Reverse Engineering
Ransomware Analysis

Dilirici Radu, 510

Link to the problem:
https://cs.unibuc.ro/~crusu/re/Reverse%20Engineering%20(RE)%20-%20Project%200x01.pdf
Link to the binaries:
https://pwnthybytes.ro/unibuc_re/asg1-files.zip

This is a trimmed down and prettified presentation of the analysis that took place. In this process I used **IDA Pro 7.0** on Windows 11 and **GNU gdb 13.1** on Kali Linux.

Task 1:

- The binary searches for files with a certain pattern and only encrypts those that match. Find out what the pattern is.

Starting with static analysis in **IDA Pro** we can observe that the **main()** function only calls one other function with the argument **"."**. Therefore, we investigate **sub_401C7F**.

```
__int64 __fastcall main(__int64 a1, char **a2, char **a3)
{
  sub_401C7F(".");
  return 0LL;
}
```

In **sub_401C7F** we can observe that there happens some kind of work with directories and files. We can also see that the only other method called here is **sub_401BA5**.

```
else
{
  sub_401BA5((__int64)a1, (__int64)v4->d_name);
}
```

Entering **sub_401BA5** we see many variables that are initialized. From the looks of it we can assume that this is actually a string. After transforming it to an array of characters it looks a little better, but the string itself doesn't have any meaning.

```
unsigned int __fastcall sub_401BA5(__int64 a1, __int64 a2)
{
  unsigned int result; // eax
  char v3; // [rsp+10h] [rbp-60h]
  _BYTE v4[7]; // [rsp+11h] [rbp-5Fh]
  char v5[30]; // [rsp+50h] [rbp-20h]

  v5[0] = 'K';
  v5[1] = '9';
  v5[2] = '�';
  v5[3] = '[';
  v5[4] = '^';
  v5[5] = '`';
  v5[6] = '�';
  v5[7] = '\x02';
  v5[8] = '�';
  v5[9] = ' ';
  v5[10] = 'R';
  v5[11] = '�';
  v5[12] = '\xFF';
  v5[13] = '�';
  v5[14] = '�';
  v5[15] = '\\';
  v5[16] = '�';
  v5[17] = '\x10';
  v5[18] = '\x18';
  v5[19] = '\x13';
  v5[20] = '�';
  v5[21] = '�';
  v5[22] = '�';
  v5[23] = '\x04';
  v5[24] = '�';
  v5[25] = '\x06';
  v5[26] = '�';
  v5[27] = '�';
  v5[28] = '�';
  v5[29] = '�';
  sub_4014CC("3.1415", v5, &v3);
  result = sub_40152C(a2, v4);
  if ( result )
  {
    sub_4019D2(a1, a2);
    result = sleep(1u);
  }
  return result;
}
```

One interesting function here is **sub_40152C** because the rest of the method depends on its return value. At this point I also started using **gdb** in a Linux environment and observed that this function always returned **0** and the program did nothing, or at least nothing noticeable.

The **sub_40152C** function seems to work with two strings and it checks that the second one is a suffix to the first one.

```
_BOOL8 __fastcall sub_40152C(const char *a1, const char *a2)
{
  int v3; // [rsp+18h] [rbp-8h]
  int v4; // [rsp+1Ch] [rbp-4h]

  v4 = strlen(a1);
  v3 = strlen(a2);
  return v4 >= v3 && !strcmp(&a1[v4 - v3], a2);
}
```

