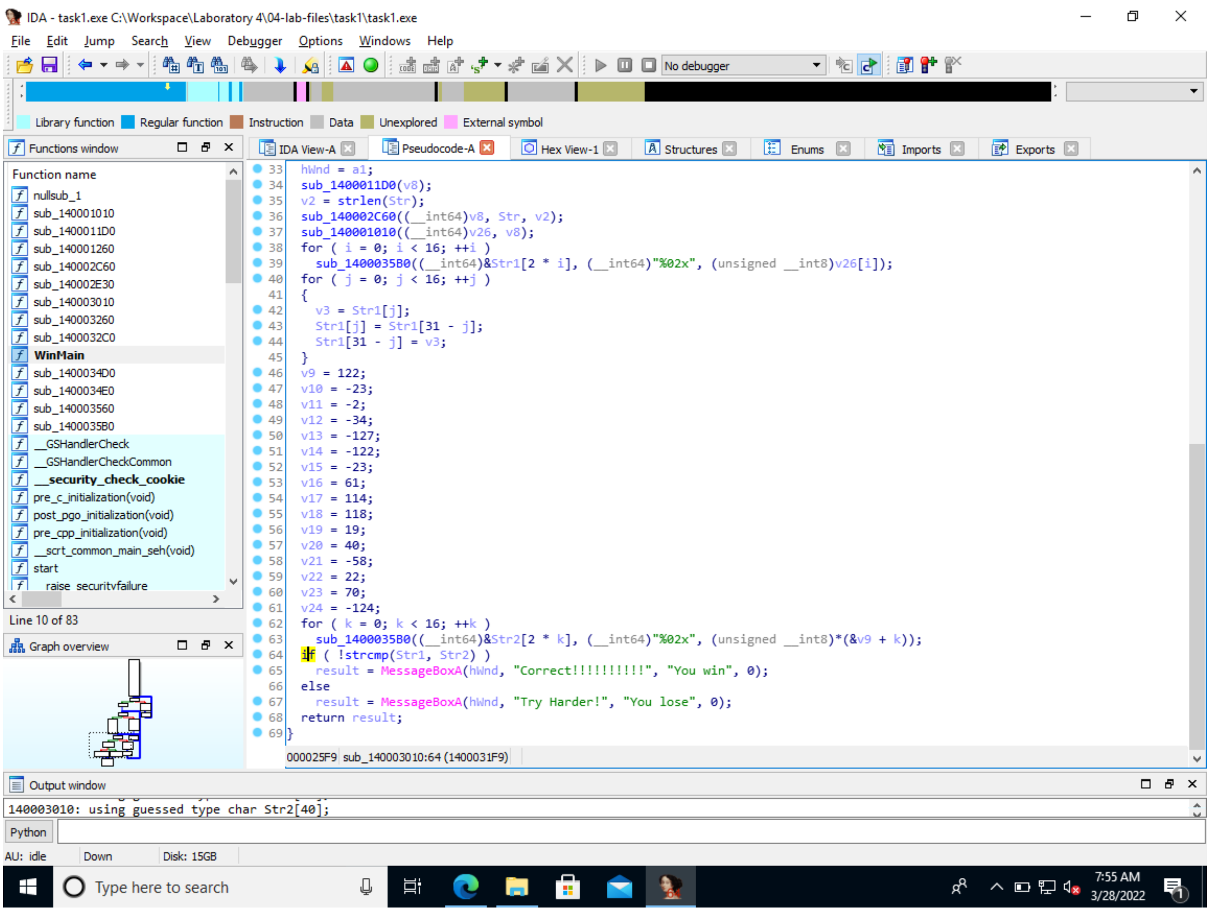
Task 1: Windows dynamic analysis

- **Open the binary in IDA and identify the password checking function (same procedure as in lab 03) and the final if condition that verifies whether the password is good or not. Also, figure out which function is sprintf . (2p)**

