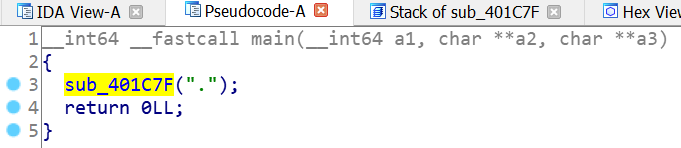
Project 0x01

*Perform the following tasks:*

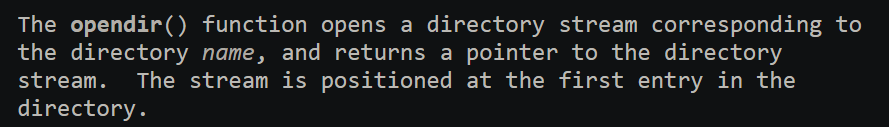
* *The binary searches for files with a certain pattern and only encrypts those that match. Find out what the pattern is. (20p)*

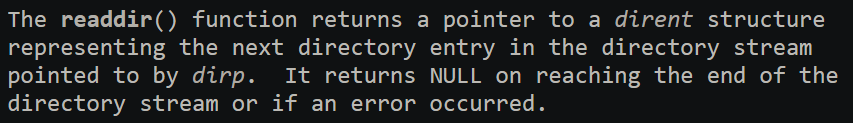
Let’s analyze the file with IDA (64-bit). The main function contain only a function (***401C7F*** – renamed as ***find\_files\_to\_encrypt***):

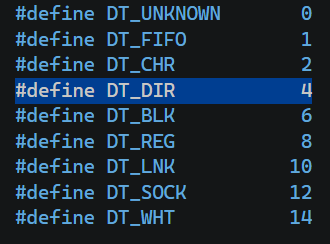
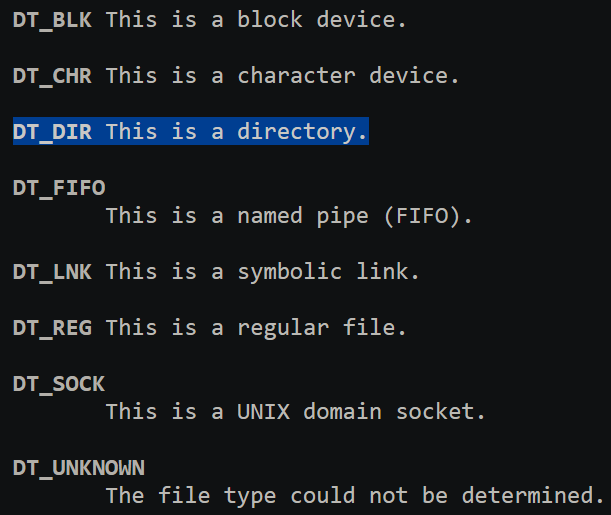


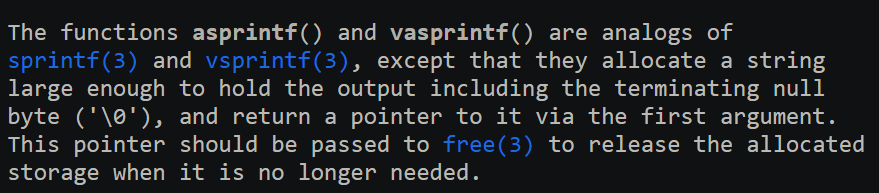
Let’s see what this function is doing:

So, it seems that our function opens the directory ***a1*** (initially, we are in the directory where the ransomware is situated – ***"."***), then finds all the directories inside ***a1*** (***v5->d\_type == 4***) and recursively calls itself (the parameter being the new found directory name). If the files found inside the directory are not directories, the function ***sub\_401BA5()*** (renamed as ***process\_files()***) will be called (we can notice this is the place where the encryption takes place). ([Source1](https://man7.org/linux/man-pages/man3/fdopendir.3.html), [source2](https://man7.org/linux/man-pages/man3/readdir.3.html), [source3](https://stackoverflow.com/questions/23958040/checking-if-a-dir-entry-returned-by-readdir-is-a-directory-link-or-file-dent), [source4](https://man7.org/linux/man-pages/man3/asprintf.3.html), [source5](http://www.crasseux.com/books/ctutorial/asprintf.html))