Inspecting this function with the debugger we notice a string that stands out in the memory of the program. That string is **“encrypt_me_baby_one_more_time”**.

```
[----------------------------------registers-----------------------------------]
RAX: 0x4
RBX: 0x7fffffffded8 --> 0x7fffffffe24f ("/media/sf_vm-shared/asg1-files/asg1")
RCX: 0x333
RDX: 0xdfd0
RSI: 0x7fffffffdd11 ("encrypt_me_baby_one_more_time")
RDI: 0x405333 --> 0x50031677361
RBP: 0x7fffffffdcf0 --> 0x7fffffffdd70 --> 0x7fffffffddb0 --> 0x7fffffffddc0 --> 0x2
RSP: 0x7fffffffdcd0 --> 0x7fffffffdd11 ("encrypt_me_baby_one_more_time")
RIP: 0x401548 (mov    DWORD PTR [rbp-0x4],eax)
R8 : 0xf8c0
R9 : 0x3
R10: 0x7ffff7ddc360 --> 0x10001a000070bc
R11: 0x7ffff7e76b50 (<__strlen_sse2>:   pxor   xmm0,xmm0)
R12: 0x0
R13: 0x7fffffffdef0 --> 0x7fffffffe27c ("COLORFGBG=15;0")
R14: 0x0
R15: 0x7ffff7ffd020 --> 0x7ffff7ffe2e0 --> 0x0
EFLAGS: 0x202 (carry parity adjust zero sign trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x40153c:    mov    rax,QWORD PTR [rbp-0x18]
   0x401540:    mov    rdi,rax
   0x401543:    call   0x401090 <strlen@plt>
=> 0x401548:    mov    DWORD PTR [rbp-0x4],eax
   0x40154b:    mov    rax,QWORD PTR [rbp-0x20]
   0x40154f:    mov    rdi,rax
   0x401552:    call   0x401090 <strlen@plt>
   0x401557:    mov    DWORD PTR [rbp-0x8],eax
[------------------------------------stack-------------------------------------]
0000| 0x7fffffffdcd0 --> 0x7fffffffdd11 ("encrypt_me_baby_one_more_time")
0008| 0x7fffffffdcd8 --> 0x405333 --> 0x50031677361
0016| 0x7fffffffdce0 --> 0xe88c14e161b73035
0024| 0x7fffffffdce8 --> 0xe556d0f46836f5d2
0032| 0x7fffffffdcf0 --> 0x7fffffffdd70 --> 0x7fffffffddb0 --> 0x7fffffffddc0 --> 0x2
0040| 0x7fffffffdcf8 --> 0x401c5b (test   eax,eax)
0048| 0x7fffffffdd00 --> 0x405333 --> 0x50031677361
0056| 0x7fffffffdd08 --> 0x40202a --> 0x1b0100002e2e002e
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value

Breakpoint 3, 0x0000000000401548 in ?? ()
gdb-peda$
```

After some testing I confirmed that this is actually the string that is checked to be a suffix for the file to be encrypted. I placed several such files that satisfy this condition on the machine and noted that only the ones placed in the program's folder or in a nested folder were being encrypted.

Analyzing the malware further I reached these conclusions:

- The function called by **main** (**sub_401C7F**) iterates through the files in the current location. If it finds a directory it navigates it recursively. We'll call this function **iterate_files**.
- The sub_401BA5 function is responsible for checking if the found file satisfies the pattern required for encryption. It firstly initializes a string with strange characters. Because this string has the same length as the suffix (**"encrypt_me_baby_one_more_time"**) and because there is a method (**sub_4014CC**) called with it as its parameter, I assumed that it's an encrypted version of the suffix. This means that **sub_4014CC** is responsible for decrypting it to the real string.

So, to this point, the flow of the malware is:

1. Search for all the files in the current directory and its subdirectories, starting from where the program is called (this is why the **iterate_files** function is called with the **"."** argument). If the program is in folder **A**, but we start it via the command line from folder **B**, it will start searching for files in the **B** folder.
2. For each file, compute the secret string and check if it's a suffix to the file name.
3. If it satisfies this condition, enter the **sub_4019D2** function with the (relative) path and name of the file as arguments. We'll call this function **encrypt_file**.

```
  decrypt_secret_ending((__int64)"3.1415", (__int64)secret_ending, (__int64)&v3);
  result = string_ends_with(file_name, encrypt_me_baby_name);
  if ( result )
  {
    encrypt_file((const char *)file_path, file_name);
    result = sleep(1u);
  }
  return result;
}
```

All the changes in the code can be seen in the final **IDA** file.

Task 1 answer:

The program searches for files in the current directory and all its subdirectories. Then, it encrypts only the ones that have the suffix **"encrypt_me_baby_one_more_time"**.

