Project 1

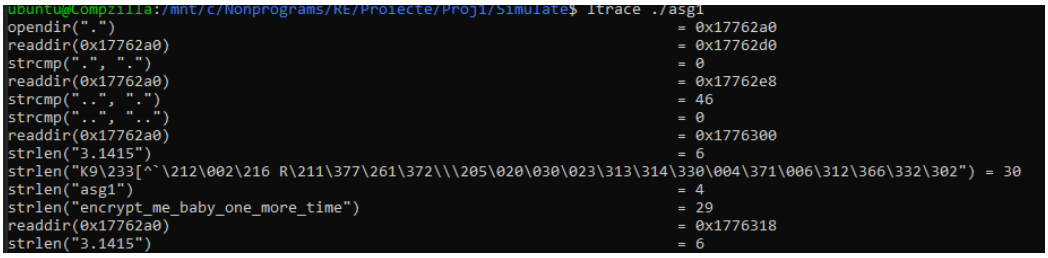
The archive (https://pwnthybytes.ro/unibuc_re/asg1-files.zip) password is **infected** and has the following contents:

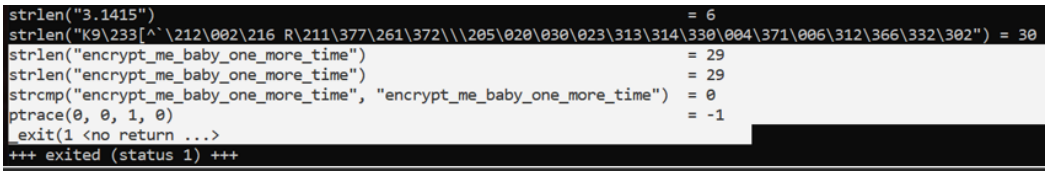
- asg1 - the binary you will analyze
- c19cf21d23c2a054462451047b202711 - the encrypted file you have to recover

1. The binary searches for files with a certain pattern and only encrypts those that match. Find out what the pattern is. (20p)
2. Describe how the encrypted files are internally structured (what bytes are written in the encrypted files and how the encryption is done). (50p)

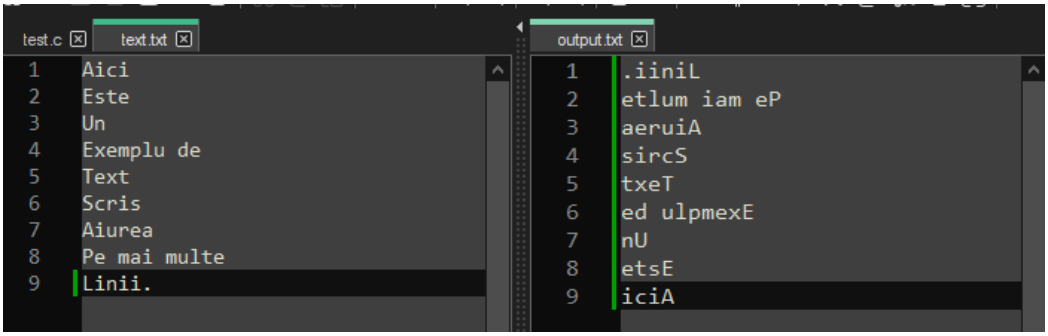
---------- 1 ----------

- I first trued to analyse the file in IDA
- I observed that the function *sub_401C7F* starts from current directory ("."), open recursively the other directories in the current one and initialize a pointer with the path of the files in the directory . I will rename this function to *recurse_parse*
- the function readdir returns a pointer to a dirent structure representing the next directory entry which is saved in v4, v4 will have more attributes that will exploited:
    1. v_type is the type of the open dir
        1. DT_UNKNOWN, which is = 0 , in this case the first comparison (if) breaks the while and return the value v1 (directory stream)
        2. DT_DIR, which is = 4, in this case the second comparison will continue more comparisons (the second attribute that will be explained) and will call recursively the function
    2. v_name is the null terminated filename which is used to verify if it contains
- The *while* statement contains the logistic of finding the files, so until this point, if the type of the pointer v4 is a directory, the path is saved in ptr pointer and then the function is recursively called with the path, if the file is not a directory, the *sub_401BA5*((__int64)a1, (__int64)v4->d_name); is called
- I the analyse the function *sub_401BA5* which is called with the parameters: directory path in a1 and file name in a2. I observed that there is a char vector (I transformed the variables into a vector) of chars which is initialised with the string v = "K9[^` R\"
- In this function there are called more functions:

- *sub_4014CC*("3.1415", v5, (__int64)&v3); -> def -> __int64 __fastcall sub_4014CC(const char *a1, const char *a2, __int64 a3)
  - After analysing with ltrace the behaviour I observed the string K9\233[^`\212\002\216R\211\377\261\372\\\205\020\030\023\313\314\330\004\371\006\312\366\332\302 which is obtained as the result of this function, after decrypting, the output is "encrypt_me_baby_one_more_time created using the calls of the functions:
    - *sub_4012DA*(a1, (__int64)&v5); -> def -> __int64 __fastcall sub_4012DA(const char *a1, __int64 a2)
    - *sub_4013B3*((__int64)&v5, a2, v3); -> __int64 __fastcall sub_4013B3(__int64 a1, const char *a2, __int64 a3)
  - the scope of the function is to create this string and save it in v3 so I will call this function *encrypt_me_string*
  - 

- *sub_40152C*((const char *)a2, v4); -> _BOOL8 __fastcall sub_40152C(const char *a1, const char *a2)
  - is a comparison function which return a result based on comparing strings (true if the last strlen(second string) characters are equal)
  - I renamed it *custom_compare*
  - This is the functions which determines the files which will be encrypted by comparing a2 (which is the file name) and &v3[1] (which is the key string without the first character)
- if the result is positive (true) - meaning that the files ending in the substring in v3 (encrypt_me_baby_one_more_time), implying the length to be >= v3 - than the *sub_4019D2*(a1, a2); is called and the sleep function after
  - As a test I created a file respecting the rules and I ran the malware in the directory; as the result:
  - 

- If the result is false, I think the function of *sub_4019D2* is the logistic of encryption. After analysing, I observed that in this function there are also file renamed, so I will modify its name into *rename_files*
  - The most calls are of the function *sub_401593* and after analysing it, I observed it calls ptrace, which "observe and control the execution of another process [...] and examine and change the trace's memory and registers.It is primarily used to implement breakpoint debugging and system call tracing ". I will rename this functions as *trace*
  - A random seed is generated using the srand function and it is stored in seed variable
  - A filename is declared and modified with asprintf (https://docs.oracle.com/cd/E36784_01/html/E36874/asprintf-3c.html). Calling asprintf, the result is that in the filename will be the name of the encrypted file, composed of the path of the original file, concatenated with v2 and "_temp". v2 is not initialised int the function but it can be the name of the original file without extension or the string at the address allocated for the variable.
  - After that, the *sub_401E00* function is called with the name of the file and a buffer. Its role is to store some information about the file identified by filename variable into the stat_buf buffer and then in v7 is stored the size of the buffer.
  - Some lines down, the file is open in write mode, the function *sub_40169A* is called have as parameters the steam (containing information stored in the original file),the size of the buffer, the filename and the steam open file opened in write mode.
  - The purpose of the function *sub_4015F7* called next is to modify the permissions of the memory region in order to make *rename_files* function writable and executable. It then XORs each byte of the function *word_401842* with the value 0x42 in order to obfuscate its content.
  - Then the streams are closed, the temporary (_temp) files are deleted and the memory is free (https://man7.org/linux/man-pages/man2/unlink.2.html)
- I will analyse next the function *sub_40169A* which seems to manage the encryption part so I will rename it as *encryption*
  - The first operation made is to set the cursor of the original file at the current position of the file pointer, with offset -1LL.

- In every iterations, ptr stores one byte read from the original file and the pointer in the original file is moved using fseek, at the beggining of the file (last parameter 1 = SEEK_SET) with the offset -2LL. every step, is added in a new pseudo-random value generated with rand() function and the result is written in the new file (with _temp in the name). This step is repeated buf_size times, buf_size being stored in a2. To resume, the fseek part has the scope of inversing the text in means that the text is read from the left to right, from up to down and the output is the same text but in the inverse order - from left to right, down to up (an example of what I am trying to say is in the picture below). Upon the scrambled text, there is added a step which adds a pseudo-random number to each character, making it unreadable.
- 

- In the next step, a for is simulating, the stop condition being verified using the variable v4, which stores the length of the string processed. At every step, in the new file is written the character from the string "fmi_re_course" in clear (without computing).
- The issue with this step seems to be that the variable storing the string is overflown because it is declared with the length 8
- In the last step, the parameter string a3 is traversed, on each character a random number is added and then every character is written in the output file, byte by byte.