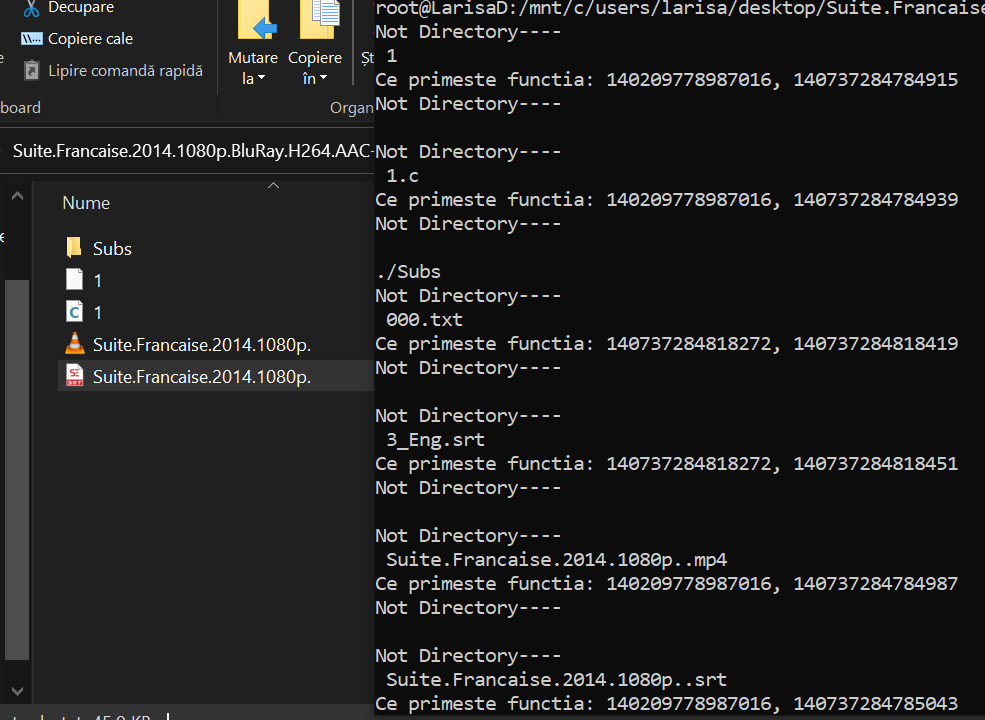




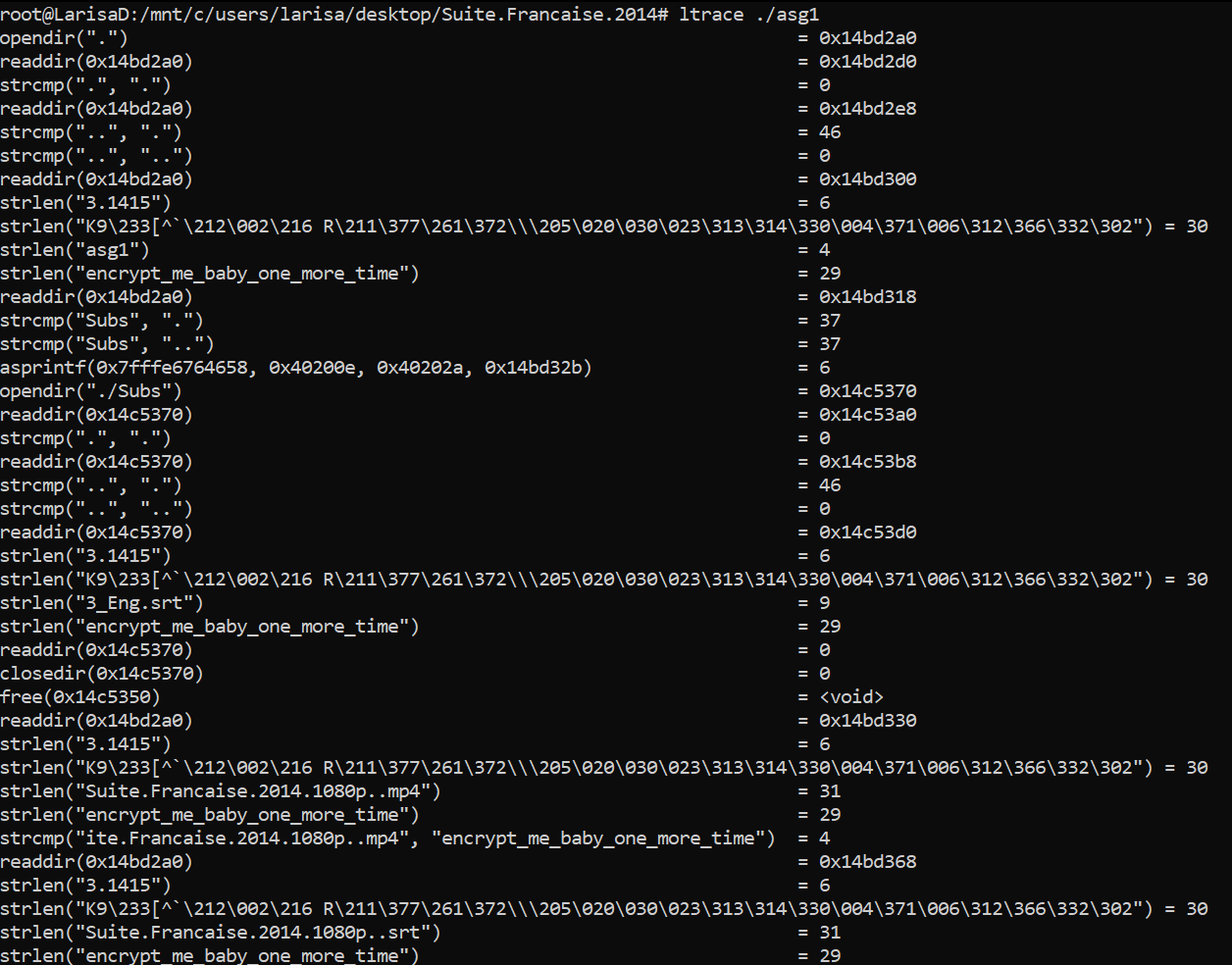
 



Based on the functionality of the ***sub\_401C7F()*** function, we can rename it: ***find\_files\_to\_encrypt()***. The algorithm finds all the files inside the directory and subdirectories where the ransomware is situated. I wrote a function that simulates this one: ***1.c***.

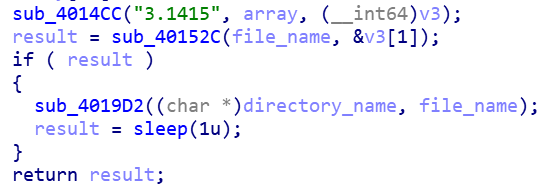


So, knowing this, let’s use ***ltrace*** and see what we get:

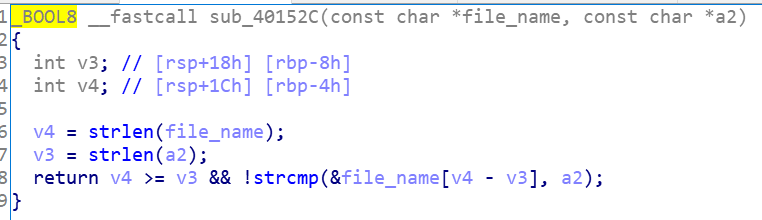


So, if the program finds a file, then it enters ***sub\_401BA5()*** (renamed as ***process\_files()***).