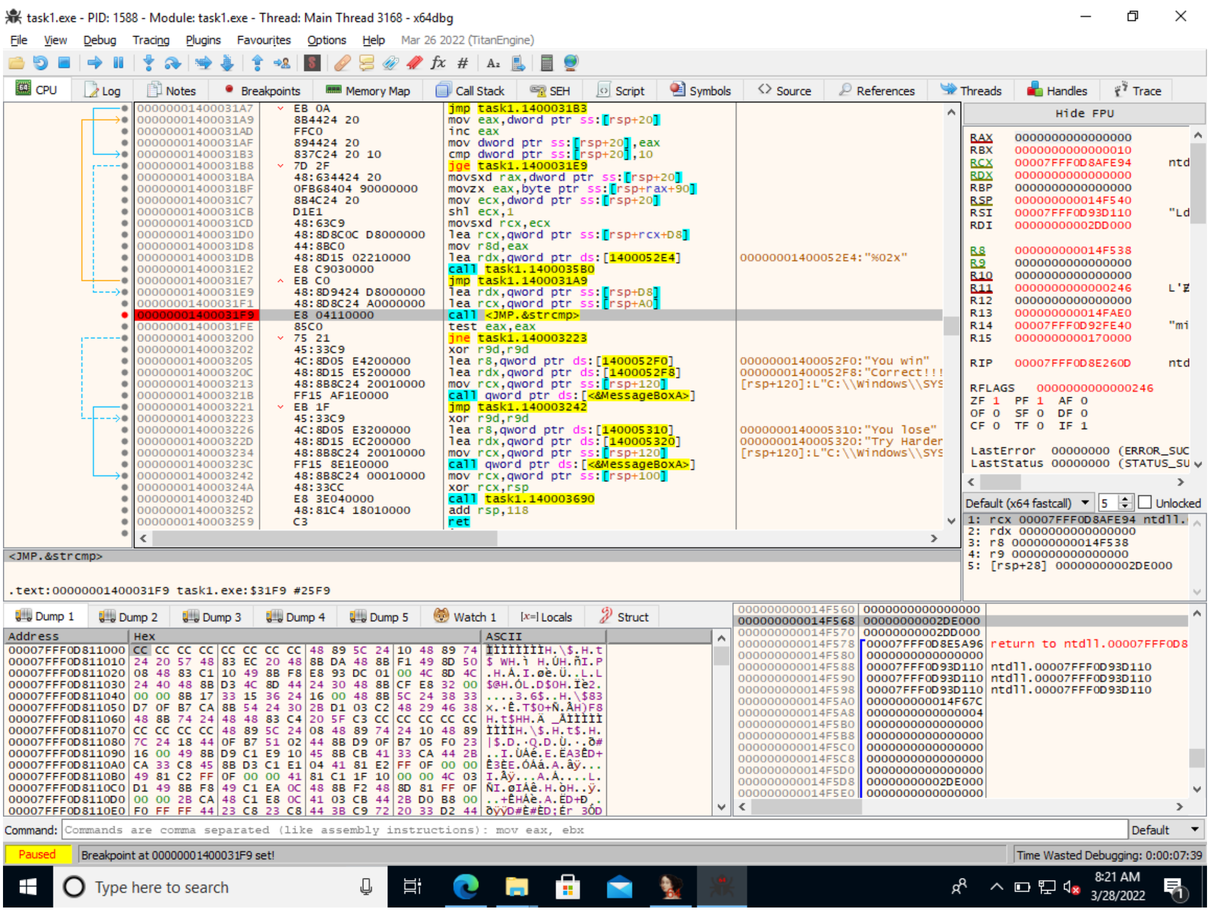
I looked through the “decompiled” code from the IDA starting with the WinMain function. In there, I have found the function “sub_140002E30” which is the one that handles all actions from the window of the application. In that function, I identified the function “sub_140003010” which is the one that checks the password. As seen in the screenshot below, I found the last “if” statement at the offset 1400031F9 and the sprintf function as “sub_1400035B0”.