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

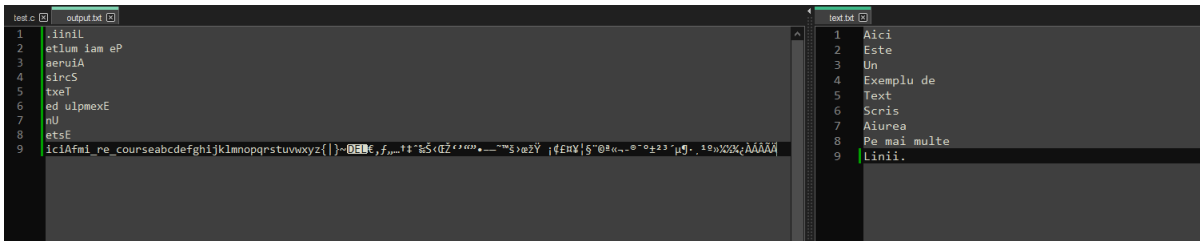
- The *rename_files* function takes two parameters: the first parameter is a directory path, and the second parameter is a filename. The function renames every file in the directory by appending "_temp" to the original filename and then encrypts the contents of the file using the *encryption* function. After encryption, the function renames the original files to the encrypted filename and deletes the temporary file. More details about how the encryption works are presented above, but to summarize, the original file is scrambled, each bytes is added a pseudo-random value and the original text is concatened with fmi_re_course string (whoch is devided intro substrings) + the value of a string modified using the same calculation with rand function.
- Theoretically, we delimitate the encrypted original text by the a3 string added by following the clear text "fmi_re_course", which was not modified before adding. We know that in the first time, before that delimiter, there is the original text, encrypted and after it it is a3 encrypted. To illustrate this, I wrote a similar encryption function (ignoring the a3 parameter in the original implementation and starting from a fixed input with length 61):

```
FILE * f = fopen ("text.txt", "r");
FILE * out = fopen ("output.txt", "w");
int i, v4, v5, result;
char *v2, *v8, *ptr, v11[18], v9;

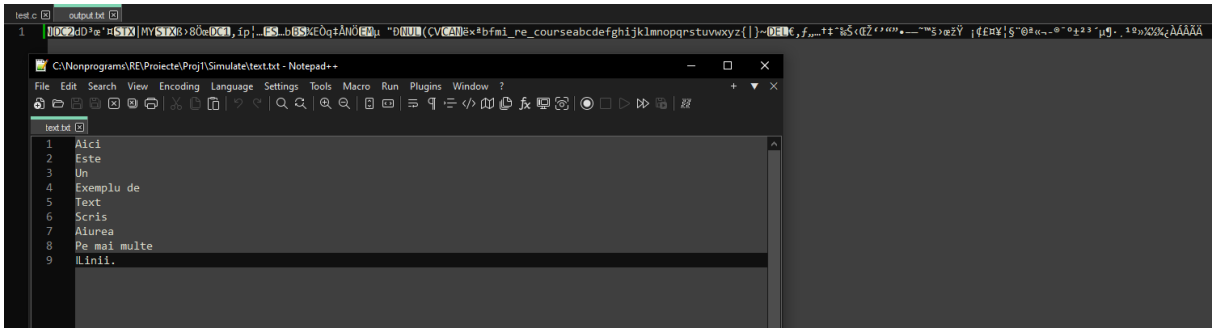
fseek (f, -1LL, 2);
for (i = 0; i < 61; i ++){
        fread (&ptr, 1uLL, 1uLL, f);
        fseek (f, -2LL, 1);
        //ptr += rand();
        fwrite(&ptr, 1uLL, 1uLL, out);
}
strcpy(v11, "fmi_re_course");
for ( i = 0; ; ++i ){
        v4 = i;
        if ( v4 >= strlen(v11) )
          break;
        fwrite(&v11[i], 1uLL, 1uLL, out);
}
for ( i = 0; i < 100 ; i++ ){
        char a = i + 'a';
        fwrite( &a, 1uLL, 1uLL, out);
}
/*
// prove fseek scope -> inverse text
fseek (f, -1LL, 2);
```

```
        for (i = 0; i < 61; i ++){
                fread (&ptr, sizeof(char), 1uLL, f);
                fseek (f, -2LL, 1);
                fwrite(&ptr, sizeof(char), 1uLL, out);
        }
        */
        fclose (f);
        fclose (out);
        printf ("The result is = %d\n", result);
        return 0;
```