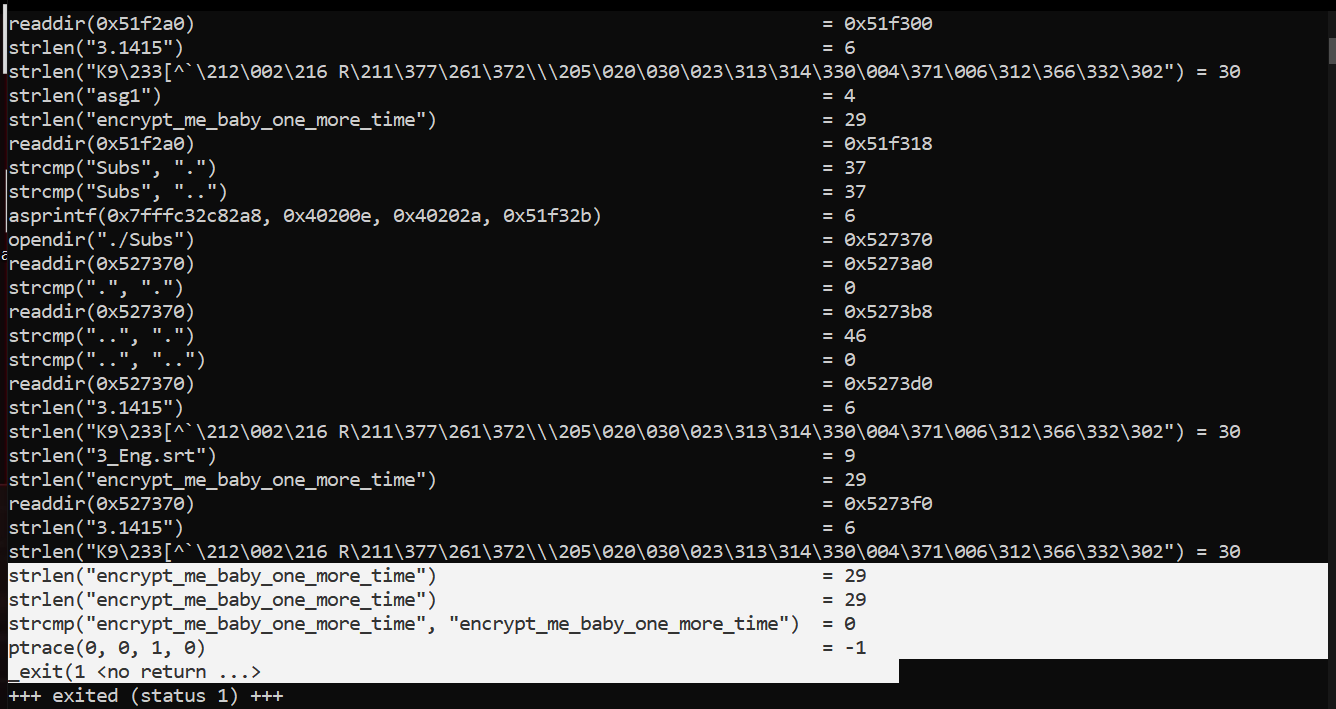


***sub\_4014CC()*** (renamed as ***create\_constant\_string()***) will create the const string *v3* = „***encrypt\_me\_baby\_one\_more\_time***” (entering the functions used inside it, we can follow the path of the *ltrace*). The function ***sub\_40152C()*** (renamed as ***if\_file\_should\_be\_encrypted()***) checks wheter or not the file should be encrypted:

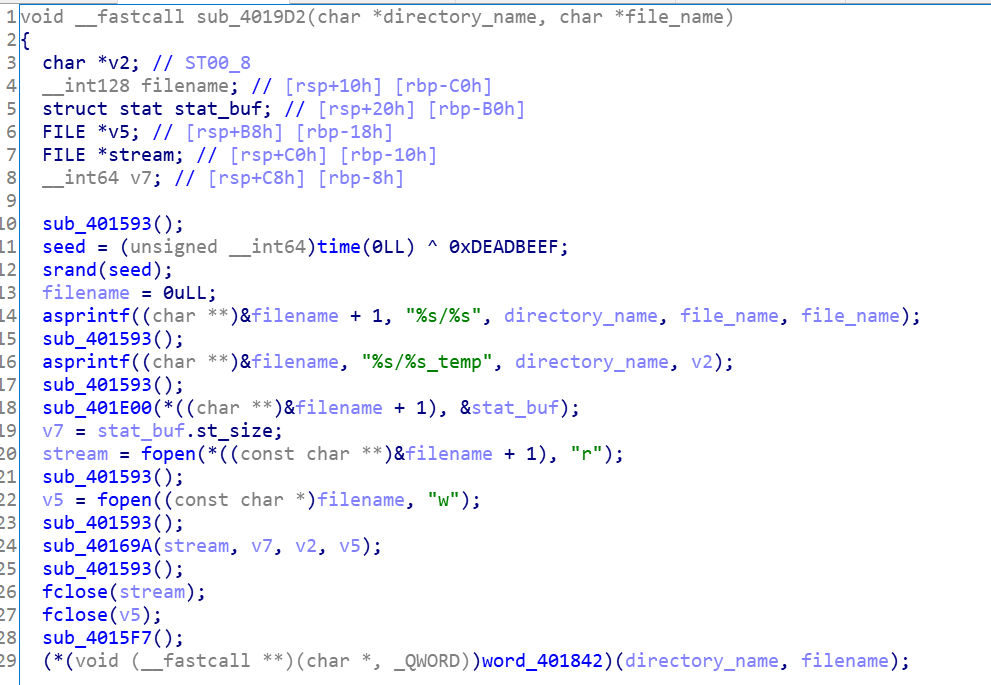


So, in order for the file to be encrypted, the length of its name must be longer or equal than strlen(*v3* from above) and the file name must contain the substring „***encrypt\_me\_baby\_one\_more\_time***” (must be the sufix) based on the rule from above.