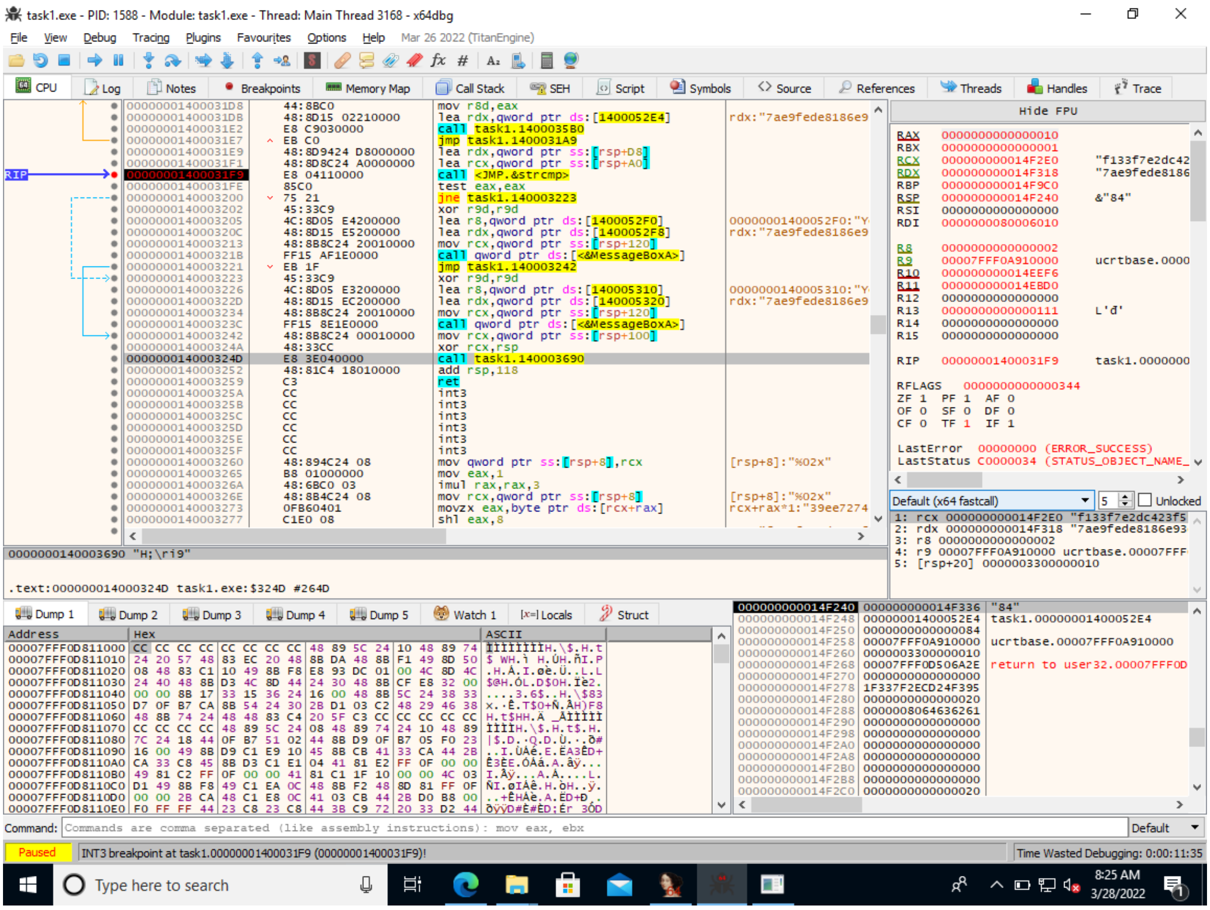
- **Open the binary in x64dbg and set a breakpoint at the function call in the if condition.**
  - **(Note that after starting, x64dbg will set some standard breakpoints which you probably do not need. Note the state of the program (Paused/Running) in the lower-left corner)**
  - **(Also note that on 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.**

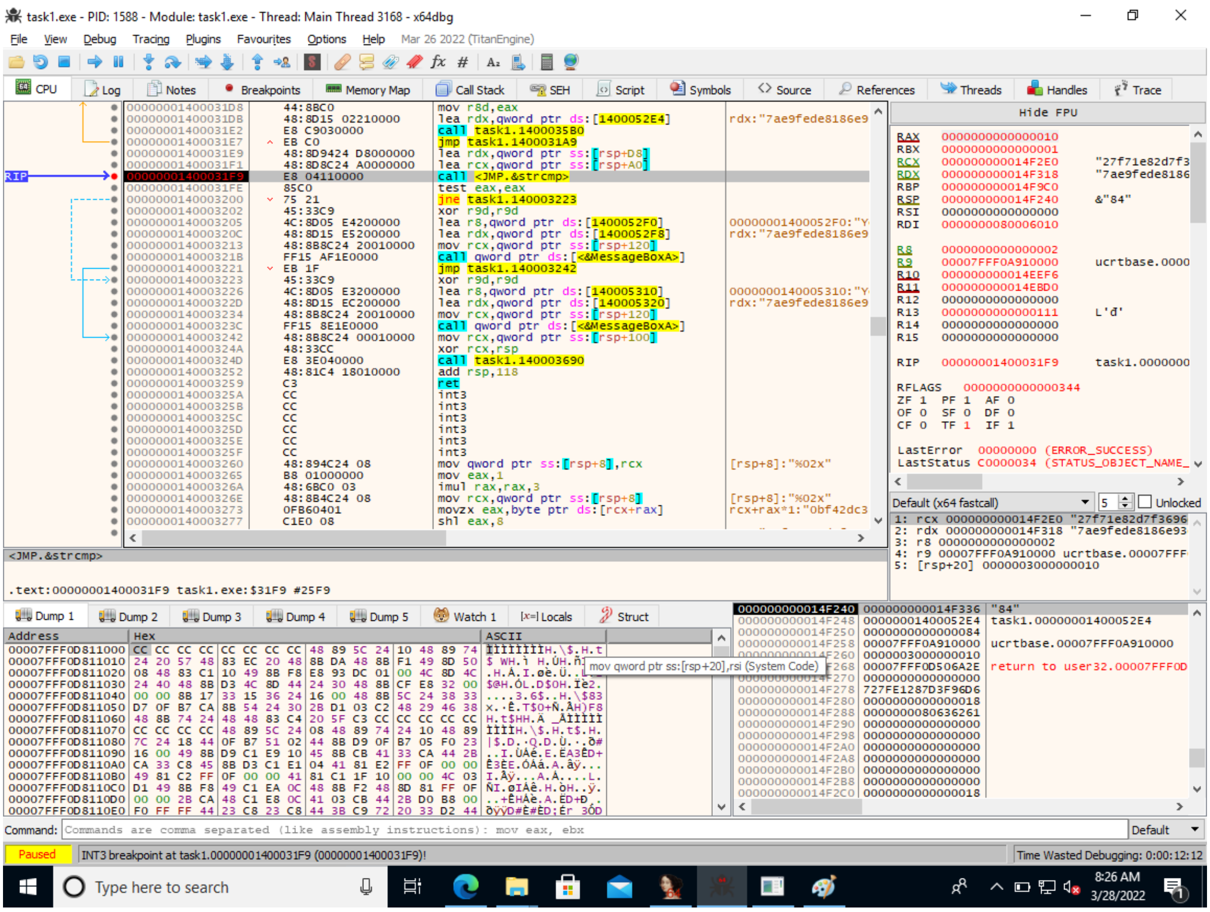
- **Identify which parameter is the result from user input and what it is compared against.(1p)**