Task 2:

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

While analyzing the program in the Linux environment, I noticed that it always crashes when running it in debug mode. This means that there is some anti-debugging mechanism inside the program.

```
gdb-peda$ run
Starting program: /home/kali/Documents/ransomware/asg1
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
[Inferior 1 (process 43250) exited with code 01]
Warning: 'set logging off', an alias for the command 'set logging enabled', is deprecated.
Use 'set logging enabled off'.
```

Indeed, entering the **encrypt_file** function we notice a method that is called quite a few times (6 to be precise, and it's also called in lots of other places). That function is **sub_401593**. If we look inside it we discover that it uses **ptrace** to avoid debugging. This is because a program can only be traced by one process at a time. We'll call this function **exit_if_debugging**.

Changing the response of **ptrace** while doing the dynamic analysis, so the program would "think" that it's not being traced by someone else, would not be a viable solution, as we'd have to do this lots and lots of times. To counter this, I applied a patch to the binary and skipped the check altogether.

The original program was going through a loop 1000 times, but only checked for an external debugger only the first time. This was probably done to make it harder to bypass the mechanism dynamically.

```
void exit_if_debugging()
{
  signed int i; // [rsp+Ch] [rbp-4h]

  for ( i = 0; i <= 999; ++i )
  {
    if ( checked != 1 && ptrace(0, 0LL, 1LL, 0LL) == -1 )
      _exit(1);
    checked = 1;
  }
}
```

Taking a look in the assembly code we see that i is compared to **999** (**0x3E7** in HEX) and if it's less or equal (**jle**) it jumps to the body of the loop. I changed this instruction to **jg**, so it does the exact opposite. Now, it will execute the inside of the loop only if **i > 999**, which will be false from the first iteration (since **i** starts as **0**). This results in not entering the loop and never checking for the debugger.

Before the patch:

```
loc_4015EB:
cmp     dword ptr [rbp-4], 3E7h
jle     short loc_4015A4
```

After the patch:

```
loc_4015EB:
cmp     dword ptr [rbp-4], 3E7h
jg      short loc_4015A4
```

And the new corresponding pseudocode:

```
void exit_if_debugging()
{
  signed int i; // [rsp+Ch] [rbp-4h]

  for ( i = 0; i > 999; ++i )
  {
    if ( checked != 1 && ptrace(0, 0LL, 1LL, 0LL) == -1 )
      _exit(1);
    checked = 1;
  }
}
```

The encrypt_file method opens two files. One in read mode and one in write mode. **IDA** used just one 128 bit variable for both file names. These should be two separated strings, but **IDA** combined them because they were placed successively.

```
__int128 filename;
```

```
filename = 0uLL;
asprintf((char **)&filename + 1, "%s/%s", a1, a2, a2);
exit_if_debugging();
asprintf((char **)&filename, "%s/%s_temp", a1, v3);
```

Although, it uses an array instead of two variables, this would be a better interpretation of the code:

```
char *full_file_names[2];
```

```
full_file_names[0] = 0LL;
asprintf(&full_file_names[1], "%s/%s", file_path, file_name, file_name);
exit_if_debugging();
asprintf(full_file_names, "%s/%s_temp", file_path, original_file_name);
```

Using dynamic analysis and placing several breakpoints inside this function we notice that the encryption takes place in two steps. First, a new file called **<filename>_temp** is created, where **<filename>** is the name of the original file. Then, the malware writes to this new file the encrypted contents of the original file. Lastly, the temporary file is renamed. This could not have been observed only by running the program, as the whole process is usually instantaneous.

Based on our current knowledge, this is a summary of the **encrypt_file** function:

1. A random seed is generated based on the current time and a constant (**0xDEADBEEF**). We don't know what it's used for at this point.

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

2. The function **xstat** is called for the original file. This provides information about the file. Only the file size is extracted from this call. Actually, another function named **sub_401E00** is called, which is a wrapper for **xstat**. I named it **get_file_status**.