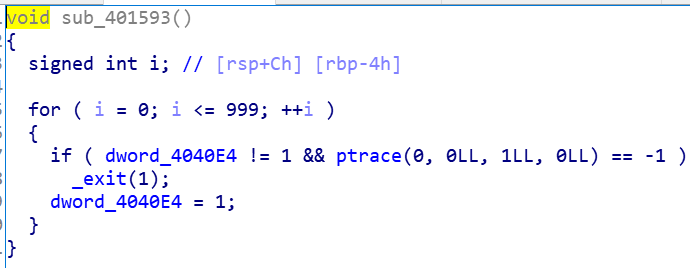
In the photo from above, none of the files respect the conditions so they are not encrypted. Let’s create a file that follows the rule:



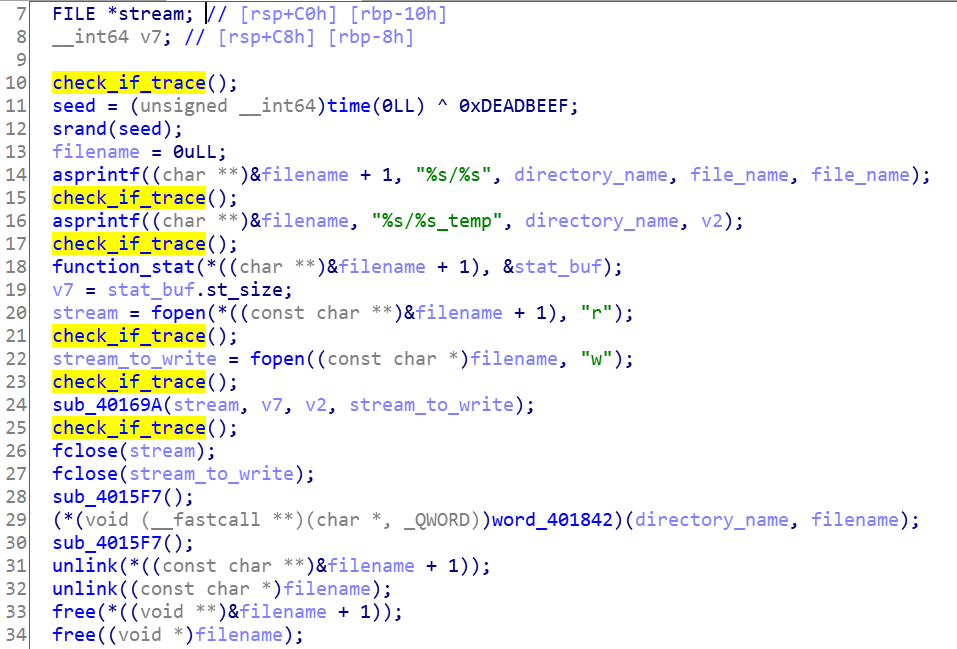
So, we manage to enter ***sub\_4019D2()*** (renamed as ***ransomware()***):



And inside ***sub\_401593()*** (renamed as ***check\_if\_trace()***):



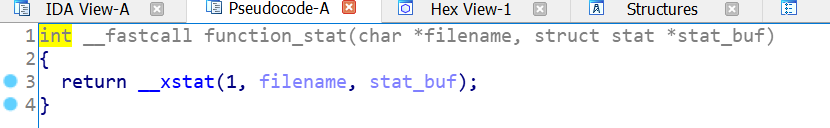
Now, ***ransomware()*** looks like this (file is *aux1.i64*):



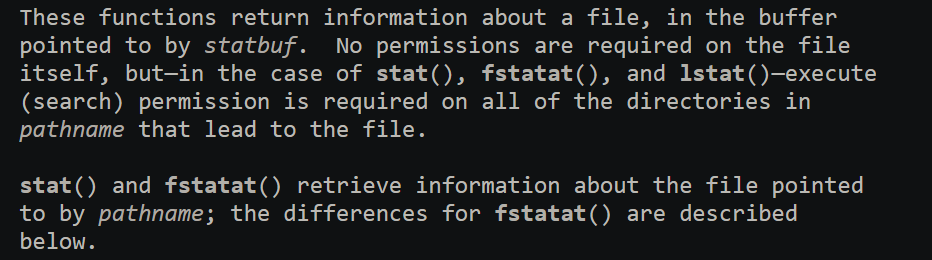
* *Describe how the encrypted files are internally structured (what bytes are written in the encrypted files and how the encryption is done). (50p)*

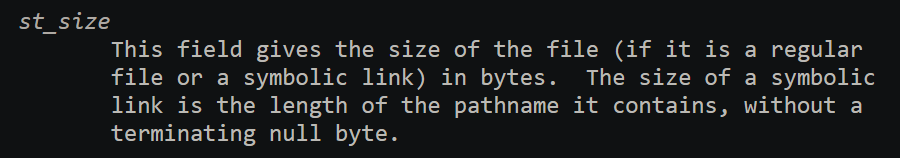
As we mentioned above, the function renamed as ***ransomware()*** is the place where the encryption of the files actually takes place. We can see that the algorithm generates a random value that has as seed the current time XORED with **0xDEADBEEF**. Moreover, the function creates a temporary file (named *%filename%\_temp*) inside the current directory.

Function ***sub\_401E00()*** renamed for convenience as ***function\_stat()***, retrieves information regarding this file.