In the following image, it can be seen that a breakpoint has been set at the instruction which calls "strcmp".



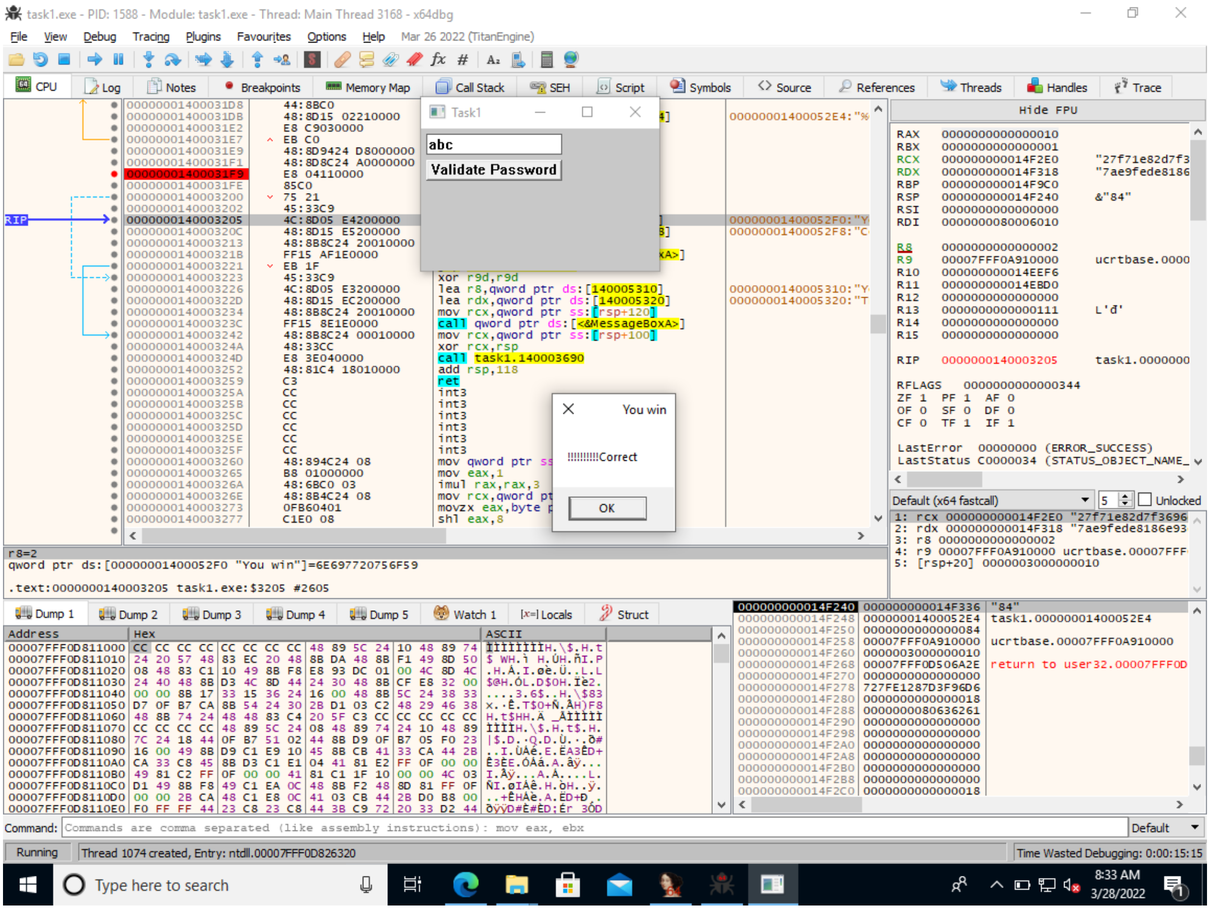
After that I ran the application two times until the breakpoint and I tried different inputs as illustrated in the following screenshots. Looking at the instructions before the call, we can see that the registers rdx and rcx contain the input and the string to which it is compared. Comparing the difference between the two screenshots, we can conclude that the value from rcx corresponds to the input (it changes when feeding different inputs) and rdx to the compared string (remains constant for different inputs).