   ```
   get_file_stats(full_file_names[1], &stat_buf);
   file_size = stat_buf.st_size;
   ```

3. The original file is opened in read mode (**full_file_names[1]** in the previous picture) and the temporary file is opened in write mode (**full_file_names[0]**, having the **<filename>_temp** name).
4. A function named **sub_40169A** is called.
5. The two files are closed. At this point the temporary file already has all the content inside it. Again, I observed this by placid breakpoints inside the debugger and checking the status of the files. From this we can deduce that **sub_40169A** handles the encryption of the file.
6. The **sub_4015F7** method is called.
7. The method **word_401842** is called. When opening this function inside **IDA**, we can't actually read its contents. Instead, we can only see some random data. But, by analyzing it dynamically we can observe that after the call, the temporary file no longer exists and instead a new file with a cryptic name

appears. For now I renamed it to **secret_function**.

```
.text:0000000000401841
.text:0000000000401841 ; ---------------------------------------------------------------------------
.text:0000000000401842 ; __unwind {
.text:0000000000401842 secret_function dw 0A17h, 0A7CBh, 0C30Ah
.text:0000000000401842                                         ; CODE XREF: encrypt_file+185↓p
.text:0000000000401842                                         ; DATA XREF: decode_encryption_function+8↑o
.text:0000000000401848                 dq 0FFCB0A42424672AEh, 92F7CB0ABDBDB99Ah, 0B9B2C7850ABDBDB9h
.text:0000000000401848                 dq 0CF0A4242426DBDBDh, 4242FABDBDB9BAD7h, 0A4242423EFB4242h
.text:0000000000401848                 dq 0BE0785E90AB195CBh, 6A1C47C942424243h, 0A207CB0A82CB4242h
.text:0000000000401848                 dq 42424242AA07850Ah, 7ED4D0AAA07C90Ah, 0AA07C90A80CB0AA2h
.text:0000000000401848                 dq 4ECF0AA207ED4D0Ah, 27B50AA207C90A40h, 0A88CB0A93430AA2h
.text:0000000000401848                 dq 4D0A8CCB0AA20FC9h, 4D0AAA0FC90AB0EDh, 27B50AB3430A8AEDh
.text:0000000000401848                 dq 0A88CB0A93430AA2h, 4D0A8CCB0AAA0FC9h, 4D0AA20FC90AB2EDh
.text:0000000000401848                 dq 27B50AB3430A88EDh, 0A88CB0A93430AA2h, 0AAA17CB0AA207CBh
.text:0000000000401848                 dq 0A9AA17CB0AA207CBh, 82F54DA207C90A0Fh, 210ABE17C982F44Dh
.text:0000000000401848                 dq 0BDBDB9B2CFCF0A90h, 77CF0A80CB93430Ah, 0FA8DCB0A42424492h
.text:0000000000401848                 dq 0BDBA5DAA42424242h, 17C90AA207C90ABDh, 830A4A92EE4D0AAAh
.text:0000000000401848                 dq 0CB0AA207CB0A4AA8h, 0C90A40BE07C1AA17h, 0C70AA207490AAA07h
.text:0000000000401848                 dq 0DA0ABE07C9E43782h, 42BDBDB9B247C684h, 42BDBDB9AAC7850Ah
.text:0000000000401848                 dq 42424242FA424242h, 0CFCF0ABDBDBE44AAh, 9AD7C90ABDBDB9B2h
.text:0000000000401848                 dq 0B9AAC7CF0ABDBDB9h, 42442277CF0ABDBDh, 424242FA85CB0A42h
.text:0000000000401848                 dq 0C90ABDBDB528AA42h, 0C7C90ABDBDB9AAD7h, 0A94CB0ABDBDB992h
.text:0000000000401848                 dq 0D2BDBDB5C3AA85CBh
.text:00000000004019D0                 db 8Bh, 81h
.text:00000000004019D0 ; } // starts at 401842
.text:00000000004019D2
.text:00000000004019D2 ; =============== S U B R O U T I N E =======================================
.text:00000000004019D2
.text:00000000004019D2 ; Attributes: bp-based frame
.text:00000000004019D2
.text:00000000004019D2 encrypt_file    proc near               ; CODE XREF: process_file+C8↓p
.text:00000000004019D2                                         ; DATA XREF: decode_encryption_function+13↑o
.text:00000000004019D2
```

8. The **sub_4015F7** function is called again.
9. The original file is deleted. This is an observation of dynamic analysis.

The **sub_40169A** function seems to be the target of our current task so we'll take a look at it. We'll also rename it to **encrypt_to_temp_file**.

