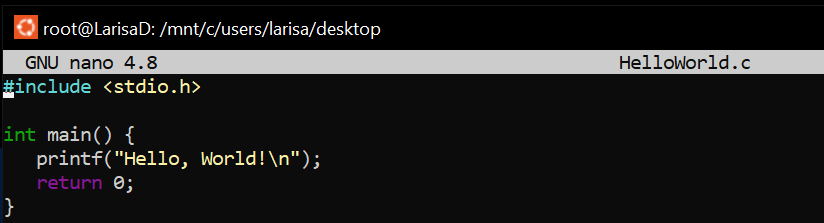
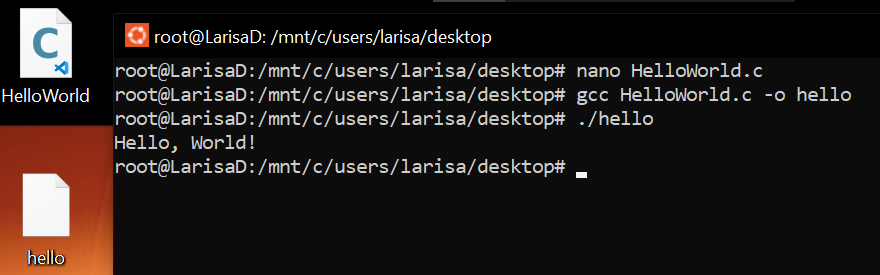
Lab Session 0x03

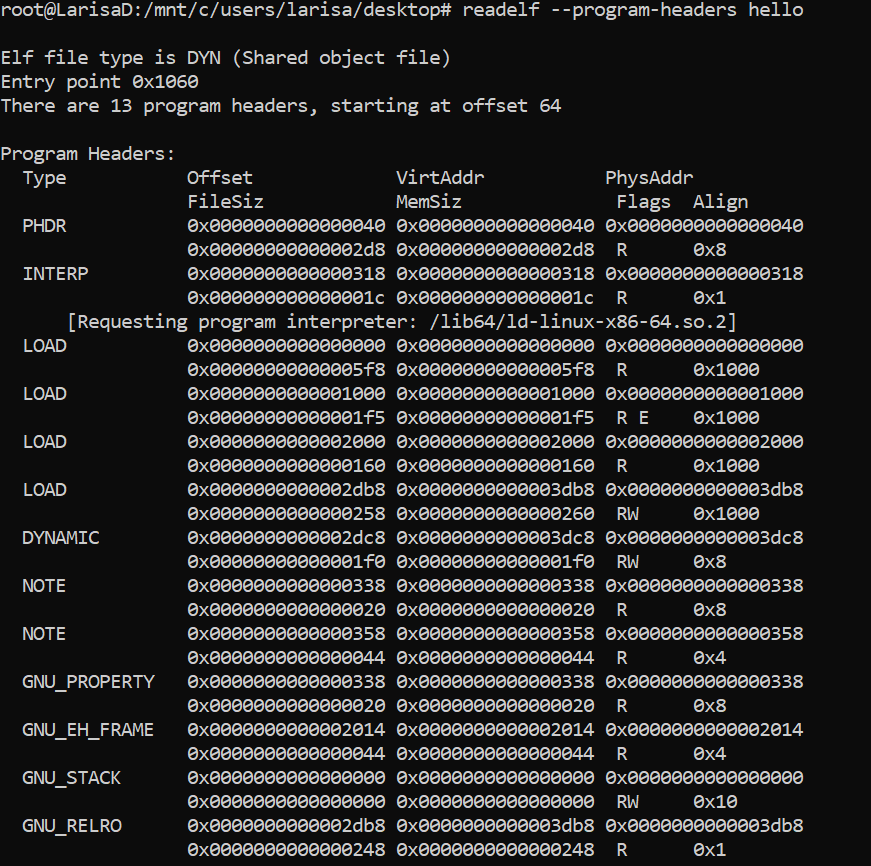
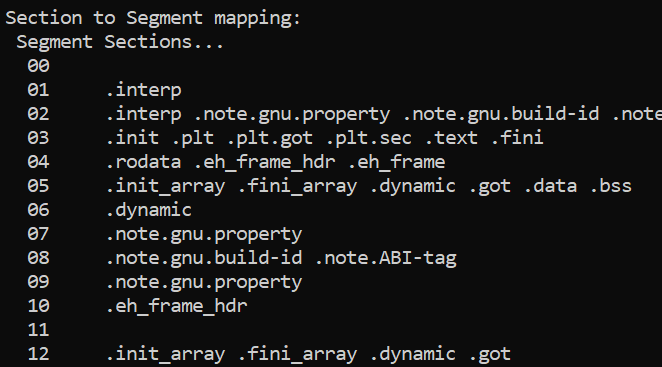
**Section 2: Basic Disassembly**

Write a basic „*hello world*” C program (*HelloWorld.c*). Compile it to the „*hello*” binary.

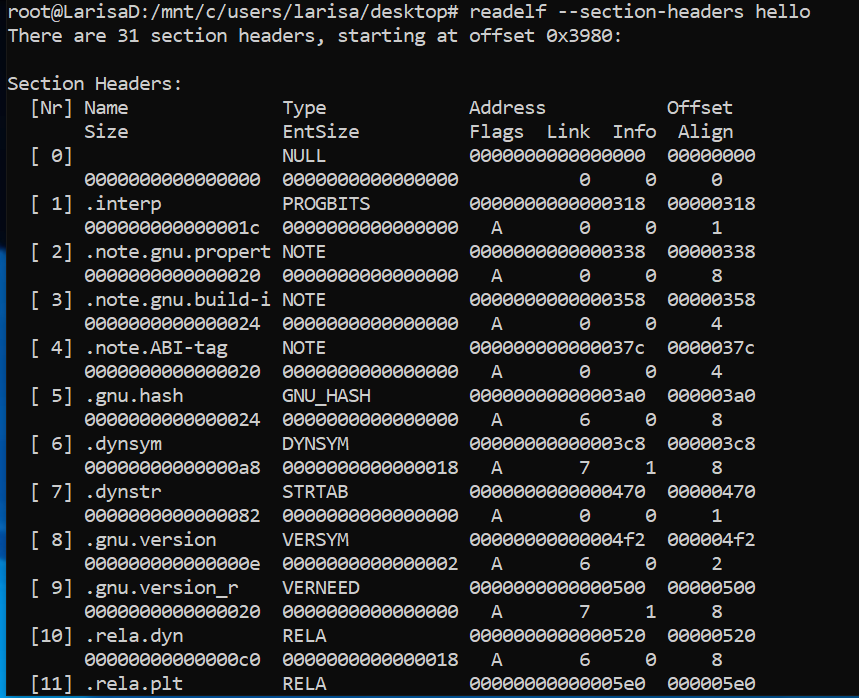




* Run *readelf --program-headers hello* to see the ELF program headers (*headers.txt*).

* Run *readelf --section-headers hello* to see the ELF program sections (*sections.txt*).