task1.exe - PID: 1588 - Module: task1.exe - Thread: Main Thread 3168 - x64dbg
.text:000000014000324D task1.exe:$324D #264D
Paused
INT3 breakpoint at task1.00000001400031F9 (00000001400031F9)!
Time Wasted Debugging: 0:00:11:35


task1.exe - PID: 1588 - Module: task1.exe - Thread: Main Thread 3168 - x64dbg
.text:00000001400031F9 task1.exe:$31F9 #25F9
Paused
INT3 breakpoint at task1.00000001400031F9 (00000001400031F9)!
Time Wasted Debugging: 0:00:12:12

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

Using the “Set New Origin here” functionality, I moved into the branch corresponding to the correct password and as it can be seen in the image, the message for the correct password was printed.

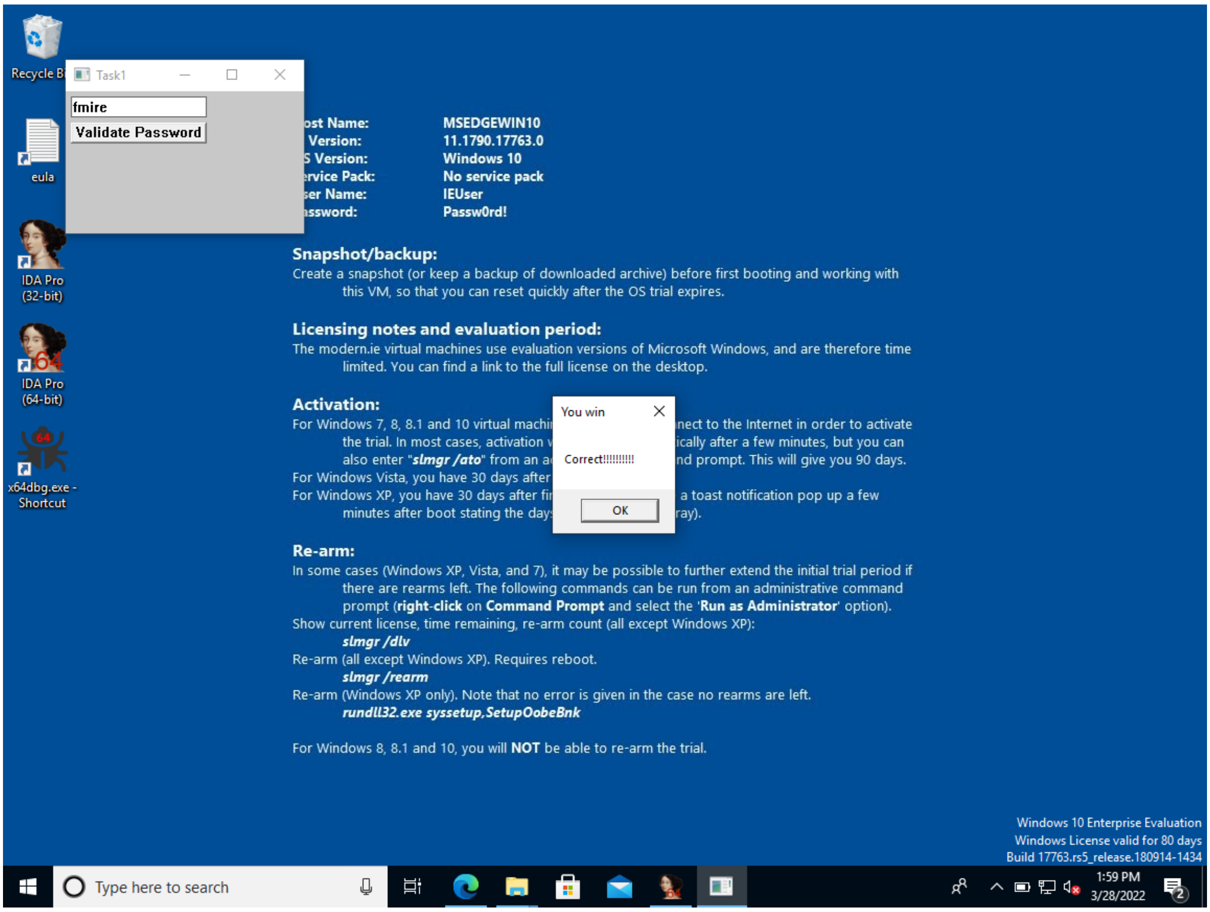


- **Find out what the three restricted functions mentioned above do by treating them as a black box. Use dynamic analysis and: (2p)**

I gave as input the value “password” and I observed the value “99fc288bed7238d16d567aa5b3ccd4f5” for the register rcx. Searching on the internet, I found out that it is the mirror of the value obtained by computing the hash function MD5 on the input. The following link contains the webpage that I have used: https://md5hashing.net/hash/md5/5f4dcc3b5aa765d61d8327deb882cf99.

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

To obtain the correct password, we need to apply the reverse steps described previously on the value that we observed in the register rdx (7ae9fede8186e93d72761328c6164684). We can revert the string and use a program that “un-hash” (such as https://crackstation.net/) which gives us the solution: “fmire”. In the following screenshot, it can be seen that introducing this input, we receive the “Correct” message without using the debugger.