Setting a breakpoint in **gdb** just before calling it, we actually get a hint for the passed parameters.

```
[──────────────────────────────────code──────────────────────────────────]
   0x401b07:    mov    rsi,QWORD PTR [rbp-0x8]
   0x401b0b:    mov    rax,QWORD PTR [rbp-0x10]
   0x401b0f:    mov    rdi,rax
=> 0x401b12:    call   0x40169a
   0x401b17:    mov    eax,0x0
   0x401b1c:    call   0x401593
   0x401b21:    mov    rax,QWORD PTR [rbp-0x10]
   0x401b25:    mov    rdi,rax
Guessed arguments:
arg[0]: 0x40d3d0 --> 0x31653963fbad2488
arg[1]: 0x2a ('*')
arg[2]: 0x4053eb ("encrypt_me_baby_one_more_time")
arg[3]: 0x40d5b0 --> 0xfbad2484
[─────────────────────────────────stack─────────────────────────────────]
```

After further analyzing the code and how these parameters are used I concluded that they are, in order:

- A pointer to the original file.
- The size of the original file. In this case **0x2a**, which is **42** in decimal.
- The name of the original file.
- A pointer to the temporary file.

There are three for loops in this method, each writing some text inside the temp file. I wasn't actually able to observe these writing operations in real time because the buffer was not flushed, but these are my guesses:

1. The position of the input stream is placed one byte before the end of the file. This is done so the next read operation results in getting the last byte.

```
fseek(original_file, -1LL, 2);
```

2. The method goes through the original file byte by byte, backwards. For each read byte a random value is added to it. Then, this new byte is written to the temporary file.

```
for ( i = 0; i < original_file_size; ++i )
{
  fread(&buf, 1uLL, 1uLL, original_file);     // Read 1 byte.
  exit_if_debugging();
  fseek(original_file, -2LL, 1);              // Go back 2 bytes, as we previously read one byte.
                                              // This will ultimately result in going one byte backwards.
  exit_if_debugging();
  buf += rand();
  exit_if_debugging();
  fwrite(&buf, 1uLL, 1uLL, temp_file);
  exit_if_debugging();
}
```

3. The string **"fmi_re_couse"** is written to the temporary file.

```
strcpy(course_name, "fmi_re_course");
// Write "fmi_re_course" into the temp file
for ( i = 0; ; ++i )                          // for (i = 0; i < strlen(course_name); ++i)
{
  i2 = i;
  if ( i2 >= strlen(course_name) )
    break;
  fwrite(&course_name[i], 1uLL, 1uLL, temp_file);
}
```

   Using this information I could confirm that the first part of the encrypted file is indeed a byte to byte correspondence to the original one. I ran the malware on files of different sizes and noticed that the number of characters before **"fmi_re_couse"** was the same as the number of characters of the original file.

4. The function goes through the name of the original file. It adds random values to each of the characters and writes the new values to the temporary file, just like in the first loop.

```
for ( i = 0; ; ++i )                          // for (i = 0; i < strlen(original_file_name); ++i)
{
  i3 = i;
  result = strlen(original_file_name);
  if ( i3 >= result )
    break;
  chr = original_file_name[i];
  chr += rand();
  fwrite(&chr, 1uLL, 1uLL, temp_file);
}
```

For now, we can't possibly decrypt this new file as all its bytes are basically random.

Task 2 answer:

The encrypted file consists of three parts. The first one represents the contents of the original file, but each byte has a random value added to it. The second one is the **“fmi_re_couse”** string. The last part represents the name of the original file, but each byte has a random value added to it, just like in the first part.

Task 3:

- Figure out how the file renaming process works and describe how decryption could theoretically be done.

There are two more functions inside **encrypt_file** that we did not cover. We can't read the contents of **secret_function**, so we'll analyze **sub_4015F7**.

```
sub_4015F7(v4);
(*(void (__fastcall **)(__int64, _QWORD))secret_function)(v2, filename);
sub_4015F7(v2);
```

This method sets the access permissions to **READ + WRITE + EXECUTE** (**7** as the numerical value) to a memory region. It then traverses the memory region of **secret_function** and applies a **XOR** operation to it. The memory is read **2** bytes at a time and each chunk is **XOR**ed with **0x42** (**66** in decimal). We can see that it only affects this function because the iteration stops when reaching **encrypt_file**, which is the next function in the code of the program.

From this, we can deduce that **secret_function** is initially encrypted and this function decrypts it. It then encrypts it back after using the secret function. We'll call this function **un_hide_secret_function**.We want to apply this decryption on the original binary, so we can statically analyze the method.

To achieve this, I ran the malware in debug mode and set a breakpoint just before the secret function was called. With the method decrypted in memory, I dumped its contents to a file. I then wrote a script that replaces the encrypted memory zone with the bytes I previously dumped, i.e. the decrypted function.

```
[------------------------------------code------------------------------------]
   0x401b4a:    mov    rax,QWORD PTR [rbp-0xc8]
   0x401b51:    mov    rsi,rdx
   0x401b54:    mov    rdi,rax
=> 0x401b57:    call   0x401842
   0x401b5c:    mov    eax,0x0
   0x401b61:    call   0x4015f7
   0x401b66:    mov    rax,QWORD PTR [rbp-0xb8]
   0x401b6d:    mov    rdi,rax
Guessed arguments:
arg[0]: 0x40202a --> 0x1b0100002e2e002e
arg[1]: 0x40d3a0 ("./encrypt_me_baby_one_more_time_temp")
arg[2]: 0x40d3a0 ("./encrypt_me_baby_one_more_time_temp")
[-----------------------------------stack------------------------------------]
0000| 0x7fffffffdc10 --> 0x4053eb ("encrypt_me_baby_one_more_time")
0008| 0x7fffffffdc18 --> 0x40202a --> 0x1b0100002e2e002e
0016| 0x7fffffffdc20 --> 0x40d3a0 ("./encrypt_me_baby_one_more_time_temp")
0024| 0x7fffffffdc28 --> 0x40d370 ("./encrypt_me_baby_one_more_time")
0032| 0x7fffffffdc30 --> 0x801
0040| 0x7fffffffdc38 --> 0x80929
0048| 0x7fffffffdc40 --> 0x1
0056| 0x7fffffffdc48 --> 0x3e8000081a4
[----------------------------------------------------------------------------]
Legend: code, data, rodata, value

Breakpoint 2, 0x0000000000401b57 in ?? ()
gdb-peda$ dump memory function.out 0x401842 0x4019d2
```

I executed the script inside **IDA** and specified that it's a function. It then recognized it as a method.

```
.text:0000000000401842
.text:0000000000401842 secret_function proc near               ; (
.text:0000000000401842                                         ; [
.text:0000000000401842
.text:0000000000401842 old             = qword ptr -430h
.text:0000000000401842 var_428         = qword ptr -428h
.text:0000000000401842 ptr             = qword ptr -418h
.text:0000000000401842 var_410         = qword ptr -410h
.text:0000000000401842 var_408         = byte ptr -408h
.text:0000000000401842 var_20          = qword ptr -20h
.text:0000000000401842 var_18          = qword ptr -18h
.text:0000000000401842 var_4           = dword ptr -4
.text:0000000000401842
.text:0000000000401842 ; __unwind {
.text:0000000000401842                 push    rbp
.text:0000000000401843                 mov     rbp, rsp
.text:0000000000401846                 sub     rsp, 430h
.text:000000000040184D                 mov     [rbp+var_428], rdi
```

This is the resulting function:

```
int __fastcall secret_function(__int64 a1, char *a2)
{
  char *v2; // ST00_8
  char *old; // [rsp+0h] [rbp-430h]
  char *ptr; // [rsp+18h] [rbp-418h]
  __int64 v6; // [rsp+20h] [rbp-410h]
  char v7; // [rsp+28h] [rbp-408h]
  unsigned __int128 v8; // [rsp+410h] [rbp-20h]
  int v9; // [rsp+42Ch] [rbp-4h]

  old = a2;
  v6 = 47LL;
  memset(&v7, 0, 0x3E0uLL);
  v9 = 1;
  v8 = seed * (unsigned __int128)seed * seed * seed;
  while ( v8 != 0 )
  {
    sprintf((char *)&v6 + v9, "%02x", (unsigned __int8)v8, old);
    v8 >>= 8;
    v9 += 2;
  }
  *((_BYTE *)&v6 + v9) = 0;
  ptr = 0LL;
  exit_if_debugging();
  asprintf(&ptr, "%s%s", a1, &v6, old);
  return rename(v2, ptr);
}
```

And this is the function after some renaming:

```c
int __fastcall rename_temp_file(char *file_path, char *file_name)
{
  char *v2; // ST00_8
  char *temp_file_name; // [rsp+0h] [rbp-430h]
  char *ptr; // [rsp+18h] [rbp-418h]
  char *new_name; // [rsp+20h] [rbp-410h]
  char v7; // [rsp+28h] [rbp-408h]
  unsigned __int128 generated_number; // [rsp+410h] [rbp-20h]
  int i; // [rsp+42Ch] [rbp-4h]

  temp_file_name = file_name;
  new_name = (char *)'/';
  memset(&v7, 0, 0x3E0uLL);
  i = 1;
  generated_number = seed * (unsigned __int128)seed * seed * seed;
  while ( generated_number != 0 )
  {
    sprintf((char *)&new_name + i, "%02x", (unsigned __int8)generated_number, temp_file_name);
    generated_number >>= 8;
    i += 2;
  }
  *((_BYTE *)&new_name + i) = 0;
  ptr = 0LL;
  exit_if_debugging();
  asprintf(&ptr, "%s%s", file_path, &new_name, temp_file_name);
  return rename(v2, ptr);
}
```

I wasn't entirely sure of the result of this function, so I simulated it by writing a simple C program (**renaming.c**). The **new_name** string is the **generated_number** interpreted as a hexadecimal, written backwards one byte at a time (two hex digits).

```
seconds: 1682269872
seed: 3135821919
generated_number: 4194530945
generated_number as HEX: 0xfa037681
new_name: /817603fa
```

This process is reversible. From the name of the file we can get the **generated_number**. And from this we can get the **seed** by computing the fourth root of **generated_number.** Note: **seed** is unsigned, thus always positive.

Task 3 answer:

The name of the encrypted file is computed from the generated **seed**. Then, **seed** to the power of **4** is stored in another variable. This new value is interpreted in hexadecimal and is used for the new name. Lastly, the bytes are reversed.

To decrypt the file we should:

1. Compute the **seed** using the described process.
2. Use this **seed** to initialize the random number generator.

3. Go through the first portion of the file (until the **"fmi_re_course"** string) and apply the reverse of the encryption procedure. This means reading the bytes, subtracting a random value from them (we should be getting the same random values as the ones in the encryption process, since the **seed** is the same) and writing to them backwards. We have to read them in the right order, so we use the same random values as in the encryption.

   Note: According to some info I found online, we should also write this program in C, as the **rand** function might return different values when using other programming languages, even when using the same seed. Furthermore, even the compiler has to be the same in order to guarantee that the **rand** will return the same values.
4. Skip the **"fmi_re_course"** string and subtract the remaining pseudo random values from the rest of the file to restore the original file name.

Task 4:

- Create a program/script that decrypts any given encrypted file including the target file in the archive.

I created several programs to achieve this. These are **1_get_info.py**, **2_decrypt.cpp** and **3_reverse_file.py**.

The first one is a script that computes the **seed** and finds the position of the **"fmi_re_course"** string in the file. The second functionality worked for my test file, but not for the one in the assignment. I found this value by manually inspecting the binary file.

The second program is responsible for decrypting the file. I hardcoded the name of the input file and the information generated by the previous Python script. It also prints the original name of the encrypted file. The problem with this program is that it does not reverse the contents of the file.

Because I did not find a simple solution to reversing the file on the go inside the C++ program, I created another script for doing exactly this.

Running these three steps seemed to work on some files, but not on the one given in the assignment. I found the name of the original file, which is **target_file.encrypt_me_baby_one_more_time** and the file seemed to be a PNG image. But the resulting file was corrupted. I checked and even the header had one wrong byte. I tried to fix it by changing that particular byte, but it did not work. There must be more mistakes inside the file.