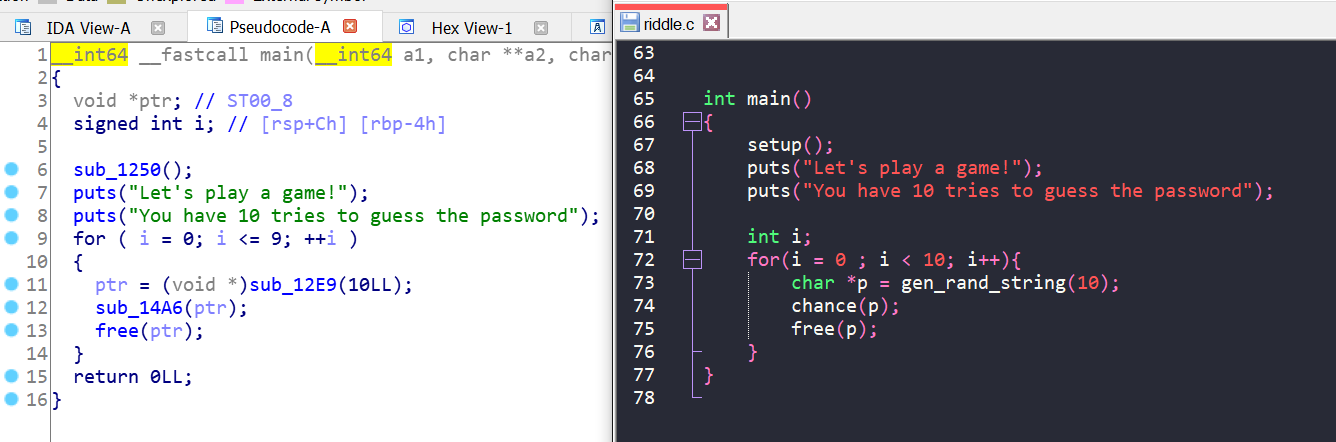


**4.1. RE with Spoilers**

*You have a binary, task1, and also its corresponding source code, task1.c. Using the stripped binary, you will simulate normal reverse engineering by using the source code (instead of guessing). Your task is to create a near-original replica of the original source in the IDA interface by:*

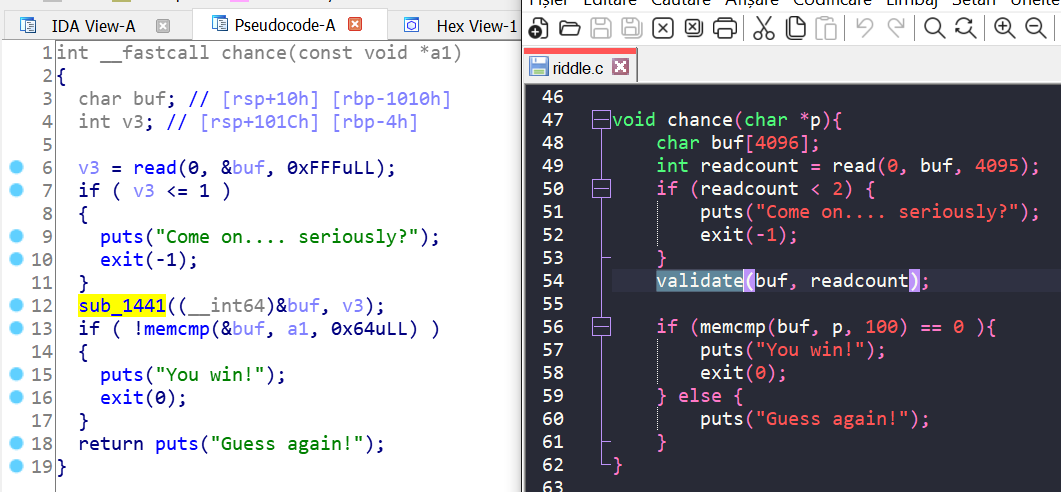
* *Renaming/retyping the 4 functions in the source code (aside from* ***main()****) (3p)*

After taking a look at the ***.c*** file we notice the following functions, aside from main: ***void setup(), char \*gen\_rand\_string(int len), void validate(char \*buf, int sz), void chance(char \*p)***.

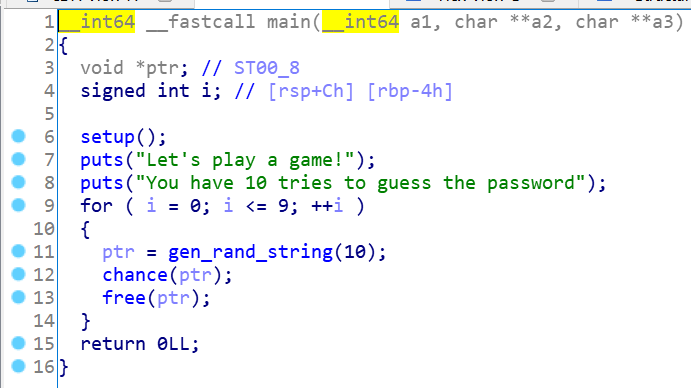
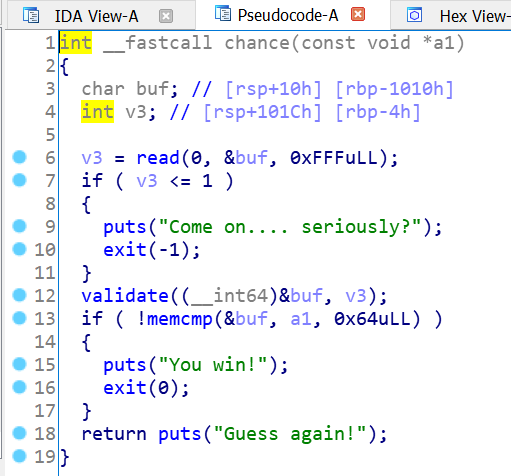


By comparing the pseudocode with the ***.c*** file we can identify the functions: ***sub\_1250()*** is ***setup()***; ***sub\_12E9()*** is ***gen\_rand\_string()***; ***sub\_14A6()*** is ***chance()***.

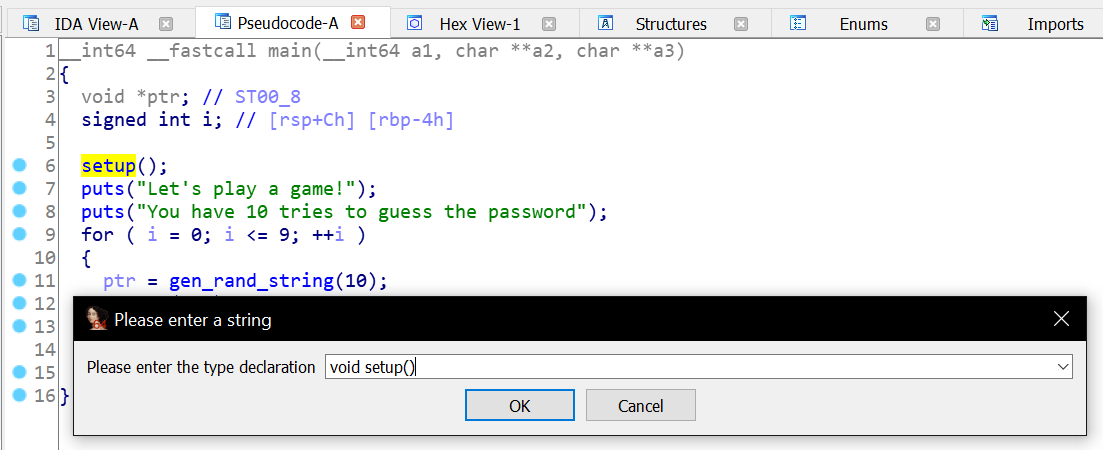
While renaming, we access *chance()* function and we notice that ***sub\_1441()*** is ***validate()***.