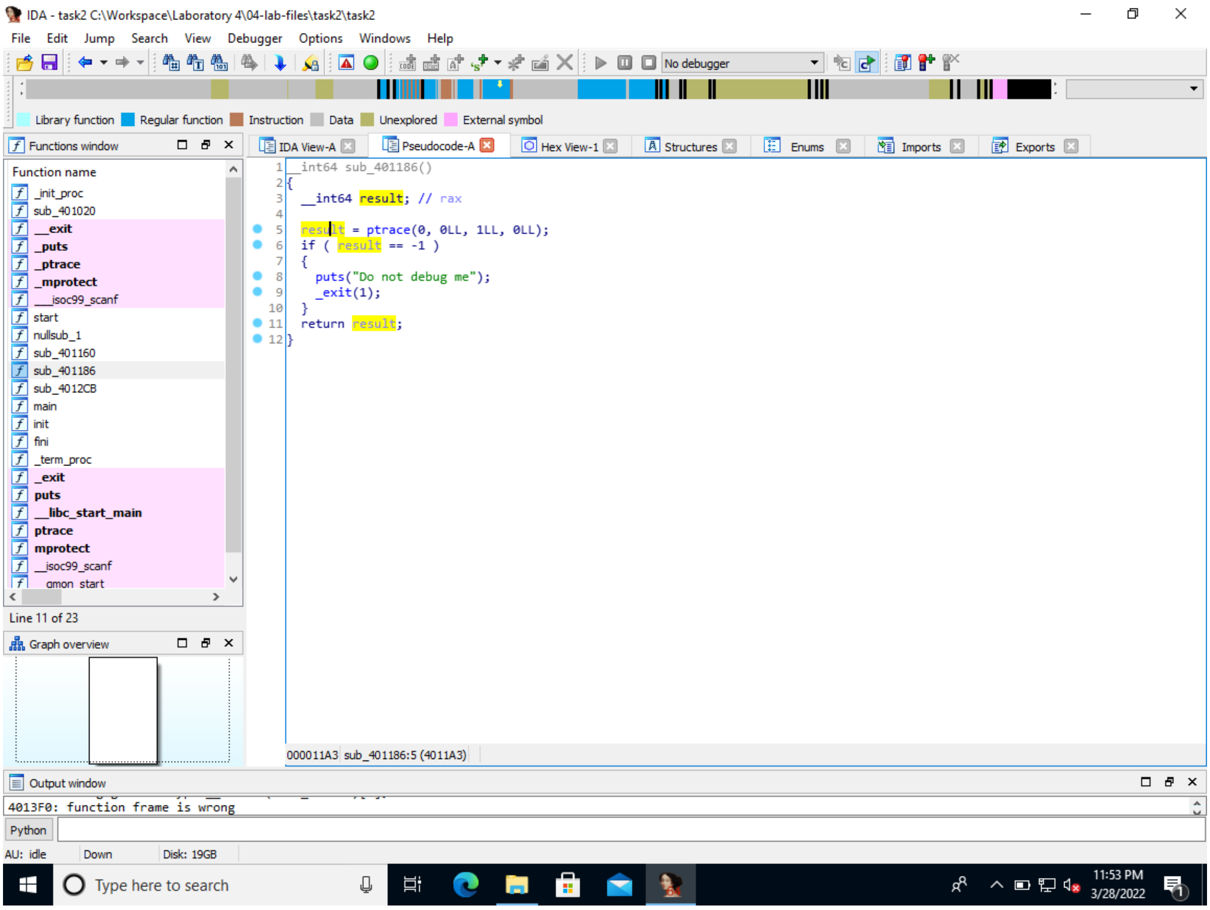


Task 2: Linux dynamic analysis

We need a different approach:

- **Find the anti-debugging mechanism by searching for the ptrace call in IDA. Notice the condition for program exit.**
- **Then, in gdb/peda, set a breakpoint on the address right after the call.**
- **When the debugger stops there, modify the corresponding register such that when continuing execution under the debugger, the program does not exit. (2p)**

When using the first approach (looking for the strings), I identified a string “Wrong” and found it in the “.rodata” segment, but xref did not offer any reference. Looking at the main function, I believe the cause is that the function (which uses the string) is generated at runtime and called through a function pointer. The approach with “ltrace” didn’t work either, because the program behaves differently when wrapped in “ltrace” and a “Do not debug” message is printed.
Beside the “Wrong” string in the “.rodata” segment, I also found the string “Do not debug me!” on which we can apply xref and find the function “sub_401186” which makes the check if the program is runned under “ltrace” or not. We can see in the following screenshot the offset for that instruction as 4011A3.



In the gdb, we set a breakpoint at the address specified before as seen in the screenshot below.

```
gdb-peda$ b *0x4011A3
Breakpoint 1 at 0x4011a3
gdb-peda$ info breakpoints
Num     Type           Disp Enb Address            What
1       breakpoint     keep y   0x00000000004011a3
gdb-peda$
```

After that we use the “run” command to execute the program until that breakpoint. And goes one more step with the “next” command, to skip the function call to ptrace. Afterwards we can change the value returned by ptrace_call from the register rax using the command “set $rax=0x0” (note: we can use any value different from -1 instead of 0x0). After these steps we reached the following context:

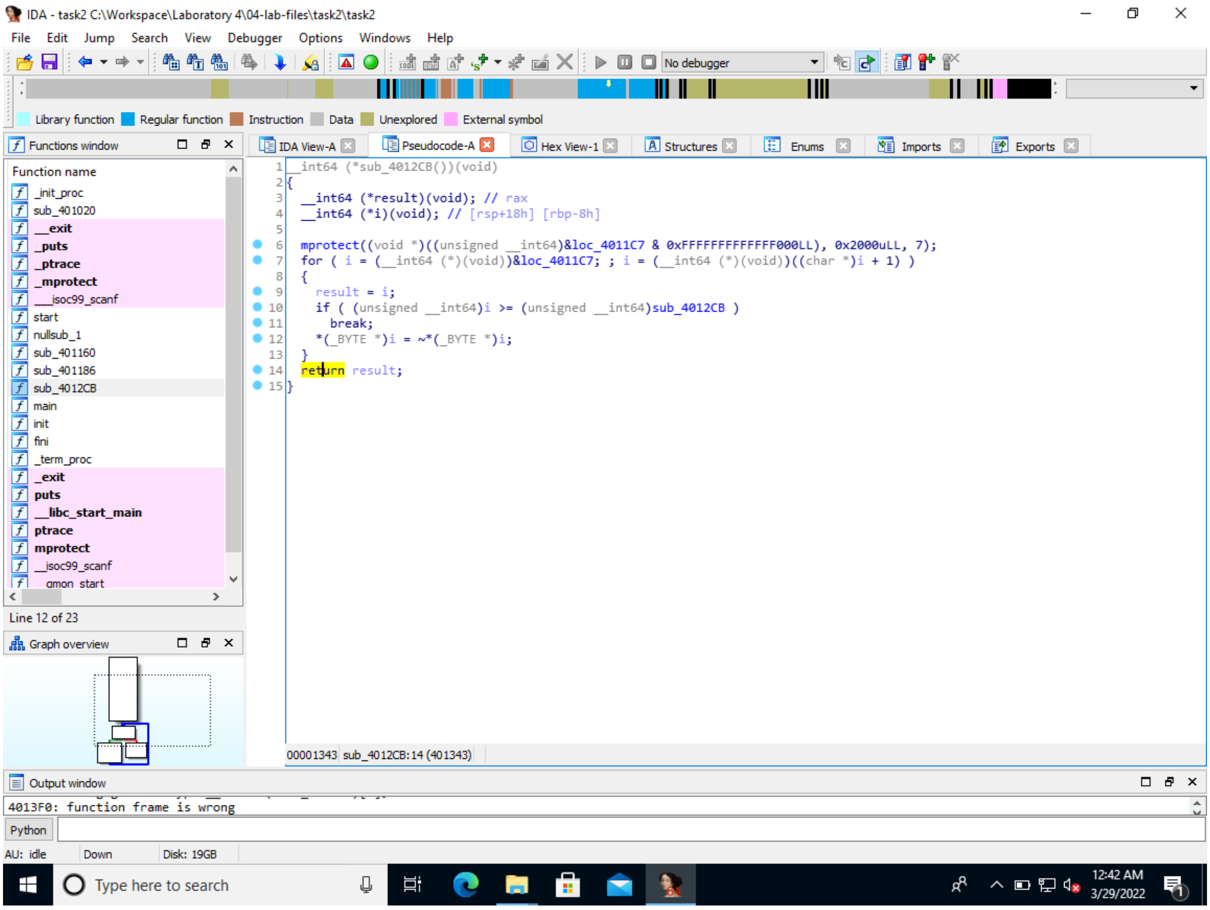
```
gdb-peda$ context
[----------------------------------registers-----------------------------------]
RAX: 0x0
RBX: 0x1
RCX: 0x0
RDX: 0xffffffffffffff80
RSI: 0x0
RDI: 0x0
RBP: 0x7fffffffde30 --> 0x2
RSP: 0x7fffffffde30 --> 0x2
RIP: 0x4011a8 (cmp    rax,0xffffffffffffffff)
R8 : 0xffffffff
R9 : 0x7ffff7fe0d50 (endbr64)
R10: 0x0
R11: 0x286
R12: 0x403e00 --> 0x401160 (mov    edi,0x403e18)
R13: 0x1
R14: 0x7fffffffdf68 --> 0x7fffffffe2cb ("/home/student/Workspace/lab4/task2/task2")
R15: 0x7fffffffdf78 --> 0x7fffffffe2f4 ("SHELL=/bin/bash")
EFLAGS: 0x206 (carry PARITY adjust zero sign trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x401199:    mov    edi,0x0
   0x40119e:    mov    eax,0x0
   0x4011a3:    call   0x401050 <ptrace@plt>
=> 0x4011a8:    cmp    rax,0xffffffffffffffff
   0x4011ac:    jne    0x4011c4
   0x4011ae:    lea    rdi,[rip+0xe4f]        # 0x402004
   0x4011b5:    call   0x401040 <puts@plt>
   0x4011ba:    mov    edi,0x1
[------------------------------------stack-------------------------------------]
0000| 0x7fffffffde30 --> 0x2
0008| 0x7fffffffde38 --> 0x4013d5 (add    rbx,0x1)
0016| 0x7fffffffde40 --> 0x7ffff7fb52e8 --> 0x0
0024| 0x7fffffffde48 --> 0x401390 (push   r15)
0032| 0x7fffffffde50 --> 0x0
0040| 0x7fffffffde58 --> 0x401080 (xor    ebp,ebp)
0048| 0x7fffffffde60 --> 0x7fffffffdf60 --> 0x1
0056| 0x7fffffffde68 --> 0x0
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value
```

From this state, we can use the “next” command twice and observe that the program jumps the code that exits when the program is debugged.

You have successfully bypassed the anti-debugging mechanism!

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

Looking at the function which decodes the code, we can see that the loop starts at the start address of the decoded function (loc_4011C7 at the offset 4011C7) and ends at the start address of the decoding function (sub_4012CB at the offset 4012CB). Also we can see the address at which we should jump to get the code for the third function at the end of the loop (offset 401343). All this information is visible in the following screenshot.



Back into gdb, we set a breakpoint at the end of the loop (b *0x401343) and run the program until that point using the “continue” command. To be able to reach that point we should also introduce an input value, but its value is irrelevant. At the end of the loop, I dumped the binary code into the file task22_code.out using the command “dump memory task22_code.out 0x4011C7 0x4012CB”.

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_byte] 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 decompilable. (3p)**

As instructed, I have written a Python script (which can be found in the file task23_script.py) that “decompiles” the code from the function that checks the password. The IDA file containing the code after applying the Python script can be found in file task23.i64. After that, I took the code from the decoded function that checks the password and ran the code that generates the password (the initial for loop on the initial values), which can also be found in the file task23_passwordcode.c. Doing such, I obtained the password as the string “dynamic_analysis_is_the_best” which was correct when given as input to the binary as seen below.