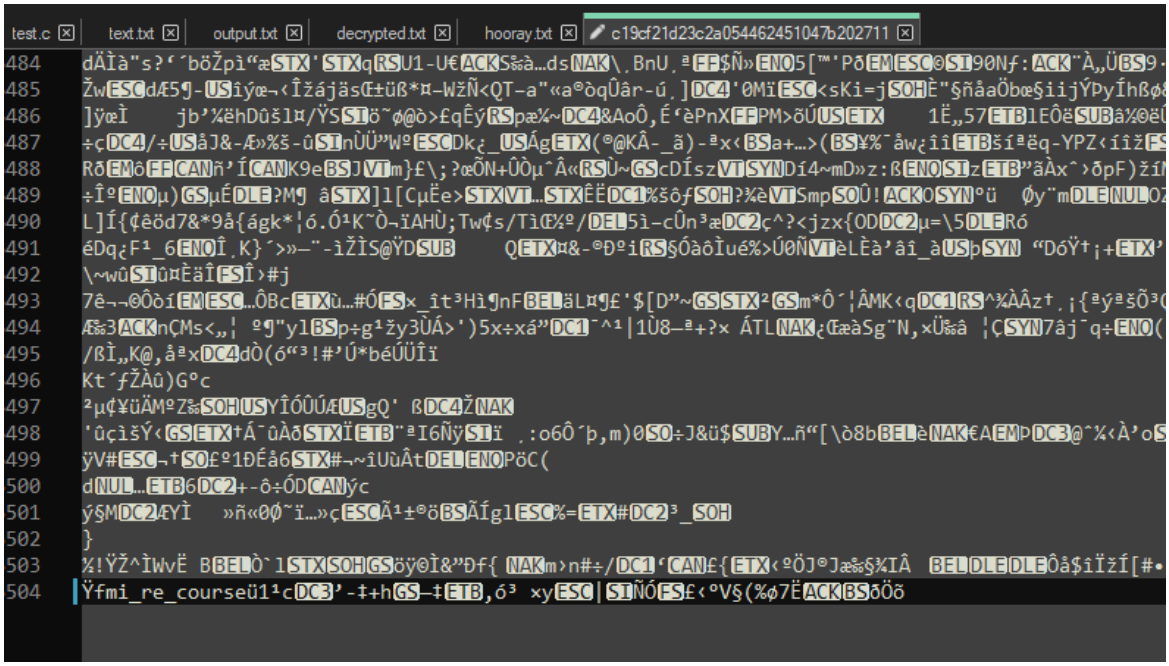
- without ptr += rand() line



- with ptr += rand() line



- So, we break the output (we delete the unnecessary characters) in order to obtain all the text before the delimiter. We now have to decrypt this one.

- 

-
- In the first place, we have to resolve the scramble, which means we have to put the random characters in the correct order. For that I will use this function:

```
fseek (f, -1LL, 2);
for (i = 0; i < 61; i ++){
        fread (&ptr, sizeof(char), 1uLL, f);
        fseek (f, -2LL, 1);
        fwrite(&ptr, sizeof(char), 1uLL, out);
}
```

  which is exactly the function that scrambled the characters in the first place. The function is symmetric so f(f(a)) will be a
- The only thing that we have to resolve now is the ptr += rand() which is applied to every byte (character).
  - Firstly, we know that rand() is a pseudo-random generator dependent on the implementation, so I searched the stdlib library version used. To do so I run objdump on the file obtaining:

- ```
  ubuntu@Compzilla:/mnt/c/Nonprograms/RE/Proiecte/Proj1/Simulate$ objdump -p asg1

  asg1:     file format elf64-x86-64

  Program Header:
      PHDR off    0x0000000000000040 vaddr 0x0000000000400040 paddr 0x0000000000400040 align 2**3
           filesz 0x0000000000000268 memsz 0x0000000000000268 flags r--
    INTERP off    0x00000000000002a8 vaddr 0x00000000004002a8 paddr 0x00000000004002a8 align 2**0
           filesz 0x000000000000001c memsz 0x000000000000001c flags r--
      LOAD off    0x0000000000000000 vaddr 0x0000000000400000 paddr 0x0000000000400000 align 2**12
           filesz 0x0000000000000910 memsz 0x0000000000000910 flags r--
      LOAD off    0x0000000000001000 vaddr 0x0000000000401000 paddr 0x0000000000401000 align 2**12
           filesz 0x0000000000000e19 memsz 0x0000000000000e19 flags r-x
      LOAD off    0x0000000000002000 vaddr 0x0000000000402000 paddr 0x0000000000402000 align 2**12
           filesz 0x0000000000000378 memsz 0x0000000000000378 flags r--
      LOAD off    0x0000000000002e08 vaddr 0x0000000000403e08 paddr 0x0000000000403e08 align 2**12
           filesz 0x00000000000002d8 memsz 0x00000000000002e8 flags rw-
   DYNAMIC off    0x0000000000002e20 vaddr 0x0000000000403e20 paddr 0x0000000000403e20 align 2**3
           filesz 0x00000000000001d0 memsz 0x00000000000001d0 flags rw-
      NOTE off    0x00000000000002c4 vaddr 0x00000000004002c4 paddr 0x00000000004002c4 align 2**2
           filesz 0x0000000000000044 memsz 0x0000000000000044 flags r--
  EH_FRAME off    0x0000000000002030 vaddr 0x0000000000402030 paddr 0x0000000000402030 align 2**2
           filesz 0x00000000000000a4 memsz 0x00000000000000a4 flags r--
     STACK off    0x0000000000000000 vaddr 0x0000000000000000 paddr 0x0000000000000000 align 2**4
           filesz 0x0000000000000000 memsz 0x0000000000000000 flags rw-
     RELRO off    0x0000000000002e08 vaddr 0x0000000000403e08 paddr 0x0000000000403e08 align 2**0
           filesz 0x00000000000001f8 memsz 0x00000000000001f8 flags r--

  Dynamic Section:
    NEEDED               libc.so.6
    INIT                 0x0000000000401000
    FINI                 0x0000000000401e10
    INIT_ARRAY           0x0000000000403e08
    INIT_ARRAYSZ         0x0000000000000008
    FINI_ARRAY           0x0000000000403e10
    FINI_ARRAYSZ         0x0000000000000008
    GNU_HASH             0x0000000000400308
    STRTAB               0x0000000000400598
    SYMTAB               0x0000000000400328
    STRSZ                0x00000000000000ca
    SYMENT               0x0000000000000018
    DEBUG                0x0000000000000000
    PLTGOT               0x0000000000404000
    PLTRELSZ             0x0000000000000228
    PLTREL               0x0000000000000007
    JMPREL               0x00000000004006e8
    RELA                 0x00000000004006b8
    RELASZ               0x0000000000000030
    RELAENT              0x0000000000000018
    VERNEED              0x0000000000400698
    VERNEEDNUM           0x0000000000000001
    VERSYM               0x0000000000400662

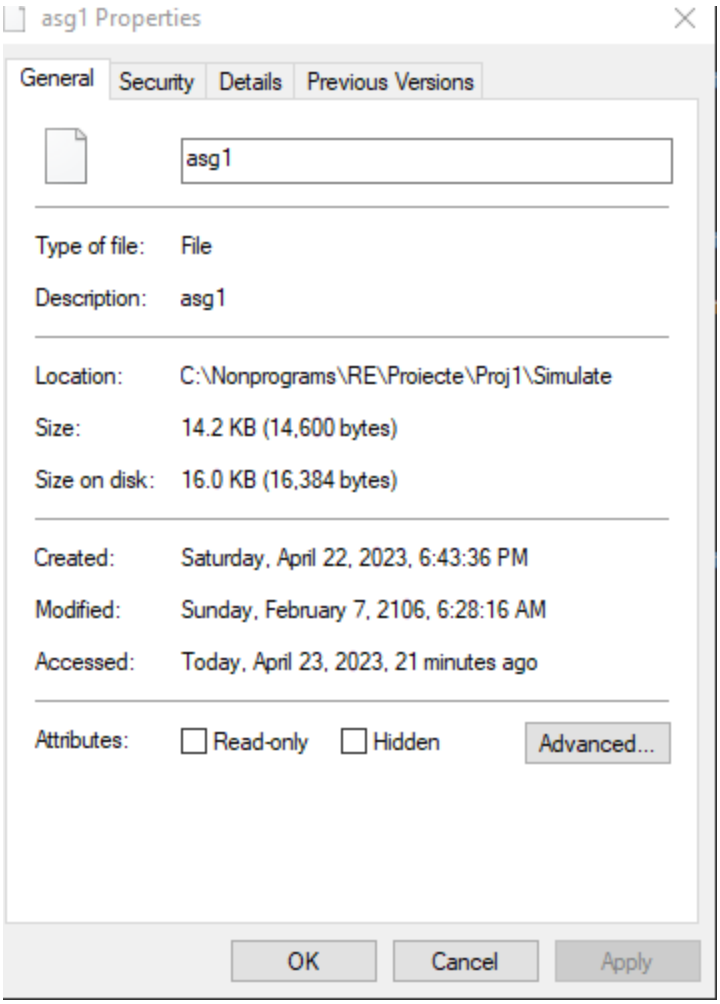
  Version References:
    required from libc.so.6:
      0x09691a75 0x00 02 GLIBC_2.2.5
  ```
- Next, I searched the implementation of rand() in glibc_2.2.5
  - rand: https://sourceware.org/git/?p=glibc.git;a=blob;f=stdlib/rand.c;h=d8458da970c05834ddd187e971c2f024d77e25f2;hb=7d04edce9211ccd0ae06cd19d3cc2098d8891935
  - __random: https://sourceware.org/git/?p=glibc.git;a=blob;f=stdlib/random.c;h=afd0a1a51693a2ec612b7630fd99ff9d3cbf9cce;hb=7d04edce9211ccd0ae06cd19d3cc2098d8891935
  - __random_r: https://sourceware.org/git/?p=glibc.git;a=blob;f=stdlib/random_r.c;h=00ba44bfffaf30965374efca416da13d131d6fa0;hb=7d04edce9211ccd0ae06cd19d3cc2098d8891935
  ```

- Observing the implementation, __random_r is dependent on unsafe_state, static struct variable, which is dependent on randtbl variable which is a static array. So, in order to obtain the same random numbers, we can use the same function because, keeping in mind the implementation, it is strictly dependent on the constant implemented in the library and the seed declared in the program. In this situation, at every restart of the program, if the seed can be computed the same, the generated array of pseudo-ransom numbers will be the same. Keeping this in mind, we can generate an array will the number of values = with the encrypted bytes in order to use it as the keys in decryption. The array needs to be reverted so that, in decryption, we use the same way to parse the bytes and replace only the computation: instead of adding to the pointer a random number, we will substract the corresponding value in the previously generated array.
- The seed is the only thing that remains to be figured out in order to decrypt. I observed in IDA that the seed is initialised:

```
seed = (unsigned __int64)time(0LL) ^ 0xDEADBEEF;
```

The value after ^ operation is a constant, but sadly the time function depends on the time the program was run on (https://man7.org/linux/man-pages/man2/time.2.html). The time function computes the time from 1970-01-01 00:00:00 +0000 (UTC) until the program is run, in seconds

- Looking in the properties, I found some strange info about the date when the file was created

- 

  - I transformed the date February 7, 2106, 6:28:16 AM in seconds, the 0 being the 1970-01-01 00:00:00 +0000 (UTC), using https://www.epochconverter.com/ and I obtained 4294967296. I transformed this number in hexa and I xored with the constant in order to obtain the seed.
4. Create a program/script that decrypts any given encrypted file including the target file in the archive. (10p) - decryption.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdint.h>
#include <time.h>

int decryption(){
        char filename[1001];
        printf ("Enter the encrypted filename: ");
        scanf ("%s", &filename);
```

```
        FILE * f = fopen (filename, "rb");
        FILE * out = fopen ("original.txt", "wb");

        if ( f == NULL || out == NULL)
            return 1; // error

        int i, dim;
        char *ptr, v11[18];
        fseek (f, 0LL, SEEK_END);
        dim = ftell (f);
        int k[dim];
        fclose (f);
        long long int diff = 0x100000000 ^ 0xDEADBEEF; // 0x100000000 = 4294967296
        srand (diff);
        for (i = 0; i < dim; i++)
                k [dim - i - 1] = rand();
        f = fopen (filename, "rb");
        fseek (f, -1LL, 2);
        for (i = 0; i < dim; i ++){
                fread (&ptr, 1uLL, 1uLL, f);
                fseek (f, -2LL, 1);
                ptr -= k[i];
                fwrite(&ptr, 1uLL, 1uLL, out);
        }
        fclose (f);
        fclose (out);
        printf ("The file was decrypted! :) \n");
        return 0;
}
```