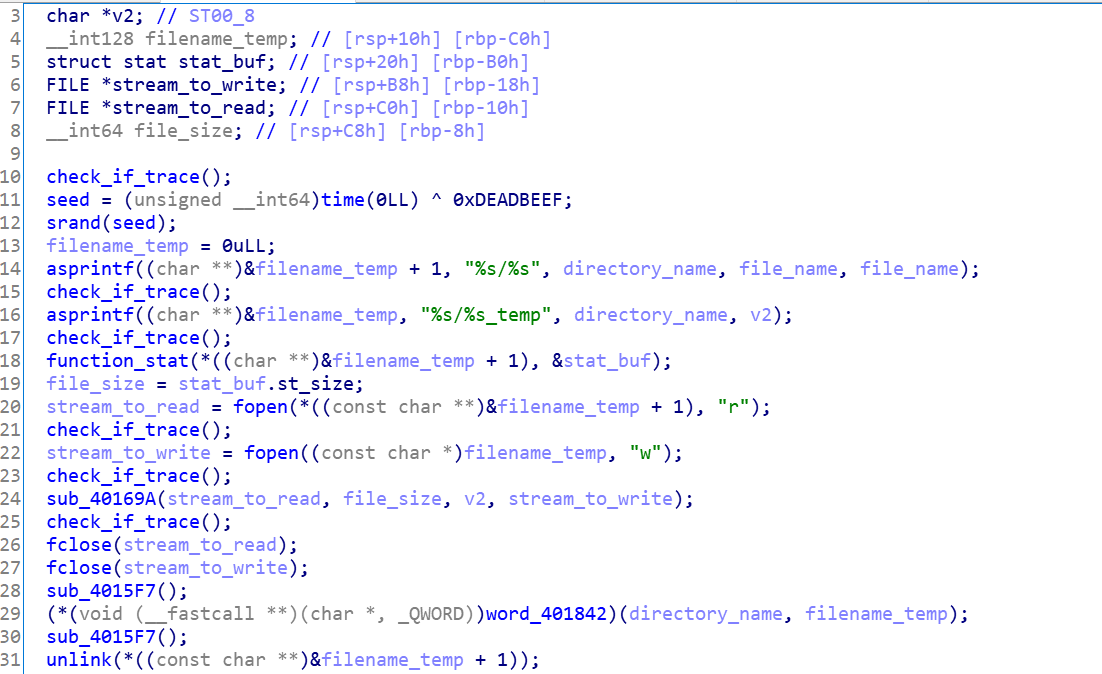




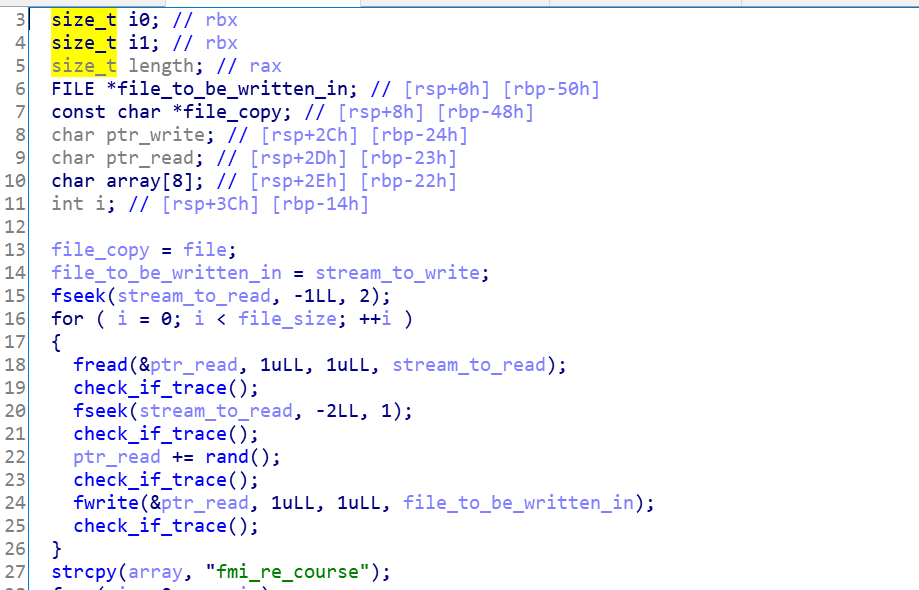


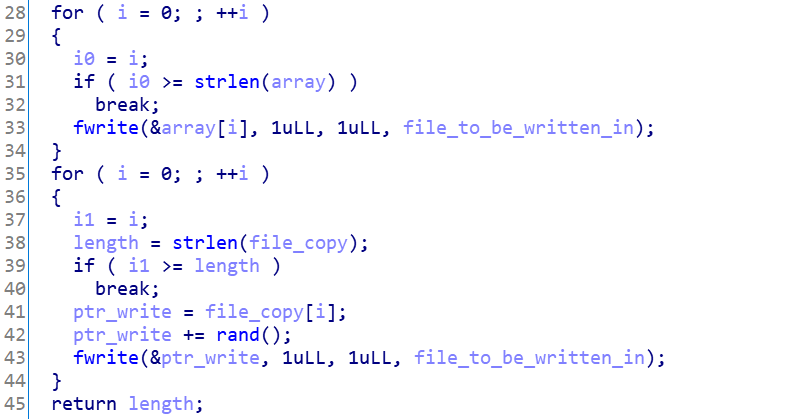
([source1](https://refspecs.linuxbase.org/LSB_3.0.0/LSB-PDA/LSB-PDA/baselib-xstat-1.html), [source2](https://man7.org/linux/man-pages/man2/lstat.2.html))

After renaming more intuitively, this is the function ***ransomware()***:

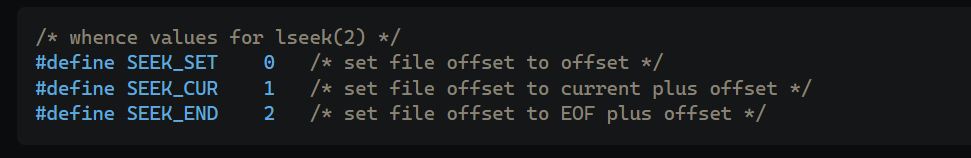
****

Of importance to us (for now), it’s ***sub\_40169A()***. After cleaning the code a little bit, we get this (*aux1.i64*):





Let’s try to understand the code.



([source](https://stackoverflow.com/questions/38226713/fseekf-0-2-how-does-this-work-without-seek-cur-seek-set-or-seek-end))

At line 15, we position the pointer in the file (from which we read) at EOF – 1. After that, in the first *for*:

- we read each byte (stored inside ***ptr\_read***) (line 18)

- position the pointer in the file at ***SEEK\_CUR – 2*** (in order to pass the current byte) (line 20)

- at ***ptr\_read*** we add a random value (obtained from the seed from the previous function; we talked above about it) (line 22)

- then we write inside the temporary file the new obtained byte (line 24)

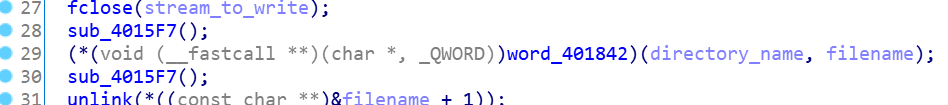
Looking at the second *for*, we can see that ***array*** gets initialised with „***fmi\_re\_course***” and then, we copy (write) each byte of this array inside the temporary file.

The 3rd *for* is very similar to the first one, but now, we write each byte (we read them in the proper order, not the reversed one) from the title of the initial file (the filename) into our temporary file.

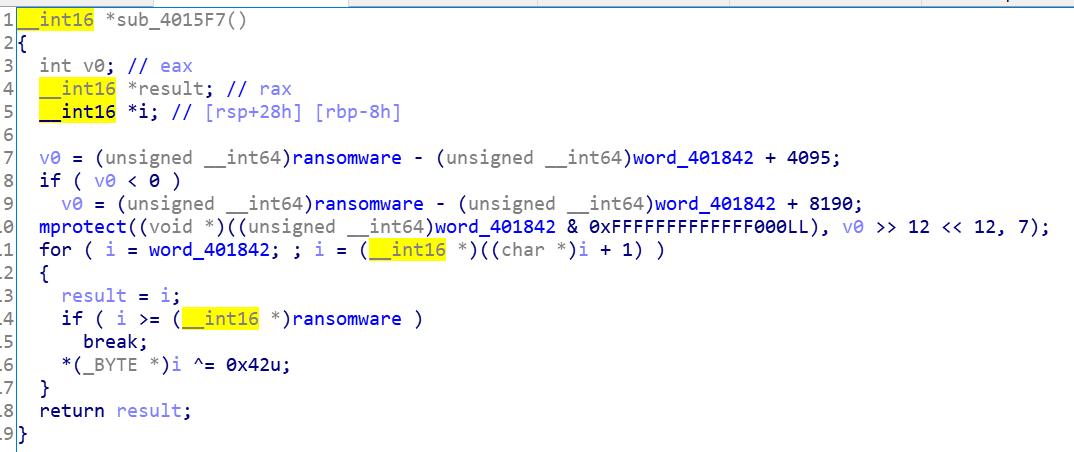
In conclusion, the temporary file contains the original file (altered, of course), the constant array and the original filename (altered).

* *Figure out how the file renaming process works and describe how decryption could theoretically be done. (20p)*

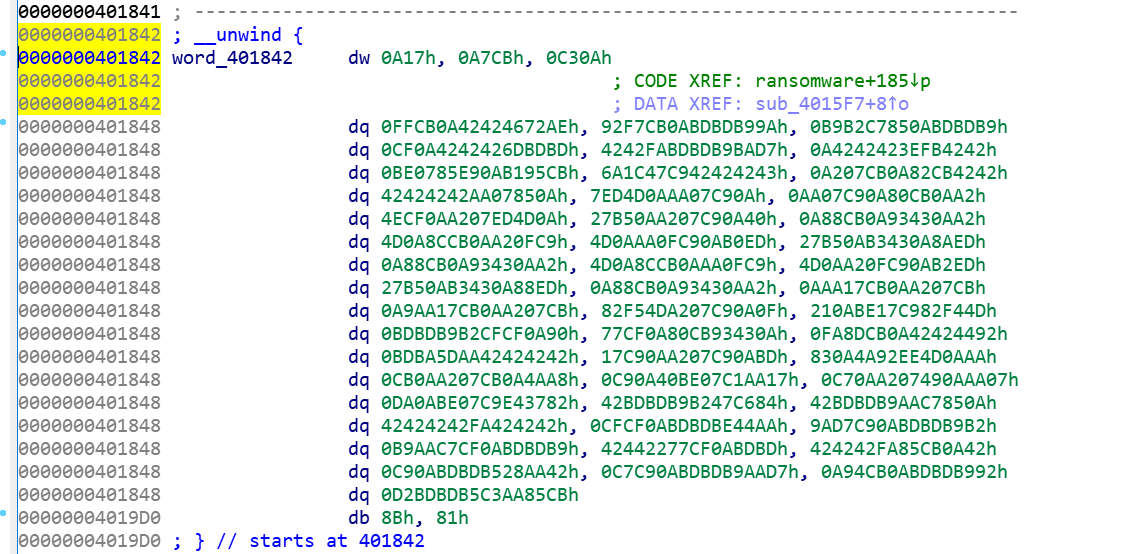
The part of code that interests us are lines 28 – 30 (inside function ***ransomware()***):



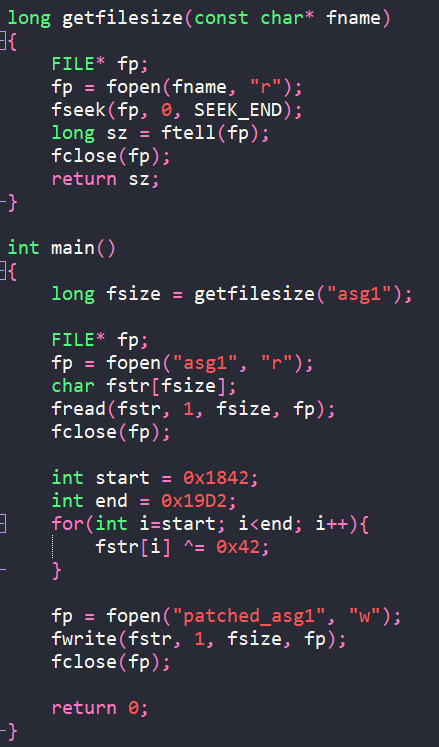
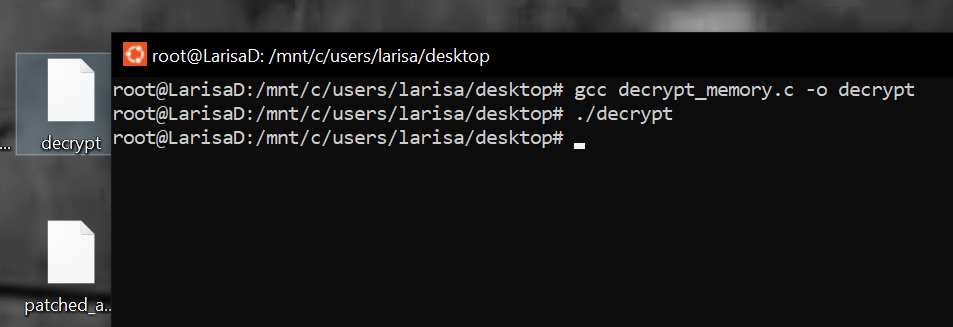
Let’s see what is inside the function ***sub\_4015F7()***:



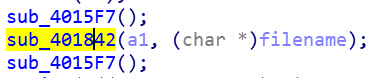
And for ***word\_401842***:

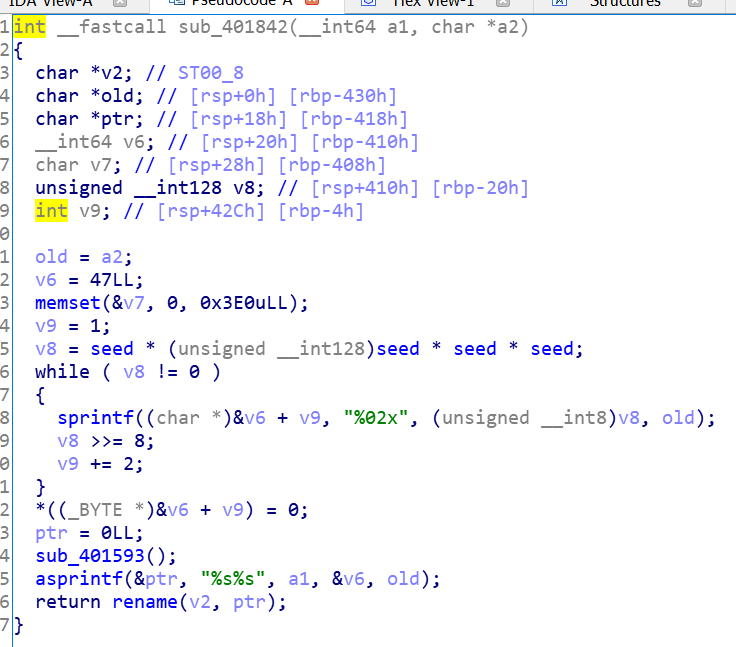


We’ve encountered this situation once before, in the 4th laboratory – basically, the function acts like a decriptor for ***word\_401842*** (that will turn into a function, after decription). We can actually modify the ELF and then open it with IDA (the start address is the address of ***word\_401842*** and the end address is the address for function ***ransomware()***, in accordance with the *for* inside ***sub\_4015F7()***). We write this code (***decrypt\_memory.c***):

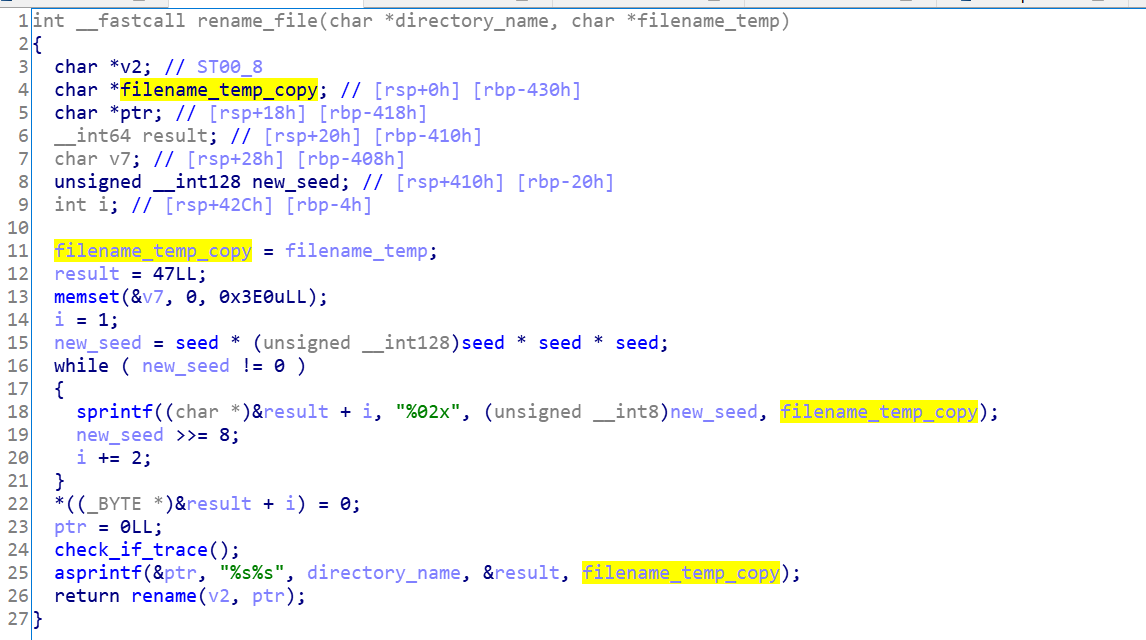
 

We get ***patched\_asg1***. And this is what’s inside the new ransomware:





So we can rename ***sub\_401842()*** to ***rename\_file()*** (*aux2.i64* – I didn’t rename the other functions in this IDA file). After cleaning the code:



The algorithm creates a new seed that is the (original one)4. We read this new seed until 0. ***Result[i]*** = 1 byte from seed (The "02" part in "%02x" specifies the minimum width of the field to be printed, which means that if the resulting string is less than two characters wide, it will be padded with leading zeros to make it two characters wide. The "x" part specifies that the integer value should be printed in lowercase hexadecimal format.). After that, we move to the next byte. ***Result*** is the new name for the file.

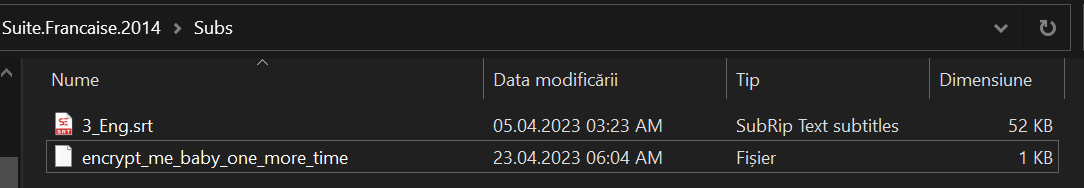
Based on this, we can conclude that the name of the file gives us the original seed. In order to find it, take the name of the file, reverse it (be aware of hex), turn into decimal the final value, then double square it and we get the original seed that can be used for decrypting the file.

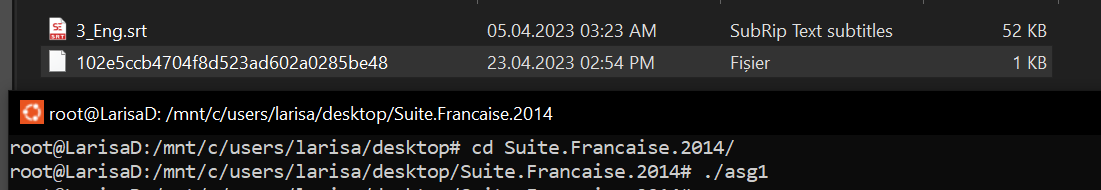
* *Create a program/script that decrypts any given encrypted file including the target file in the archive. (10p)*

First, we find the original seed, just the way we described it above. We take the encrypted file and, knowing that „***fmi\_re\_course***” is hard-coded inside it, we split the file and we know that the first part is the content of the original file and the second part is the name of the original file (above there is a detailed description of how this is achieved). Decrypt it using ***byte – rand()***. ([Source](https://stackoverflow.com/questions/70300623/python-ctypes-time0-and-c-time0))

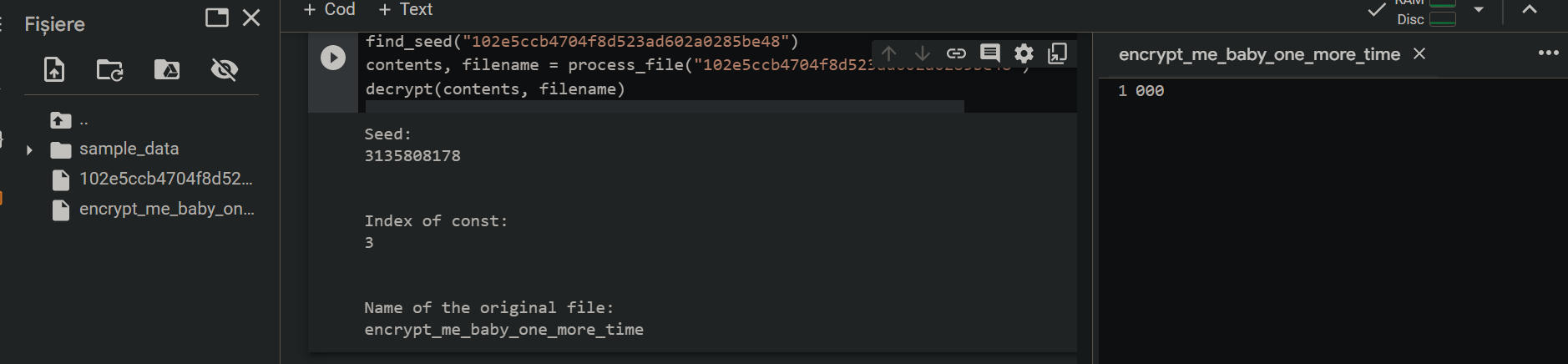
The script for decryption is *decrypt\_files.py*

Let’s do a test to see if it works:





Now, we apply the decryption algorithm:



And it works just fine. So let’s apply it on the file that was provided (we used Google Colab, in order to move faster):