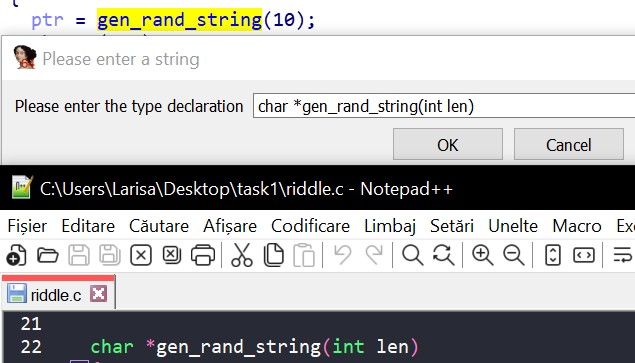
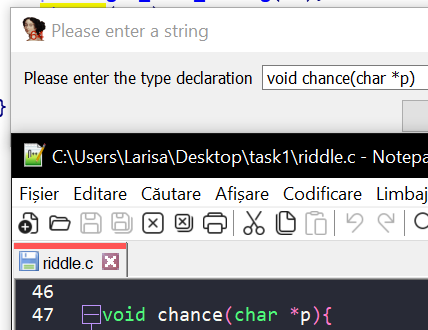


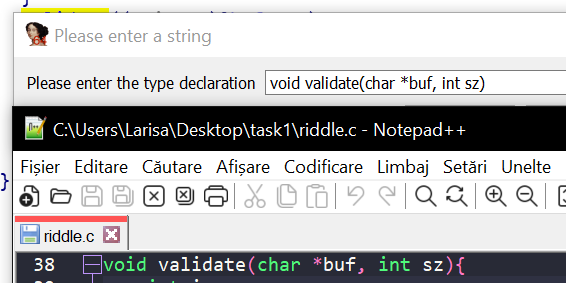
Result after renaming:

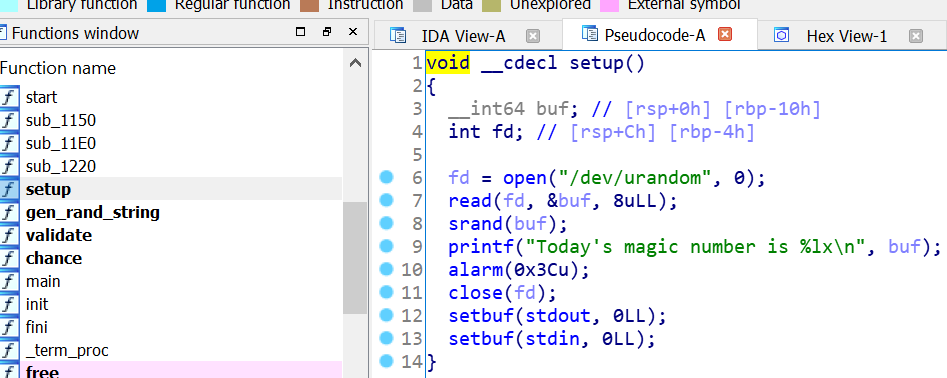
We go back to the *main* function (*chance* function respectively) and for each function, ***right click → Set Item Type*** (or ***click on the function + Y***) in order to retype them:

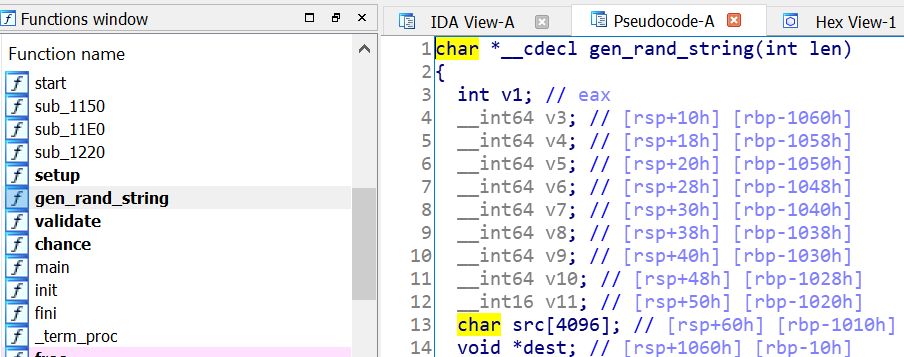


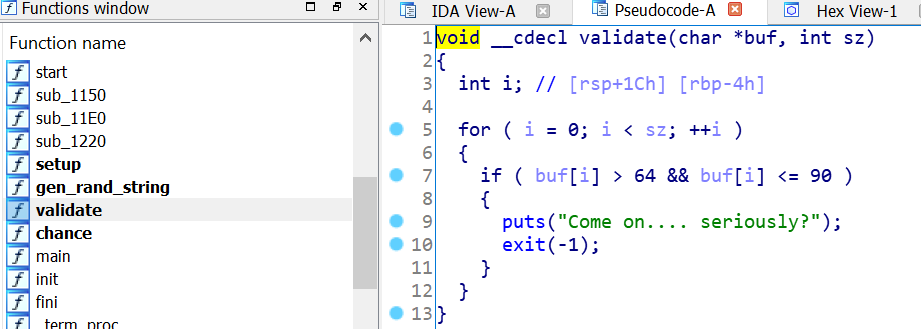
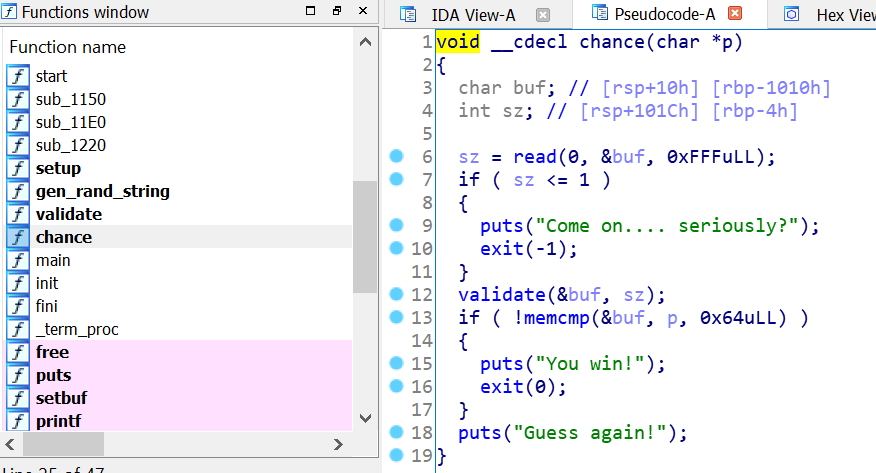
 



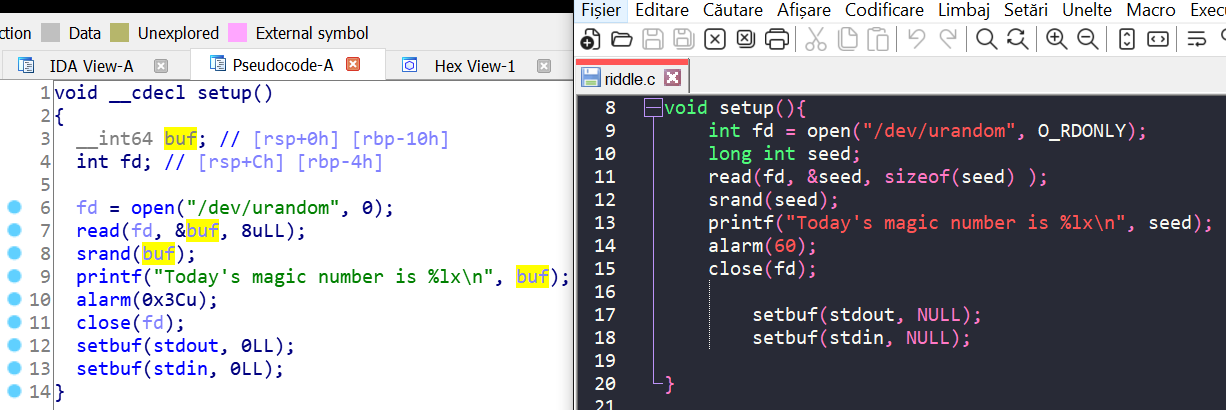
Result (*task1.1.i64*):



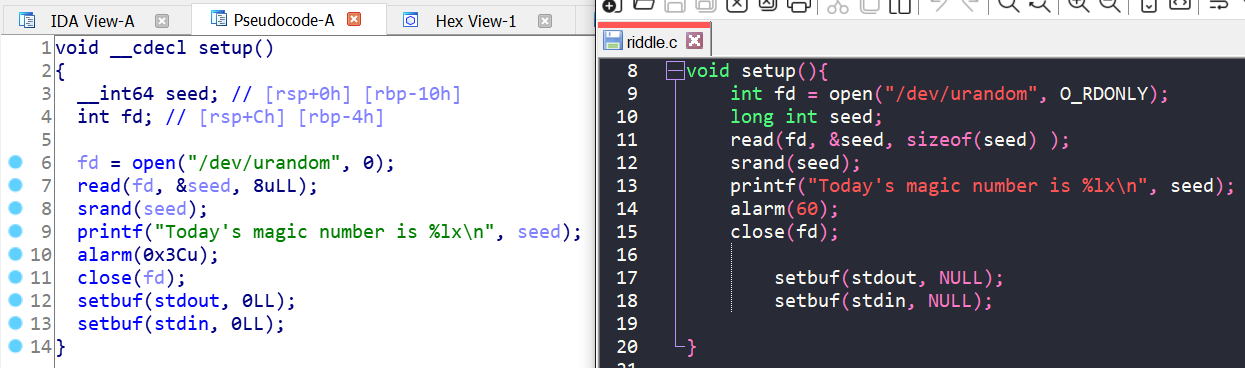


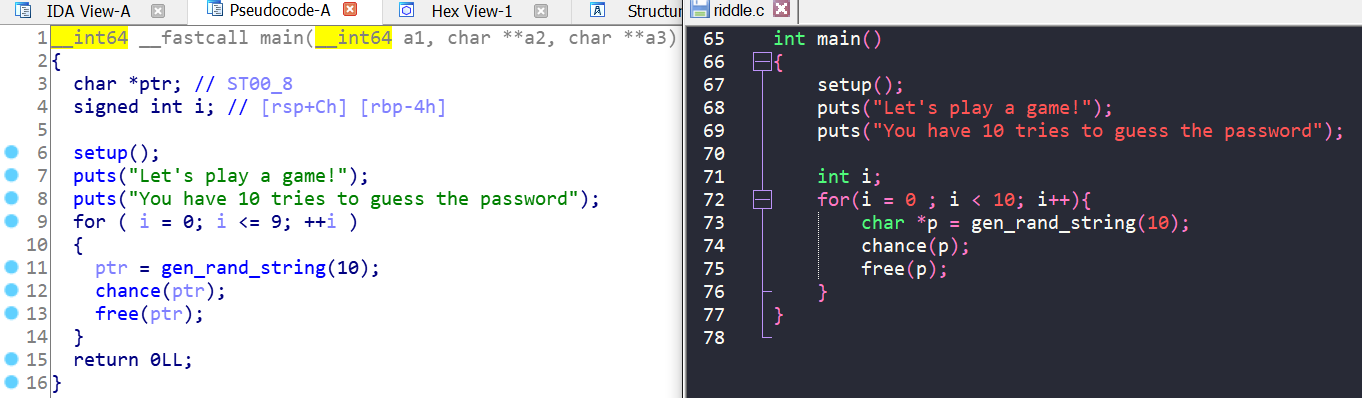
* *Renaming/retyping the stack variables in* ***setup()*** *and* ***main()****. (1p)*



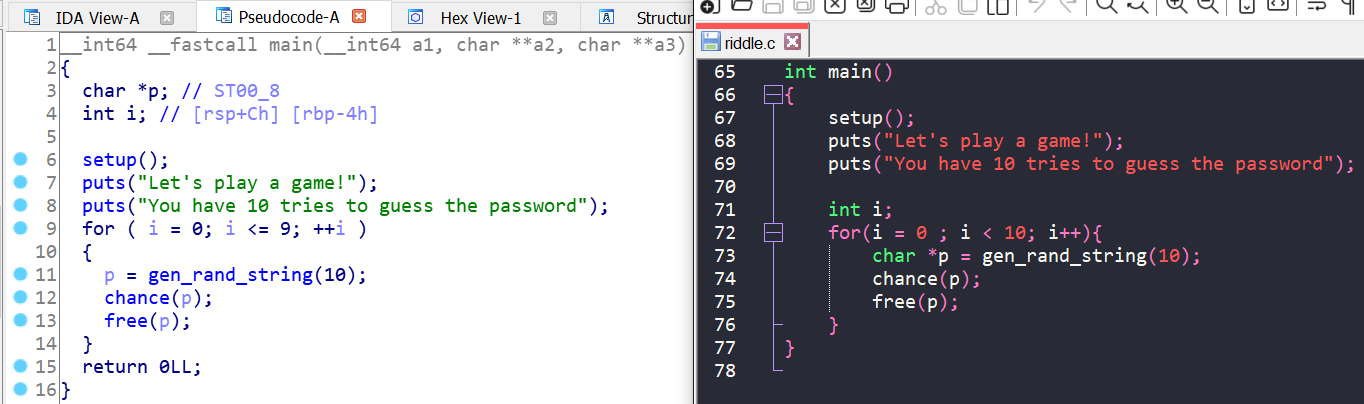
Stack variables aka local variables. So, we need to rename ***buf*** to ***seed***. And that’s it since ***fd*** is already correct and IDA does not make a difference between ***long int*** and ***\_\_int64***.



Before ***main()***:

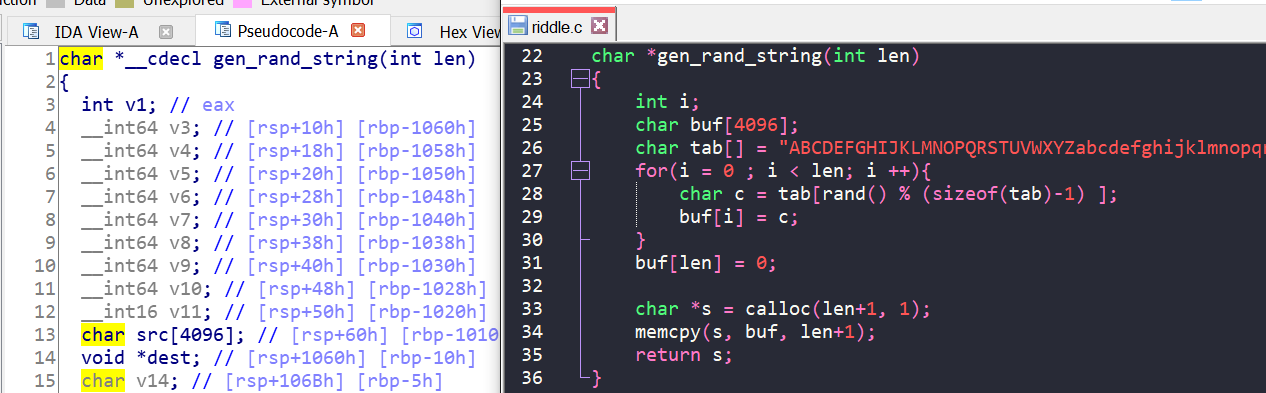


After ***main()***:



The solution can be found in *task1.2.i64*.

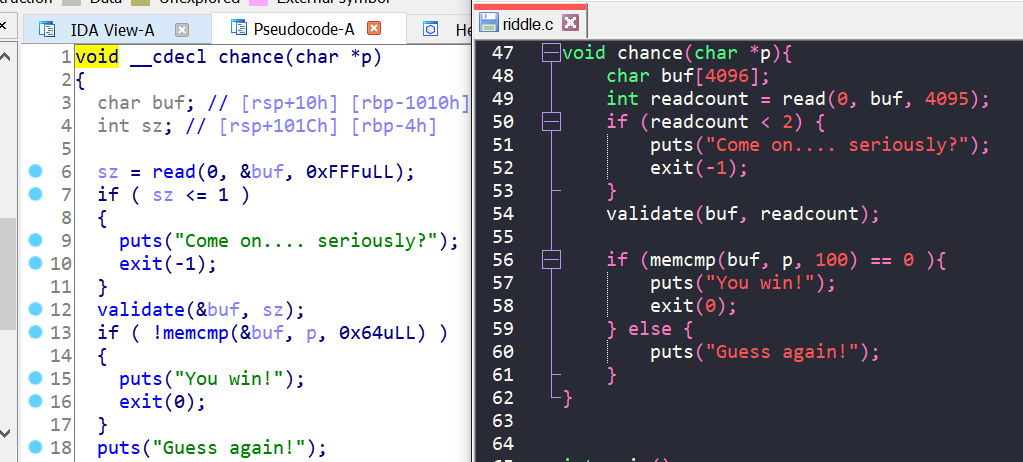
* *Renaming/retyping the stack variables (including the arrays) in* ***chance()*** *and* ***gen\_rand \_string()****. (2p)*
* Before ***gen\_rand\_string()***:



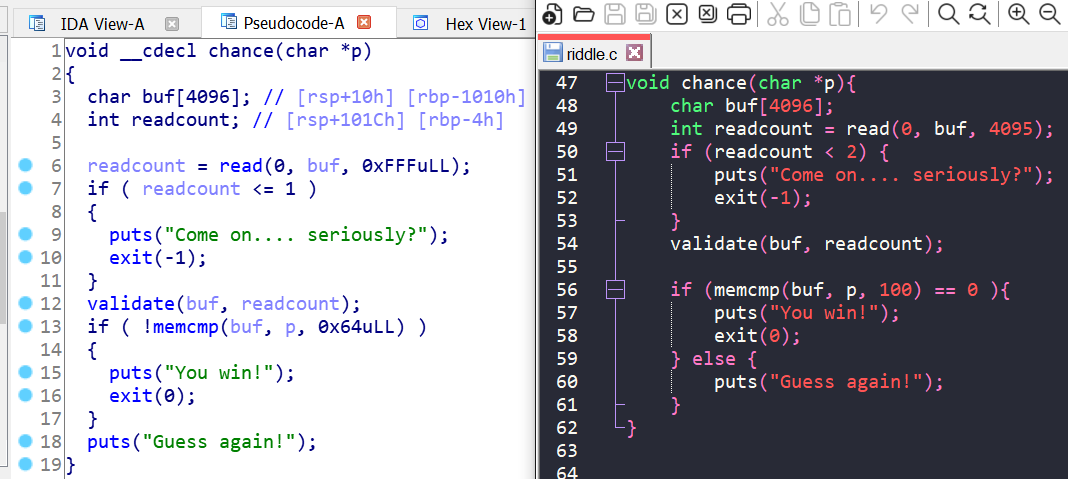
After ***gen\_rand\_string()***:



Before ***chance()***:



After ***chance()***:



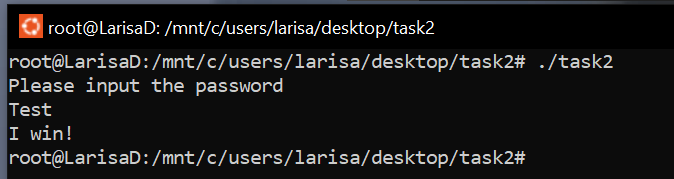
The solution can be found in *task1.3.i64*.

**4.2. Statically linked crackme - graybox analysis**

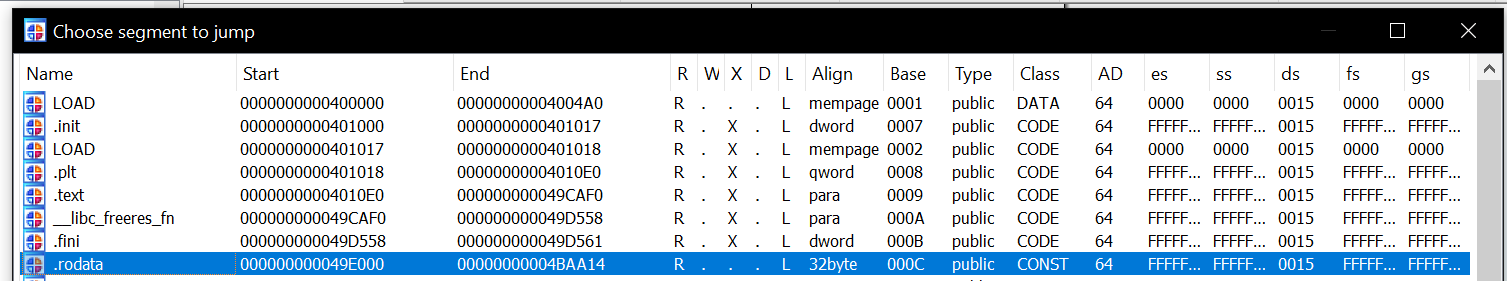
*In this task, you will learn to navigate through functions in a statically linked and stripped crackme. Since the binary has a whopping 783 functions detected, you do not have the time or motivation to go through all of them. As such, you need to approach the problem in a clever and elegant way:*

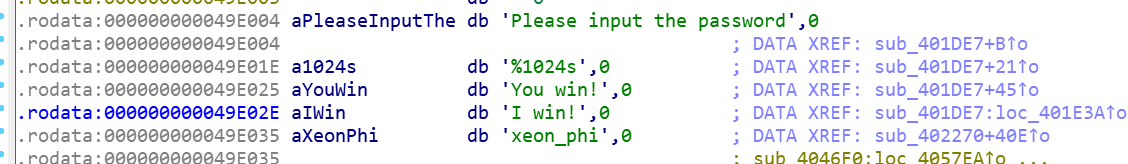
* *Run the program once and take note of any strings. Go to the* ***.rodata*** *segment (****Ctrl-s****) and find any/all of the strings. Using the* ***xref*** *functionality, determine where the* ***main()*** *function is. (1p)*

At first, we can notice 2 strings: „*Please input the password*” and „*I win!*”. Most likely, we will get another string when we input the correct password.

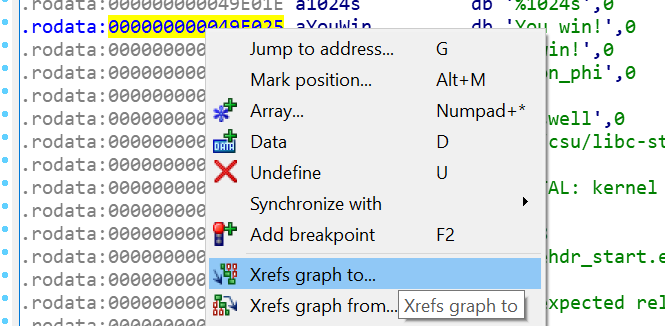
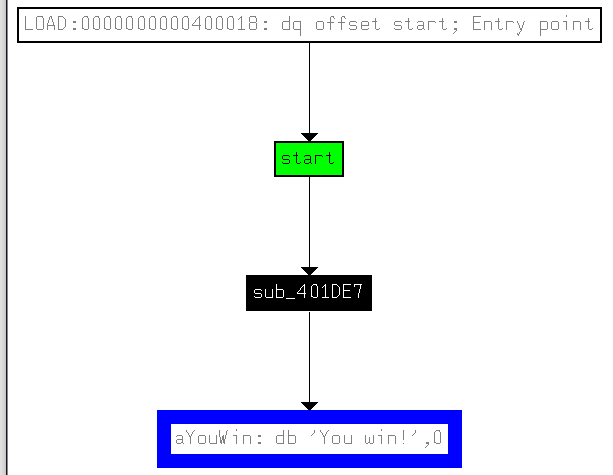


Now, we look at the **.rodata** segment in IDA (**CTRL + S**):

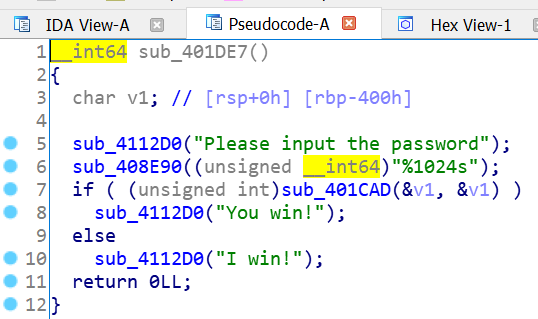




As I mentioned previously, we can observe another string, for the case when we input the correct password: „*You win!*”. And, let’s find out where the ***main()*** using ***xref*** is:

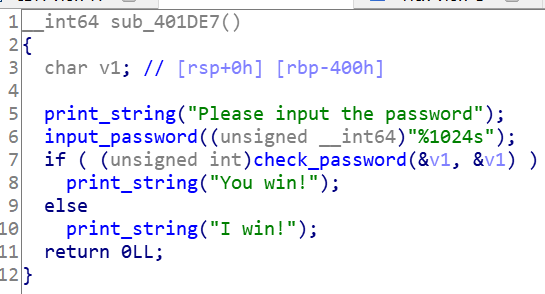
 

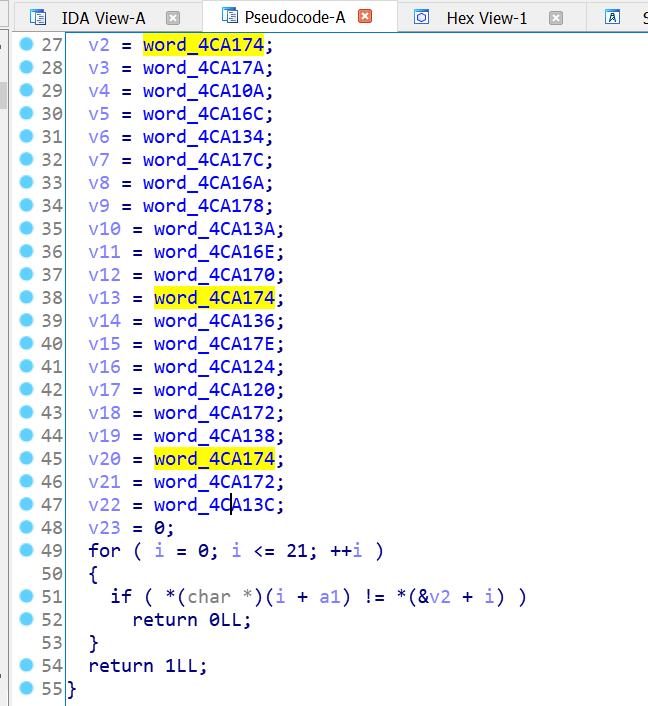
So, ***sub\_401DE7()*** is ***main()***:



* *Rename all the functions in* ***main()*** *and determine the password-checking function. (1p)*

Without looking further inside the functions (only observing the code form *main()*) we can guess what each individual function does; so, after renaming we get:

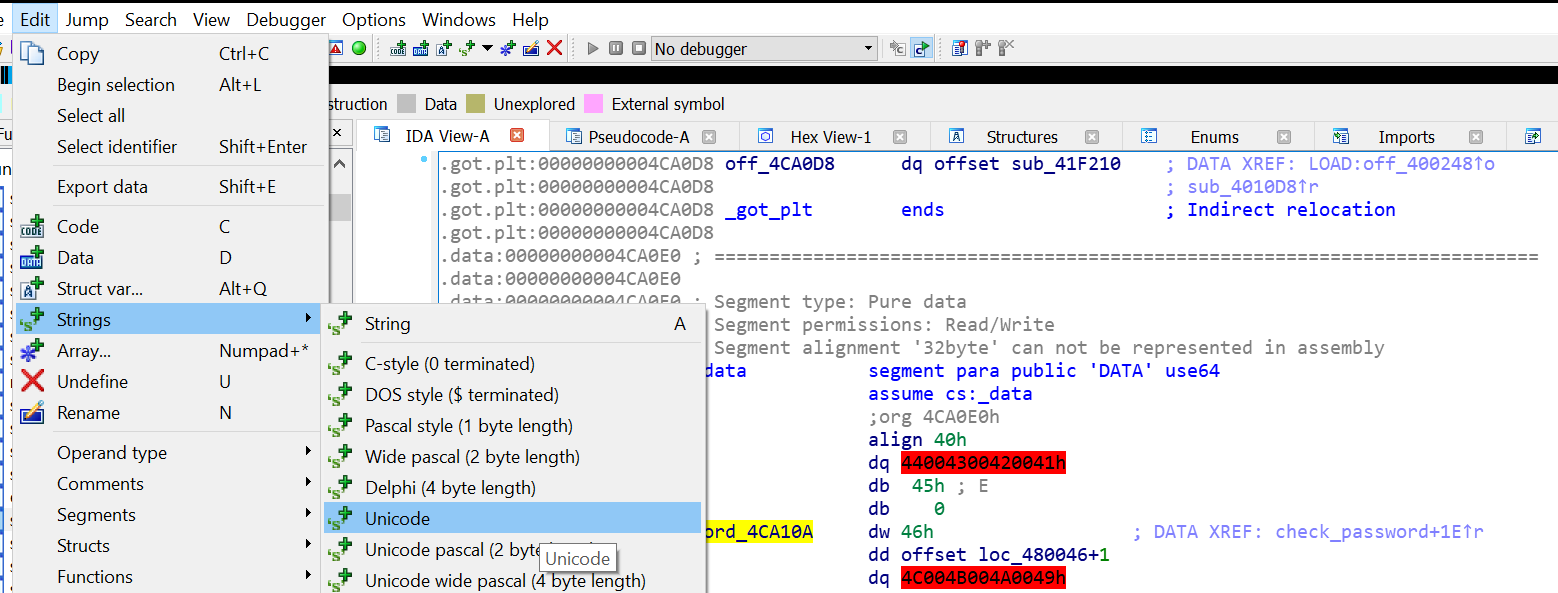


* *In the password-checking function, observe how the correct password is generated; we want to make this function more readable.*

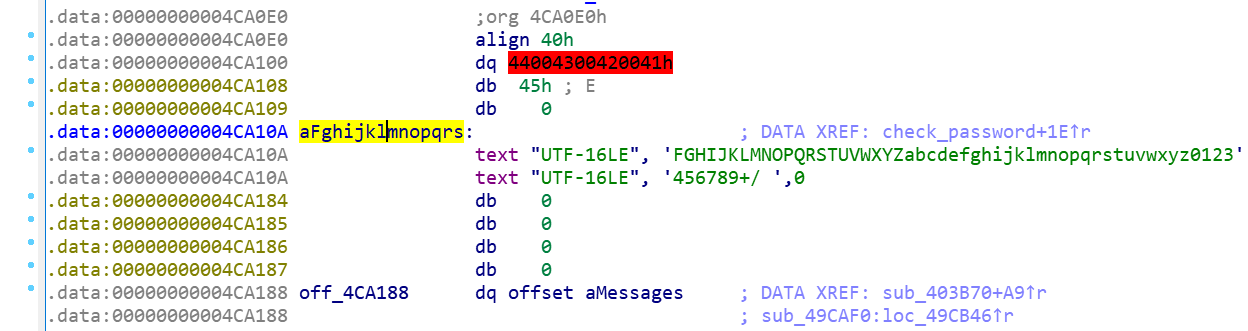
It seems that every character of the password provided by the user is checked, in order, against *v2, v3, ..., v23*. If, at some point, we don’t have a match, the function exits with 0 (FALSE), otherwise with 1 (TRUE).

* *Go to the location of any* ***word .....*** *variable in IDA-view and find the location of the start of the alphabet and redeclare that address as a wide C string (****Edit****→****Strings****→****Unicode****).*

Double click on the the first „*word\_*”, then scroll up:



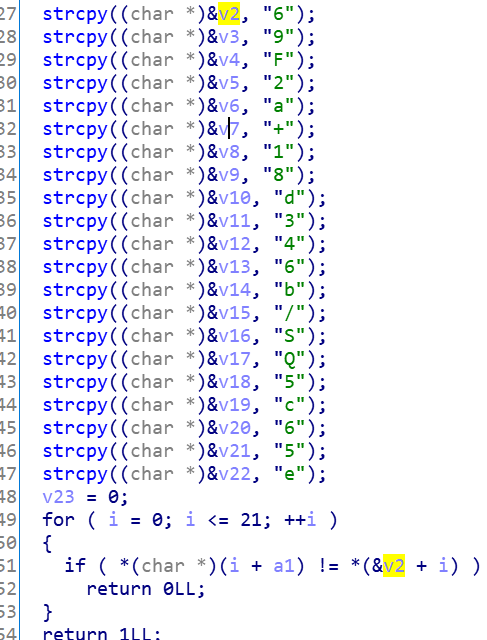
What we get:



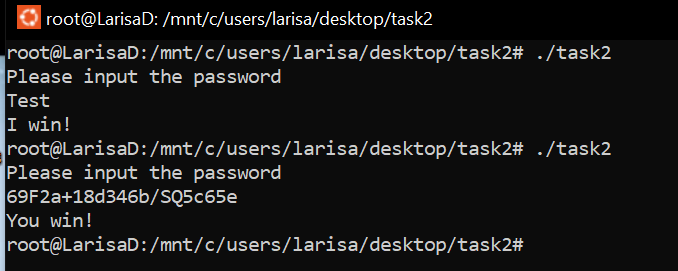
The file is: *task2.1-2.i64*

* *Again, in the password checking function, observe how the right-hand side looks now. Redeclare the alphabet with the “const” modifier at the beginning. This should collapse the function and reveal the correct password. Finally, check that the password is accepted. (2p)*

On the first *word\_* + Y, then declare as ***const*** and:



We found the password: ***69F2a+18d346b/SQ5c65e***

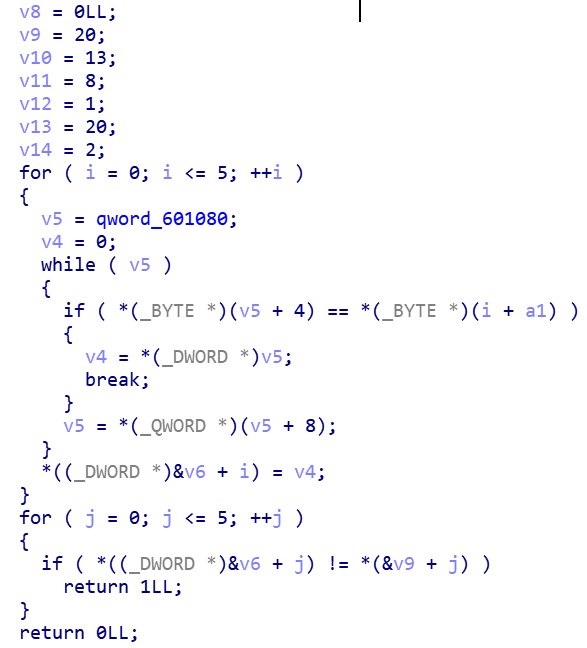
Let’s check it:

The solution is in: *task2.3.i64*

**4.3. Data Structures**

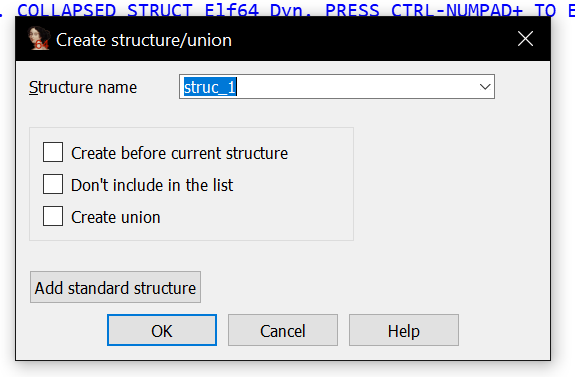
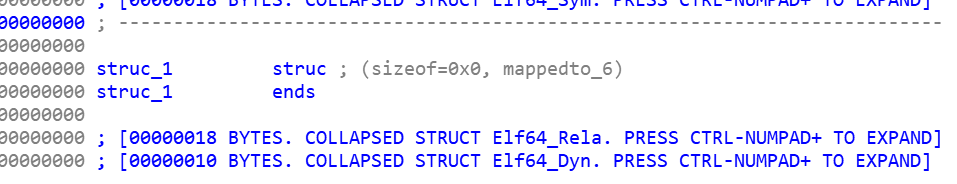
*Look at the code in* ***main()*** *and the* ***password checking function****, analyze the access patterns, and verify that it matches the linked list structure below.*

***Main()***: ***sub\_40074D()*** → ***check\_pass()***:

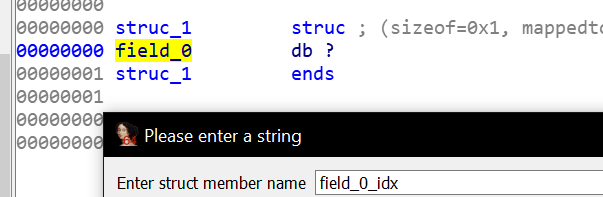
 

* *Use the* ***Structures*** *tab and create the following list structure (also declare field 8 next as a struc 1\* pointer) (2p)*

Edit → Add structure type:

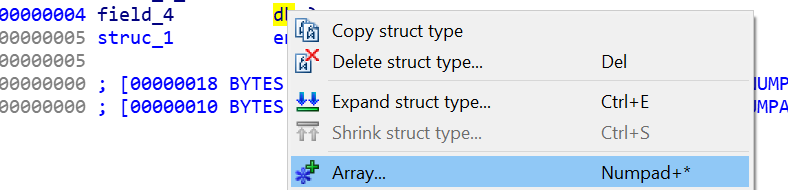
Let’s define it as it is requested (Press ***D*** to create structure member and rename it):



After renaming, click the name so it is highlighted yellow and press ***D*** to change the type:

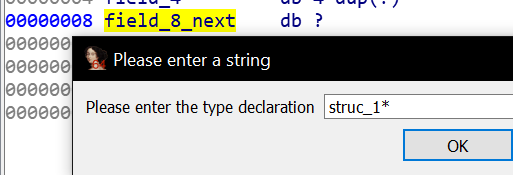


*Field\_4* is an array with 4 elements:

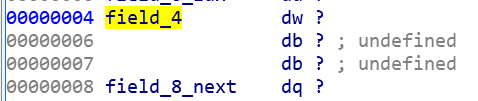




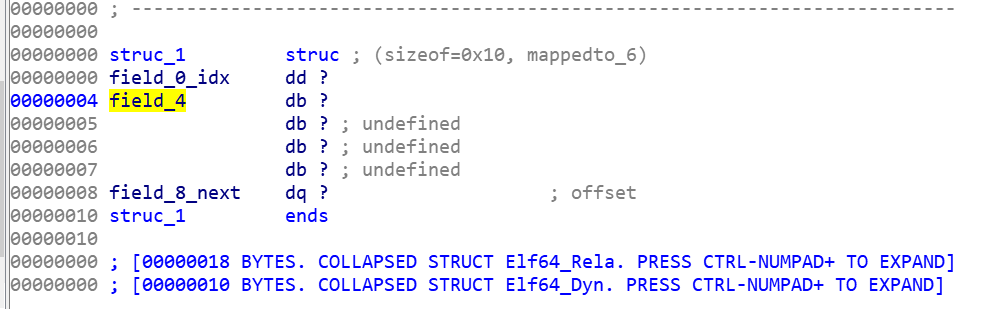
For *field\_8\_next*:



Now press ***D*** while on *field\_4*:

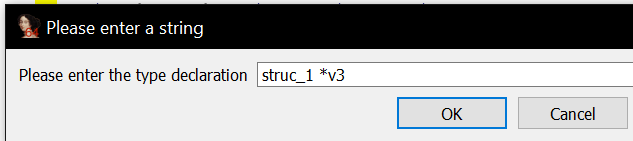


And retype it as char. The final result:



The solution is in: *task3.1.i64*

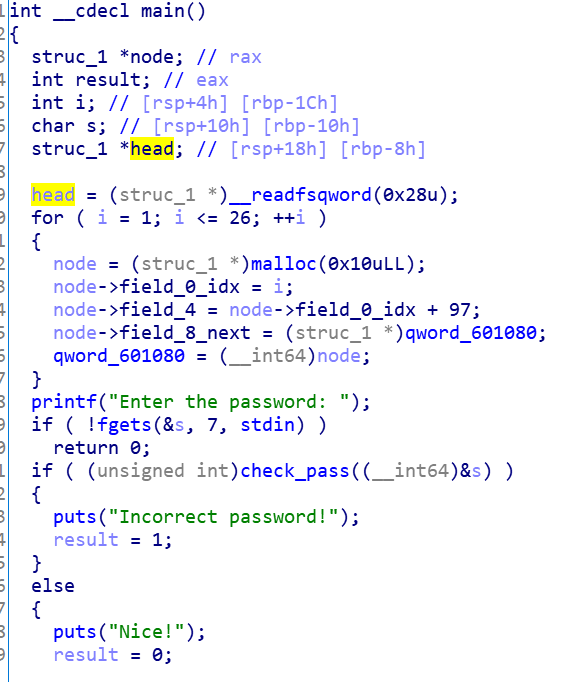
* *In* ***main()****, cast the buffer returned from malloc and the head of the list to this struct type and propagate in the password checking function, renaming and retyping where necessary. (2p)*



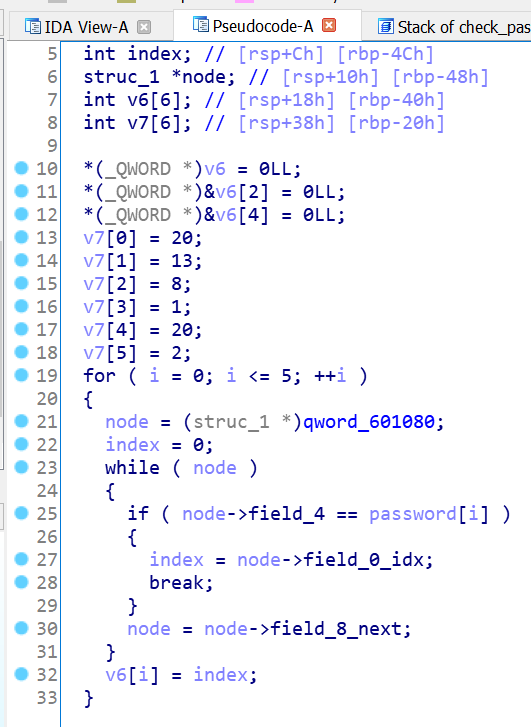
We get:



After refining the data:



And for ***check\_pass()***:



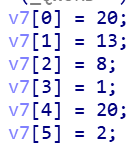
The solution is in: *task3.2.i64*

* *Describe what the code does and figure out the correct password. (2p)*

In *main()*, first, a linked-list is created, where ***field\_0\_idx*** {1, 2, ..., 26}, ***field\_4*** {b, c, ..., z, ?} (because 97 is ASCII for ***a***) and ***field\_8\_next*** is the pointer to the next node.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **1** | **2** | 3 | 4 | 5 | 6 | 7 | **8** | 9 | 10 | 11 | 12 | **13** | 14 | 15 | 16 | 17 | 18 | 19 | **20** | 21 | 22 | 23 | 24 | 25 | 26 |
| **b** | **c** | d | e | f | g | h | **i** | j | k | l | m | **n** | o | p | q | r | s | t | **u** | v | w | x | y | z | { |

Then, it takes the first 6 characters of the password inputed by the user and passes them to the *check\_pass()* function. Inside this function, the first for, takes those 6 characters and inserts the corresponding number (from the table above) in the v6 array. Then compares v6 to v7 array.



So, following the table from above, that means the password should start with: ***unibuc***

Test:

