60, sub\_140001010***:

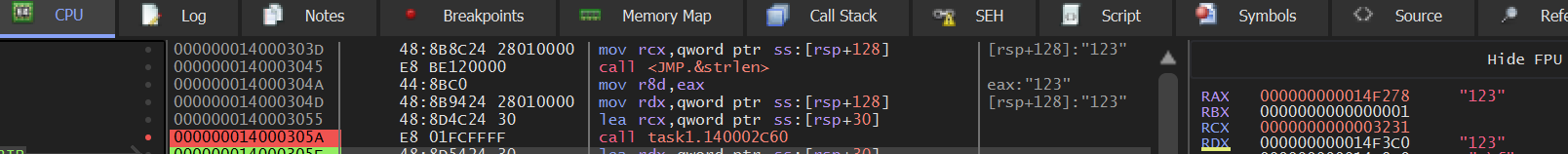




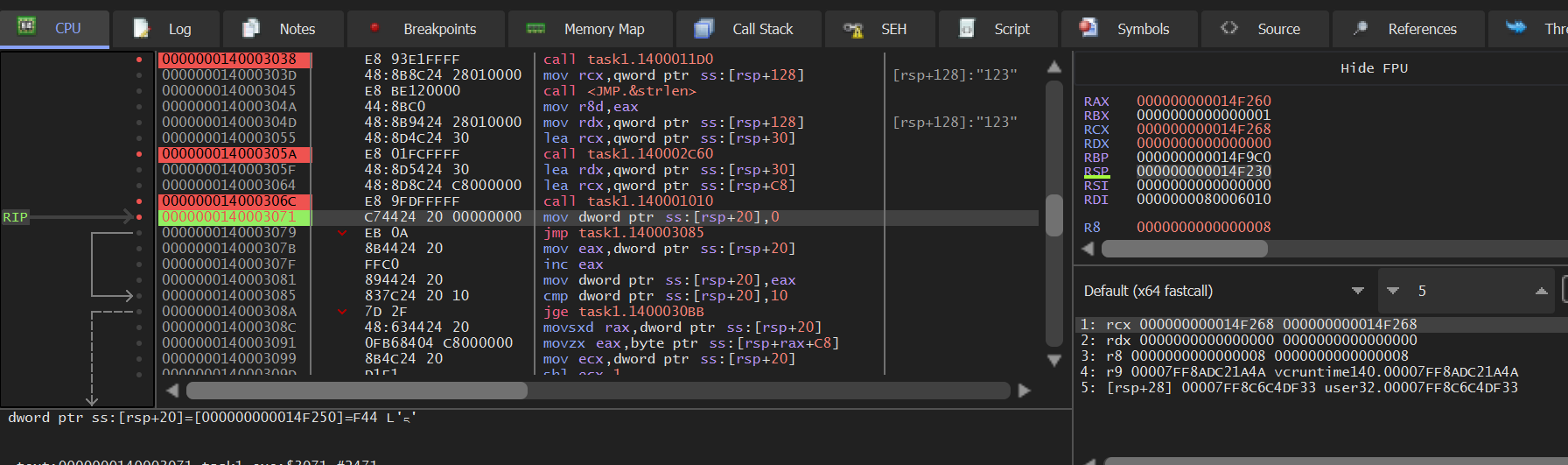
At the end of the first function, ***rax*** = 4:



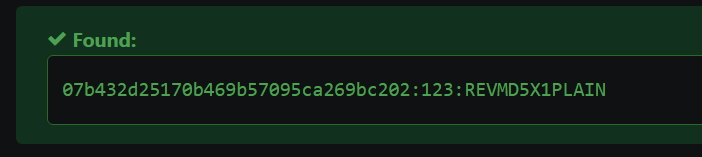
The second function returns the inputed password:



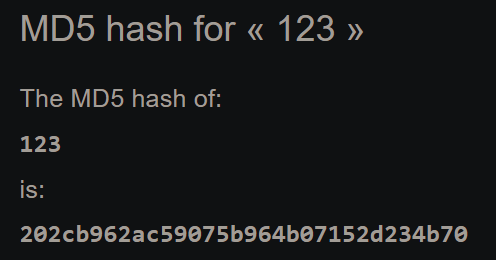
For the third function:



We got *07b432d25170b469b57095ca269bc202* after we inputed *123* as password. Using the [site](https://hashes.com/en/decrypt/hash) from above, we can figure out the algorithm:



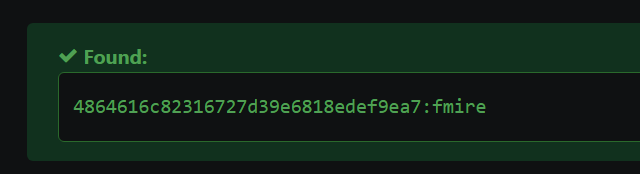
So we have a reverse MD5 ([site](https://md5.gromweb.com/?string=123)):



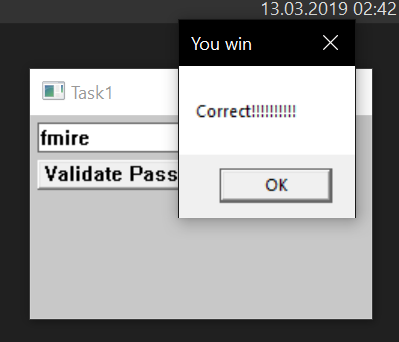
As we can see, the program reverses the MD5 hash created from the original password.

* *Figure out the correct input. (2p)*

The hash that is compared our input to is: *7ae9fede8186e93d72761328c6164684*. We reverse it [here](https://string-functions.com/reverse.aspx): *4864616c82316727d39e6818edef9ea7*. And using the site from above that decrypts hashes:



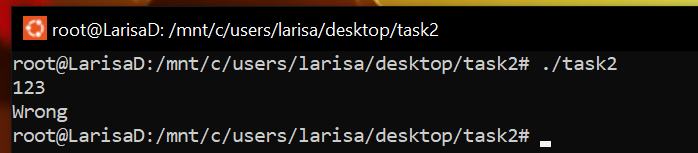
Let’s test it (running the .exe):



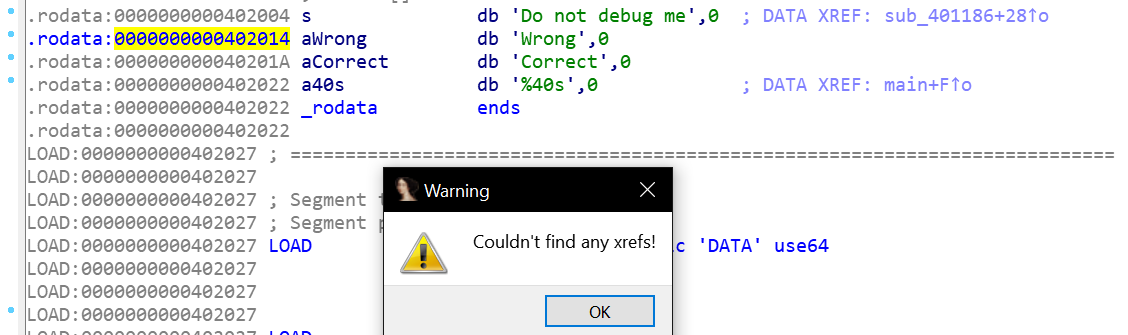
**4.2. Task: debugging in Linux**

Before solving:

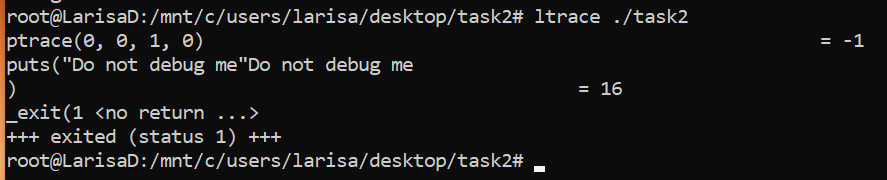
* *Run the program once, feed it a random input and take note of any strings for later use;*

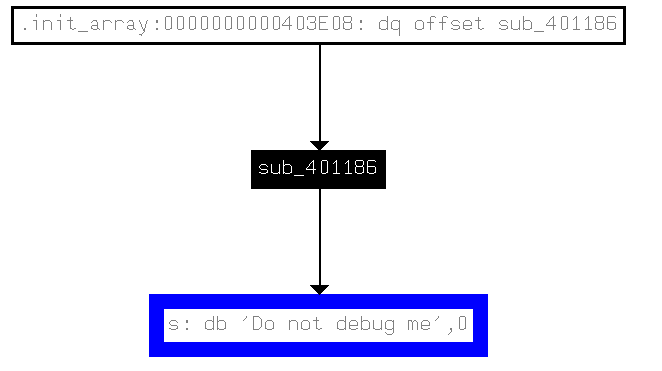


* *Try the approach in lab session 0x03 by looking for xrefs to the strings. What do you observe? What might be the cause?*

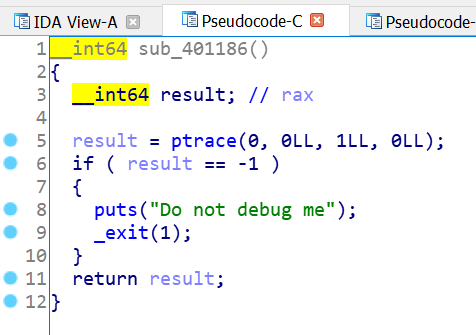


* *Now try the approach in lab session 0x01 by running with ltrace. Does it work?*





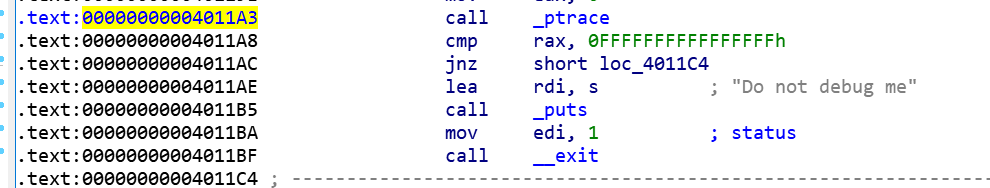
The function checks if the program is runned with ***ltrace***:



We need a different approach:

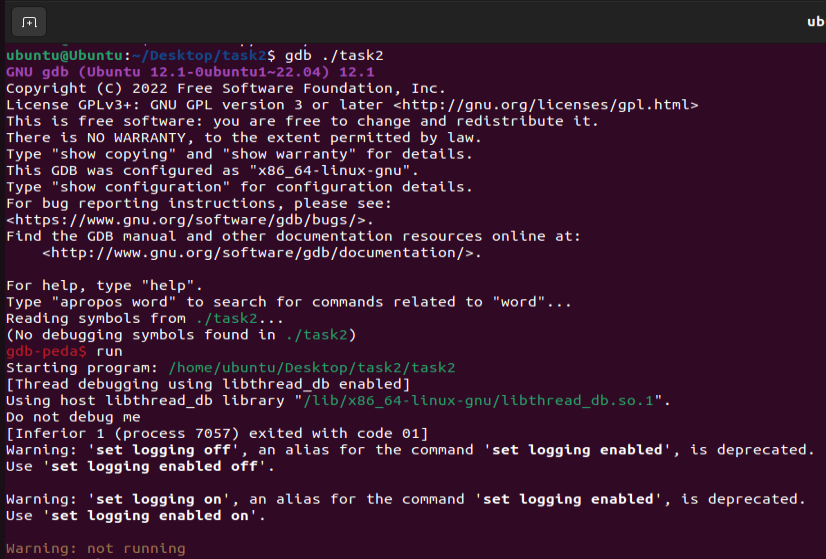
* *Find the anti-debugging mechanism by searching for the* ***ptrace call*** *in IDA. Notice the condition for program exit;*

We have the screenshot from above and here are the addresses:

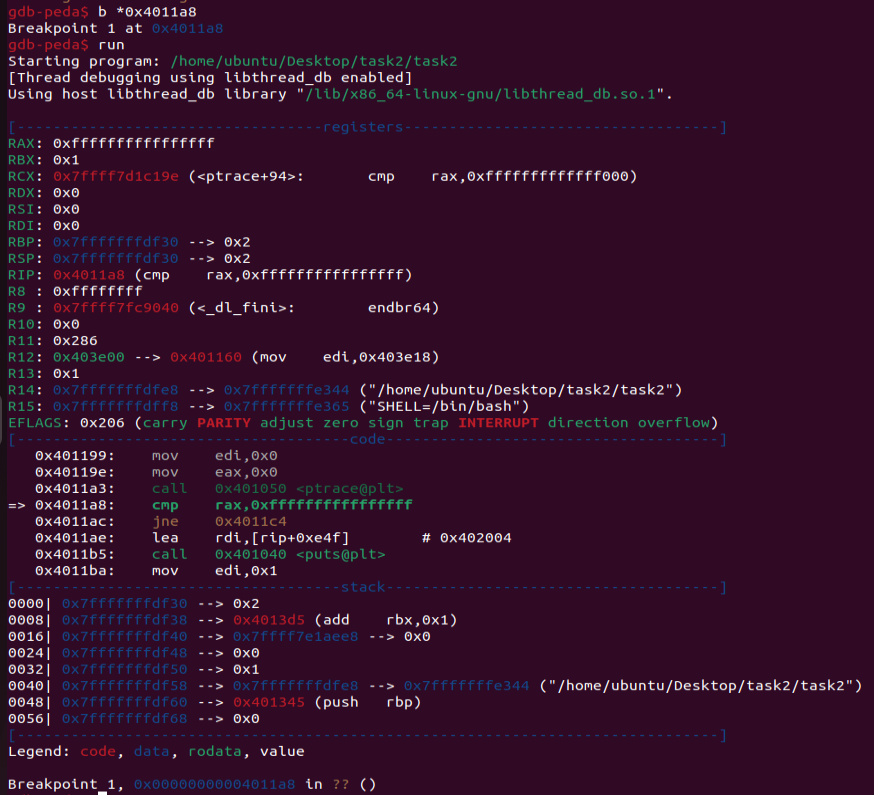


* *Then, in gdb/peda, set a* ***breakpoint*** *on the address right after the call;*

We start the program:



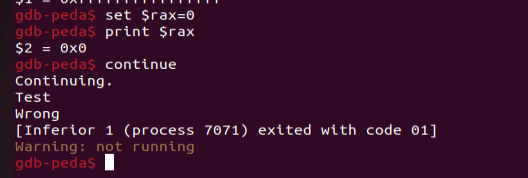
We set the breakpoint at the address ***\*0x4011a8***:



* *When the debugger stops there, modify the corresponding register such that when continuing execution under the debugger, the program does not exit. (2p)*

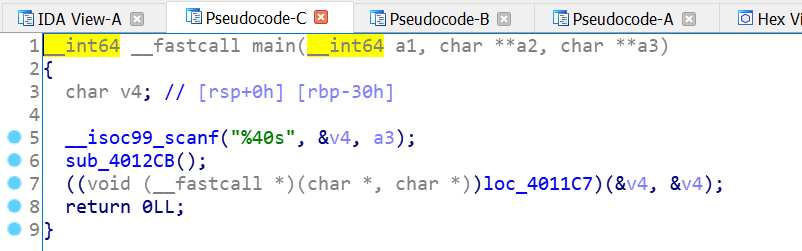


Set ***$rax*** to something different in order to continue the execution:



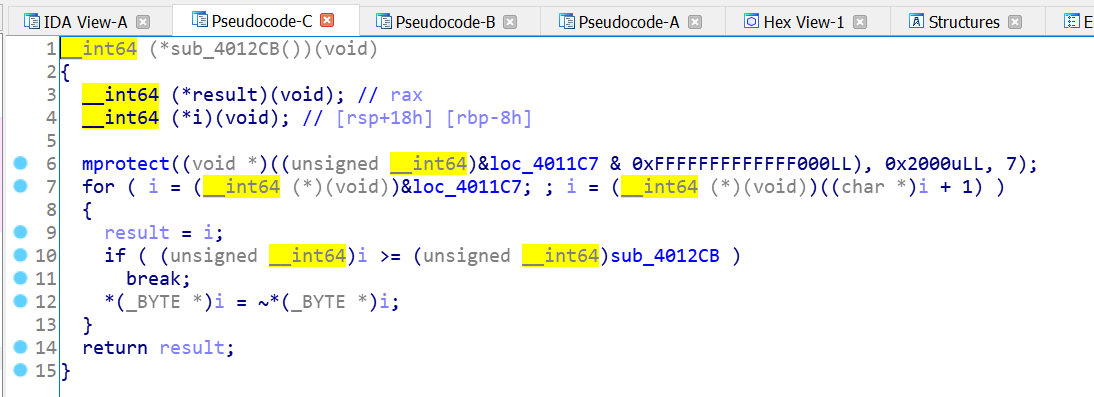
*You have successfully bypassed the anti-debugging mechanism! Continue the analysis:*

* *Using IDA, analyze the main function:*
  + ***scanf*** *gets the user input;*
  + *The third function is called with the user input as its parameter but going into it we see it is just garbage code, impossible to analyze in its current state;*
  + *The second function actually decrypts the code for that function.*



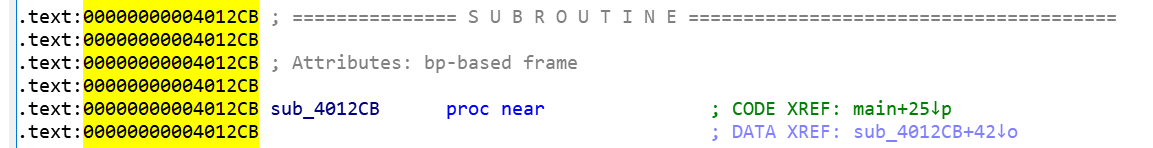
* *Go into the decryption function and pay attention to the* ***for loop****. Determine the start address and the end address for the decryption process;*

The ***sub\_4012CB()*** function:



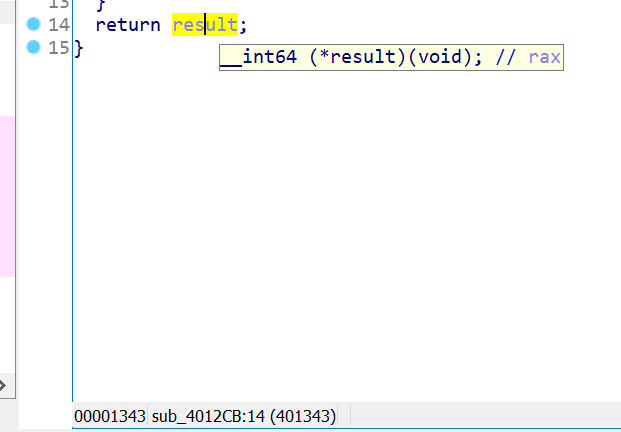
As we can see in the screenshot from above, the start address is the starting address of the function ***loc\_4011C7*** (offset: ***4011C7***; picture below) and the end address for the decryption is ***4012CB*** (because of the if-break function – ***sub\_4012CB***; picture below):



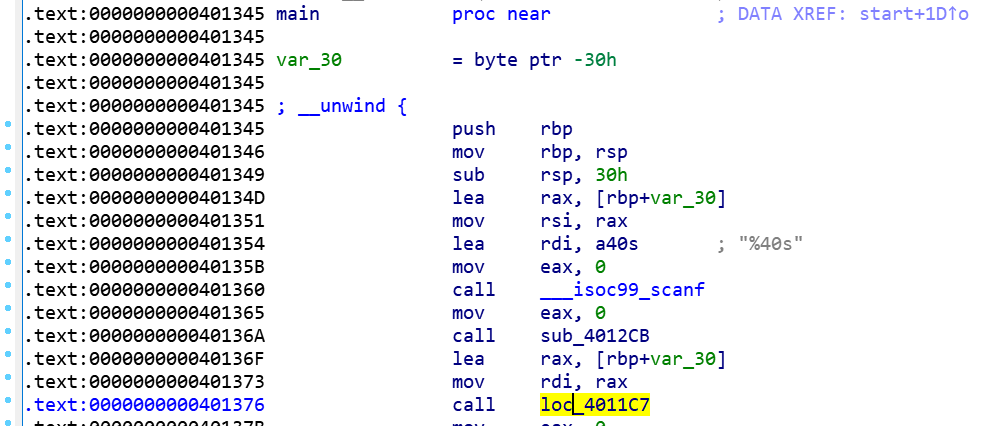


* *Then, in gdb, set a* ***breakpoint*** *after the decryption finishes (right before the decrypted function is called) and* ***dump*** *the decrypted memory. (2p)*

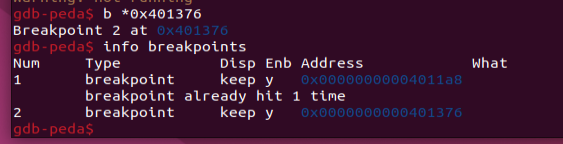
We need to dump the memory between the 2 memory addresses from above: ***0x4011C7*** – ***0x4012CB***. If we click the ***result*** (is highlighted if selected), we can see the address in memory:



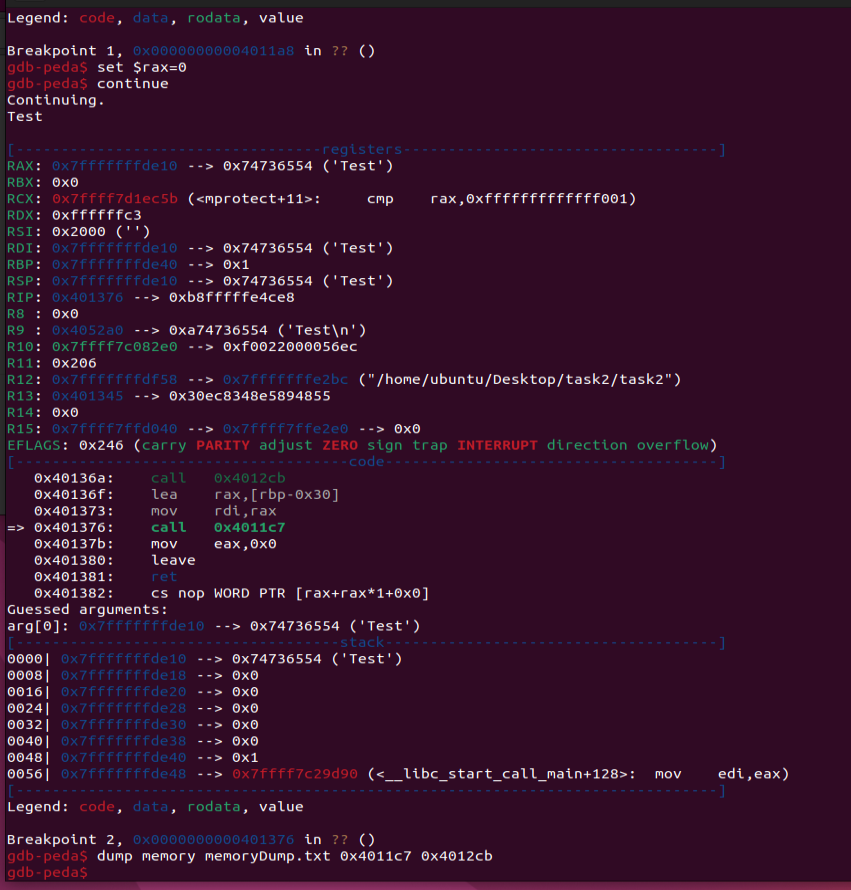
The address is: ***0x401343***. And the address right before the decrypted function is called is: ***0x401376***.



So, let’s set the breakpoint:



And let’s dump the memory:



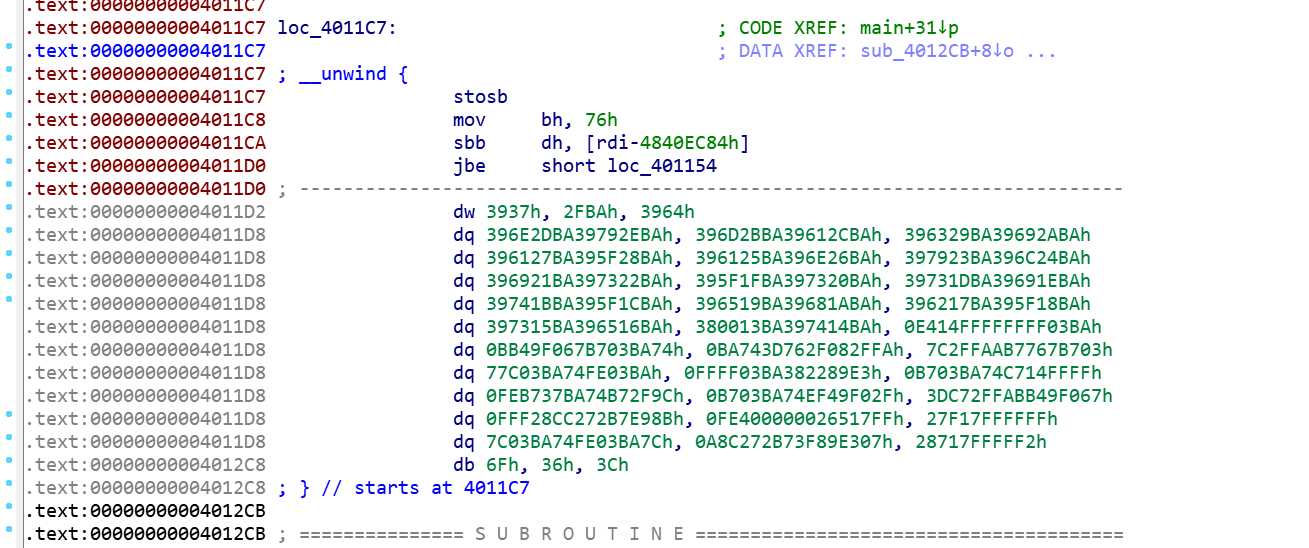
The memory dump is in: ***memoryDump.out***.

*You now have the third function decrypted, but in binary form. For the following, if you do not have IDAPython (e.g., IDA Trial), use this IDC guide.*

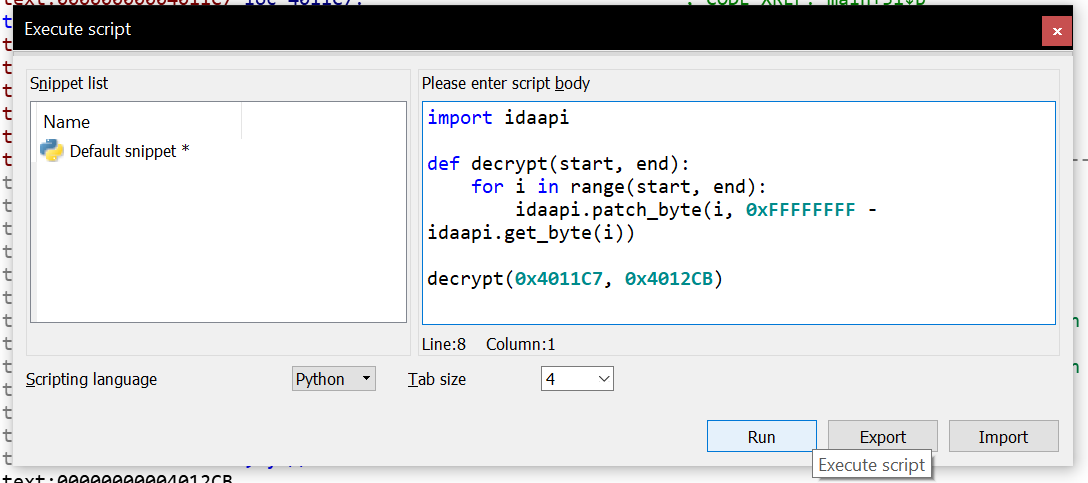
* *Using get byte and patch byte3 in the Python scripting interface (File→Script Command with Scripting Language set to Python), decrypt the bytes of the function. You can either use:*
  + *Only* ***patch byte*** *with the contents of the dumped memory;*
  + *Or* ***get byte****, replicate the decryption and then* ***patch byte****.*
* *The end result should be fully decompiled. (3p)*

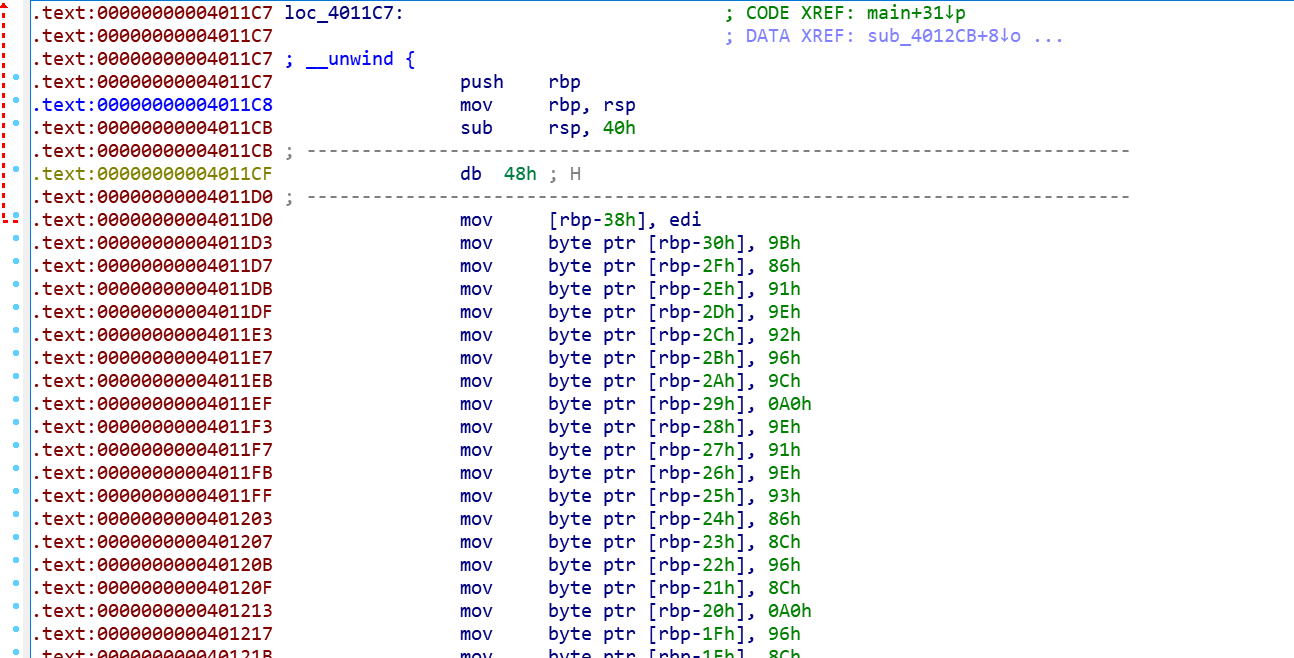
I’ve created a Python script that can decrypt the bytes (it can be found in *scriptDecrypt.py*).

Before running the script:

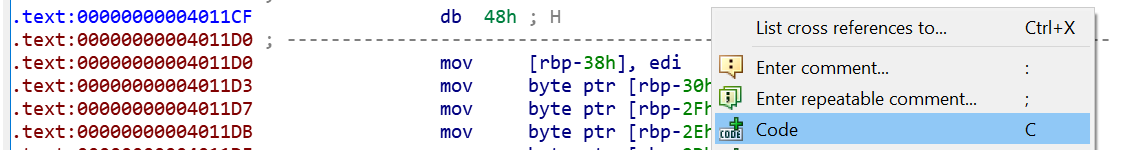


After running the script (*File → Script Command* → select *Python* from the *Scripting language* dropdown → paste the script and hit *Run*):

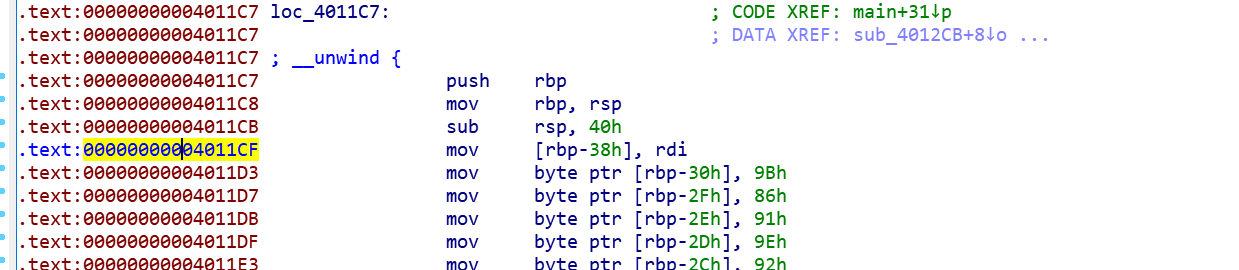




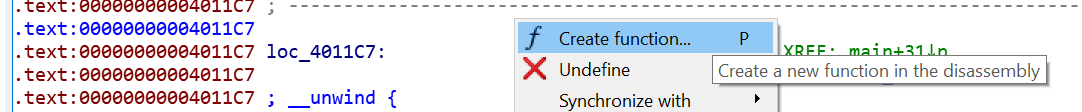
Now, we need to convert from variable (*loc*) to function (*sub*). But first, we need to convert line ***0x4011CF*** into code, so we press ***C*** (*right click + Code*):



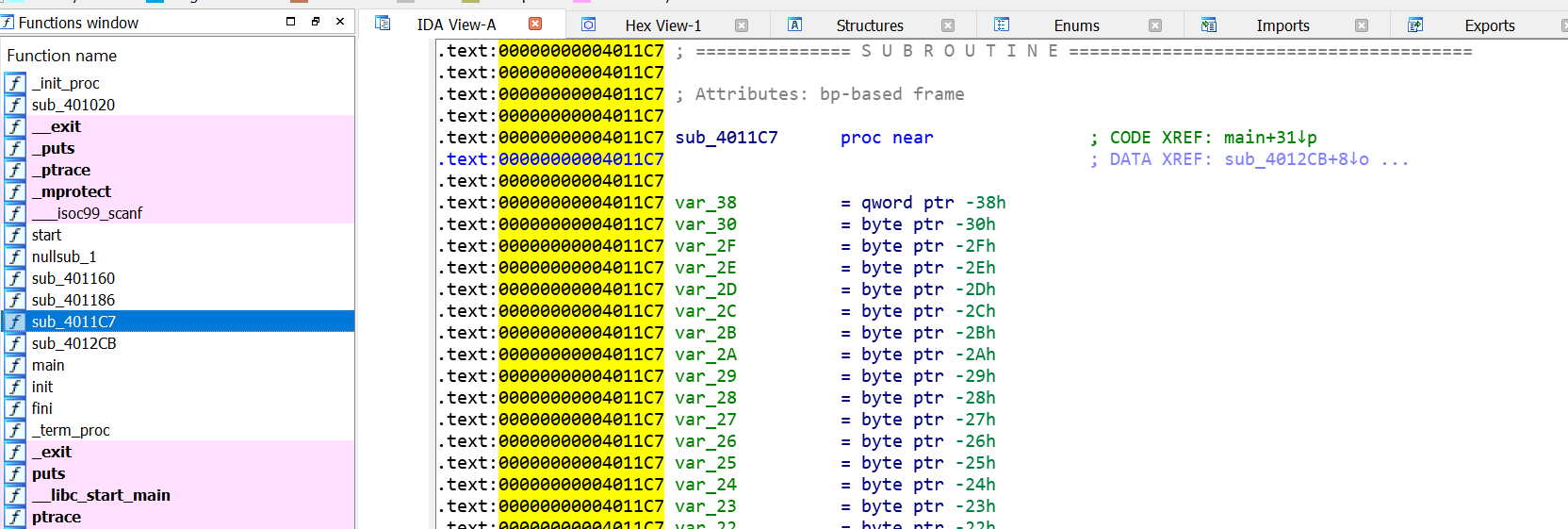
Result:



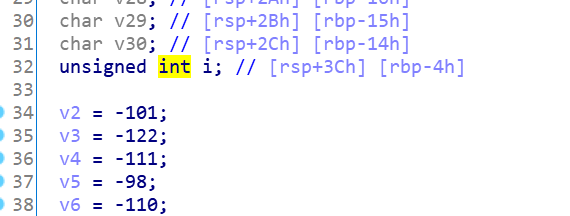
And now, we can create the function by pressing ***P*** (*right click + Create function*):



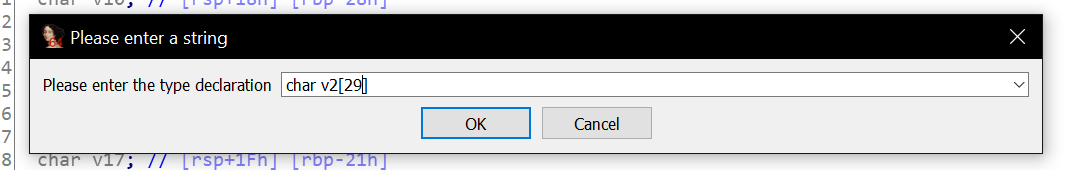
Result (we can notice that IDA even named the function and put it in the left menu, along side the others):



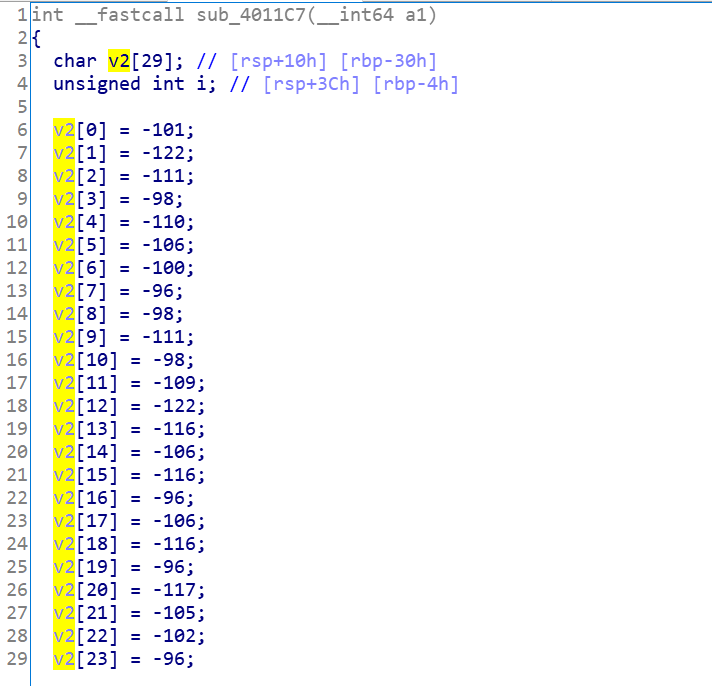
By pressing ***F5***, we can see inside the function:

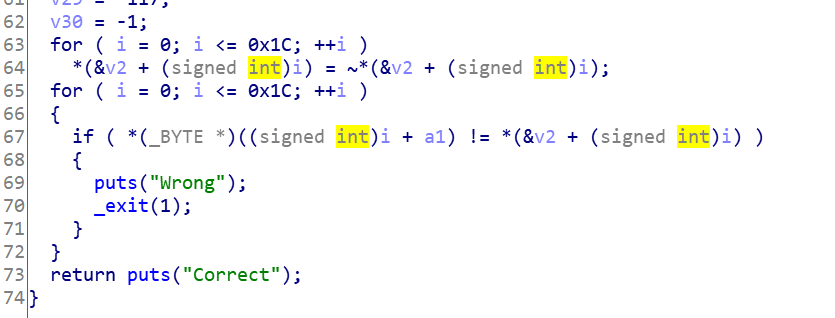


We change the data type of v2:



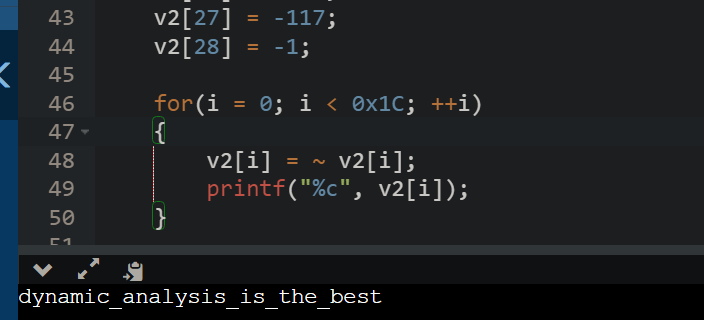
Result:





The IDA file: *task2.i64*

We take this code and write a C program (*find\_pass.c*) in order to find the password:



We can notice from the screenshot the password: ***dynamic\_analysis\_is\_the\_best***

Let’s test it:

