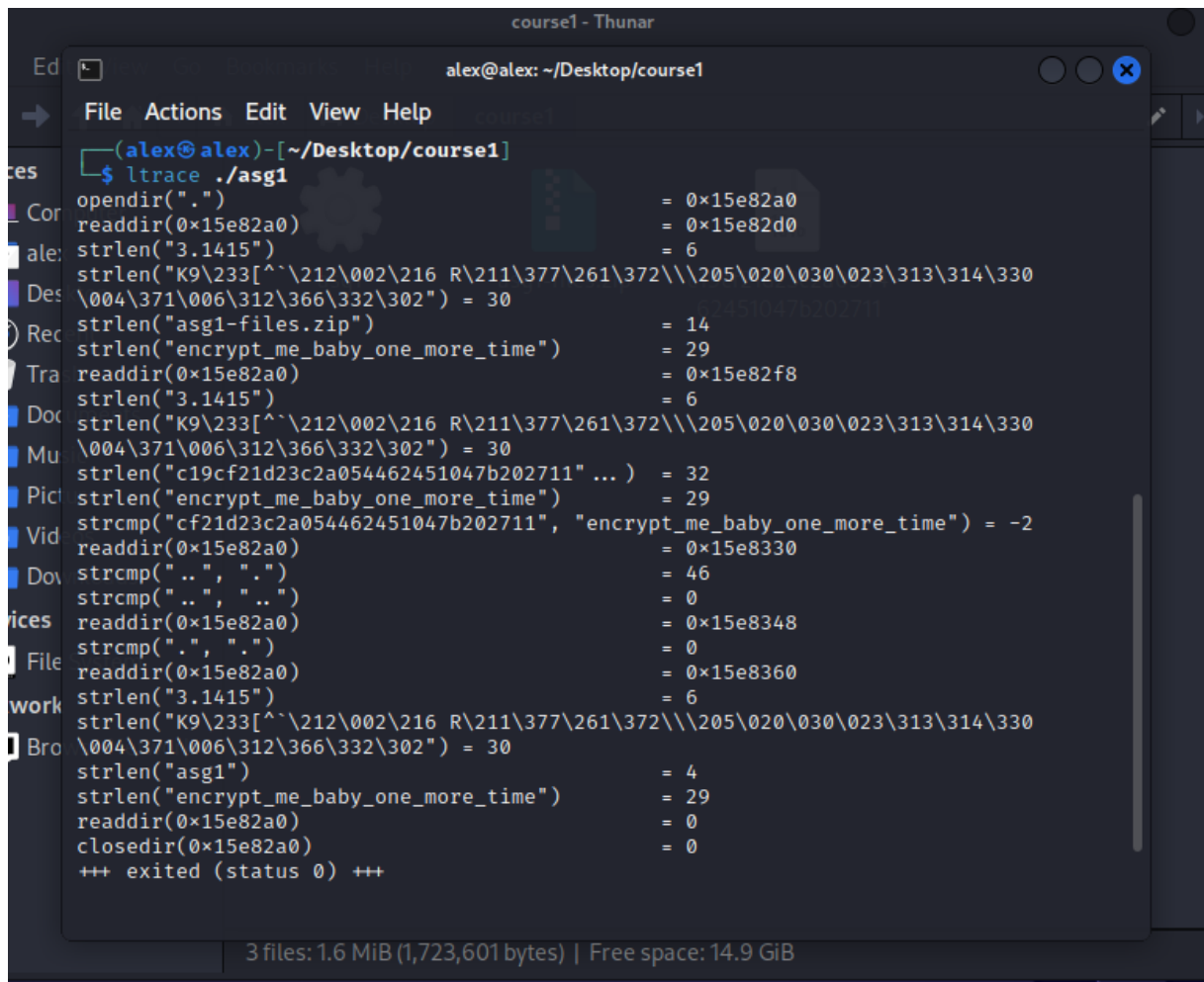


Project 1 - Maftei Ioan-Alexandru, grupa 510

Exercitiu 1

Am incercat sa ma folosesc de comanda strings sa vad daca primesc ceva folositor, dar nu am primit. Astfel, am decis sa vad ce functii se apeleaza la runtime si cu ce valori se apeleaza. Am facut asta prin intermediul comenzii *ltrace*.



```
alex@alex: ~/Desktop/course1
$ ltrace ./asg1
opendir(".") = 0x15e82a0
readdir(0x15e82a0) = 0x15e82d0
strlen("3.1415") = 6
strlen("K9\233[^^\212\002\216 R\211\377\261\372\\\205\020\030\023\313\314\330
\004\371\006\312\366\332\302") = 30
strlen("asg1-files.zip") = 14
strlen("encrypt_me_baby_one_more_time") = 29
readdir(0x15e82a0) = 0x15e82f8
strlen("3.1415") = 6
strlen("K9\233[^^\212\002\216 R\211\377\261\372\\\205\020\030\023\313\314\330
\004\371\006\312\366\332\302") = 30
strlen("c19cf21d23c2a054462451047b202711" ...) = 32
strlen("encrypt_me_baby_one_more_time") = 29
strcmp("cf21d23c2a054462451047b202711", "encrypt_me_baby_one_more_time") = -2
readdir(0x15e82a0) = 0x15e8330
strcmp("..", ".") = 46
strcmp("..", "..") = 0
readdir(0x15e82a0) = 0x15e8348
strcmp(".", ".") = 0
readdir(0x15e82a0) = 0x15e8360
strlen("3.1415") = 6
strlen("K9\233[^^\212\002\216 R\211\377\261\372\\\205\020\030\023\313\314\330
\004\371\006\312\366\332\302") = 30
strlen("asg1") = 4
strlen("encrypt_me_baby_one_more_time") = 29
readdir(0x15e82a0) = 0
closedir(0x15e82a0) = 0
+++ exited (status 0) +++
```

Din imaginea de mai sus putem observa cateva string-uri: "3.1415" (pe care l-am gasit si cu strings) si "encrypt_me_baby_one_more_time". Tot acolo observam ca se face un *strcmp* cu un fisier si string-ul cu "encrypt". Astfel, ca sa gasim unde se face aceasta comparatie, vom trece in IDA si incepem cu functia *main*.

Functia *main* este simpla deoarece se apeleaza doar o functie si se returneaza 0. Contin-

uam analiza cu functia gasita in main.

Dupa ce am analizat functia in care am ajuns, am redenumit cateva variabile si am redenumit functia in *EncryptRecursive*. Functia este recursiva deoarece in interiorul ei se apeleaza pe sine (linia 22).

```
1 int __fastcall EncryptRecursive(const char *currentDirectory)
2 {
3     DIR *v1; // rax
4     char *ptr; // [rsp+18h] [rbp-18h]
5     struct dirent *crtItem; // [rsp+20h] [rbp-10h]
6     DIR *dirp; // [rsp+28h] [rbp-8h]
7
8     v1 = opendir(currentDirectory);
9     dirp = v1;
10    if ( v1 )
11    {
12        while ( 1 )
13        {
14            crtItem = readdir(dirp);
15            if ( !crtItem )
16                break;
17            if ( crtItem->d_type == 4 )
18            {
19                if ( strcmp(crtItem->d_name, ".") && strcmp(crtItem->d_name, "..") )
20                {
21                    asprintf(&ptr, "%s/%s", currentDirectory, crtItem->d_name);
22                    EncryptRecursive(ptr);
23                    free(ptr);
24                }
25            }
26            else
27            {
28                FileEncryption((char *)currentDirectory, crtItem->d_name); // crtItem->d_name = file name
29            }
30        }
31        LODWORD(v1) = closedir(dirp);
32    }
33    return (signed int)v1;
34 }
```

De asemenea, functia de la linia 28 a fost analizta pe scurt si a fost redenumita in *FileEncryption*.

Astfel, observam ca functia *EncryptRecursive* face o parcurgere recursiva prin ierarhia de fisiere Linux, pornind din folderul radacina: daca readdir returneaza NULL, executia se va opri, iar daca este fisier (dar nu unul special) se va apela recursiv metoda de criptare, in cazul in care este un fisier se va apela functia de *FileEncryption*.

Trecem in functia *FileEncryption* in care vom face niste redenumiri si observam ca se foloseste un vector care trebuie retype-uit.

```

1 unsigned int __fastcall FileEncryption(char *directoryName, char *fileName)
2 {
3     unsigned int result; // eax
4     char v3; // [rsp+10h] [rbp-60h]
5     _BYTE v4[7]; // [rsp+11h] [rbp-5Fh]
6     char v5[32]; // [rsp+50h] [rbp-20h]
7
8     v5[0] = 75;
9     v5[1] = 57;
10    v5[2] = 155;
11    v5[3] = 91;
12    v5[4] = 94;
13    v5[5] = 96;
14    v5[6] = 138;
15    v5[7] = 2;
16    v5[8] = -114;
17    v5[9] = 32;
18    v5[10] = 82;
19    v5[11] = -119;
20    v5[12] = -1;
21    v5[13] = -79;
22    v5[14] = -6;
23    v5[15] = 92;
24    v5[16] = -123;
25    v5[17] = 16;
26    v5[18] = 24;
27    v5[19] = 19;
28    v5[20] = -53;
29    v5[21] = -52;
30    v5[22] = -40;
31    v5[23] = 4;
32    v5[24] = -7;
33    v5[25] = 6;
34    v5[26] = -54;
35    v5[27] = -10;
36    v5[28] = -38;
37    v5[29] = -62;
38    sub_4014CC((__int64)"3.1415", (__int64)v5, (__int64)&v3);
39    result = sub_40152C(fileName, v4);
40    if ( result )
41    {
42        sub_4019D2(directoryName, fileName);
43        result = sleep(1u);
44    }

```

In cadrul acestei functii observam ca are loc o criptare (la prima vedere) prin intermediul functiei *sub_4014CC* cu cheia *3.1415*. Momentan nu este de interes deci vom sari peste analiza acesteia in detaliu.

Dupa ce trecem de functia de criptare, observam ca se mai apeleaza o functie *sub_40152C* pe care o vom analiza in continuare.

```

1  BOOL __fastcall sub_40152C(const char *fileName, const char *a2)
2  {
3      int a2_length; // [rsp+18h] [rbp-8h]
4      int a1_length; // [rsp+1Ch] [rbp-4h]
5
6      a1_length = strlen(fileName);
7      a2_length = strlen(a2);
8      return a1_length >= a2_length && !strcmp(&fileName[a1_length - a2_length], a2);
9  }

```

Observam ca se returneaza true daca lungimea primului sir este mai mare sau egala decat lungimea celui de al doilea si daca al doilea sir este un sufix al primului. Astfel, putem decide ca apelul functiei *sub_40152C* se face cu parametrii (fileName, v4), unde v4 este un sufix pentru fileName.

Pentru a verifica acest lucru, vom trece la analiza dinamica si vom pune in gdb un breakpoint la adresa apelului functiei *sub_40152C*, adica la adresa 0x401C56. Facem acest lucru pentru a verifica argumentele functiei (trimisi prin RDI si RSI).

```

RCX: 0xc2
RDX: 0x4052e3 ("asg1-files.zip")
RSI: 0x7fffffffdd61 ("encrypt_me_baby_one_more_time")
RDI: 0x4052e3 ("asg1-files.zip")
RBP: 0x7fffffffddc0 → 0x7fffffffde00 → 0x7fffffffde10 → 0x1
RSP: 0x7fffffffdd50 → 0x4052e3 ("asg1-files.zip")
RIP: 0x401c56 (call 0x40152c)
R8 : 0x426000 ('')
R9 : 0x21001
R10: 0x7ffff7ddb360 → 0x10001a000070bc
R11: 0x7ffff7f21d00 (<__strlen_avx2>: mov eax,edi)
R12: 0x0
R13: 0x7fffffffdf38 → 0x7fffffffef297 ("COLORFGBG=15;0")
R14: 0x0
R15: 0x7ffff7ffd020 → 0x7ffff7ffe2e0 → 0x0
EFLAGS: 0x202 (carry parity adjust zero sign trap INTERRUPT direction overflow)

[-----code-----]
--]
0x401c4c: mov rdx,QWORD PTR [rbp-0x70]
0x401c50: mov rsi,rdx
0x401c53: mov rdi,rdx
⇒ 0x401c56: call 0x40152c
0x401c5b: test eax,eax
0x401c5d: je 0x401c7c
0x401c5f: mov rdx,QWORD PTR [rbp-0x70]
0x401c63: mov rax,QWORD PTR [rbp-0x68]
Guessed arguments:
arg[0]: 0x4052e3 ("asg1-files.zip")
arg[1]: 0x7fffffffdd61 ("encrypt_me_baby_one_more_time")
arg[2]: 0x4052e3 ("asg1-files.zip")

[-----stack-----]
--]
0000| 0x7fffffffdd50 → 0x4052e3 ("asg1-files.zip")
0008| 0x7fffffffdd58 → 0x40202a → 0x1b0100002e2e002e
0016| 0x7fffffffdd60 ("encrypt_me_baby_one_more_time")
0024| 0x7fffffffdd68 ("_me_baby_one_more_time")
0032| 0x7fffffffdd70 ("_one_more_time")
0040| 0x7fffffffdd78 → 0x656d69745f65 ('e_time')
0048| 0x7fffffffdd80 → 0x93ebb05
0056| 0x7fffffffdd88 → 0x7ffff7e9bef5 (<__GI__readdir64+117>: test
    rax,rax)

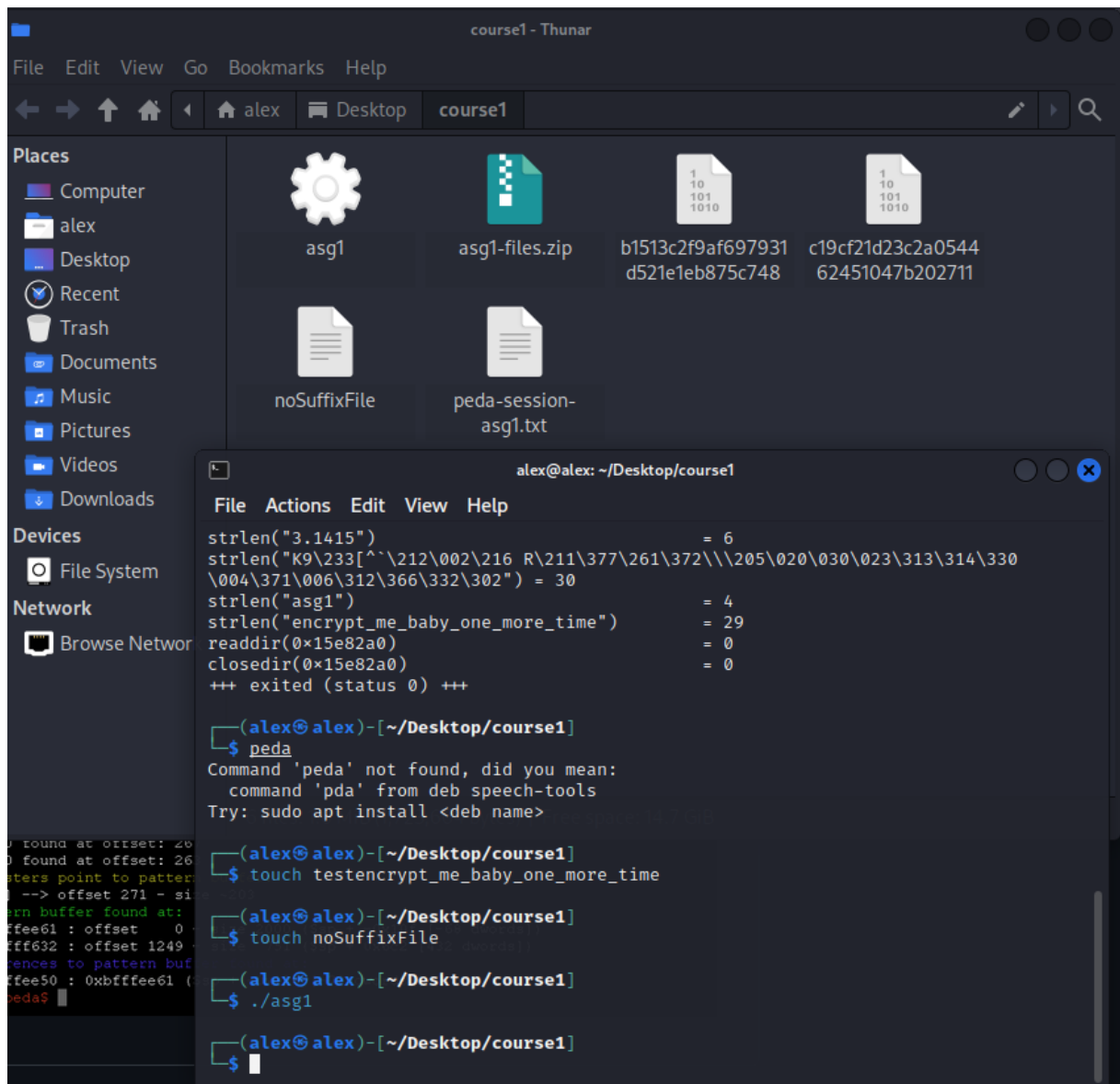
[-----]
--]
Legend: code, data, rodata, value

Breakpoint 1, 0x0000000000401c56 in ?? ()
gdb-peda$

```

La fel cum am observat si in ltrace, putem vedea in RSI valoarea "encrypt_me_baby_one_more_time". Deci in acest moment, observam ca se verifica numele fisierului sa fie sufix al sirului enuntat mai sus.

Asfel vom rula executabilul cu un fisier denumit "testencrypt_me_baby_one_more_time" si un fisier care sa nu contina sirul enuntat anterior pentru a vedea ce se intampla.



Dupa cum putem obseva, fisierul care a avut sufixul enuntat a fost criptat dupa ce am rulat executabilul, iar fisierul care nu avea sufixul, nu a fost criptat.

Exercitiu 2

In continuare, vom analiza modul in care se face criptarea in cazul in care se returneaza TRUE de la verificarile din task-ul precedent. Pentru aceasta, vom analiza functia *sub_4019D2* pe care o vom denumi in continuare *encrypt*. In interiorul ei vom face cateva redenumiri conforme cu informatiile adunate pana acum.

```
1 void __fastcall encrypt(const char *directoryName, const char *fileName)
2 {
3     const char *_directoryName; // ST08_8
4     __int64 v3; // ST00_8
5     __int64 v4; // rdi
6     __int64 v5; // rdi
7     __off_t v6; // rsi
8     FILE *v7; // rdi
9     FILE *v8; // rdi
10    __int128 filename; // [rsp+10h] [rbp-C0h]
11    struct stat stat_buf; // [rsp+20h] [rbp-B0h]
12    FILE *v11; // [rsp+B8h] [rbp-18h]
13    FILE *stream; // [rsp+C0h] [rbp-10h]
14    __off_t v13; // [rsp+C8h] [rbp-8h]
15
16    _directoryName = directoryName;
17    sub_401593(directoryName, fileName);
18    seed = (unsigned __int64)time(0LL) ^ 0xDEADBEEF;
19    srand(seed);
20    filename = 0uLL;
21    asprintf((char **)&filename + 1, "%s/%s", directoryName, fileName, fileName);
22    sub_401593((char *)&filename + 8, "%s/%s");
23    asprintf((char **)&filename, "%s/%s_temp", directoryName, v3);
24    sub_401593(&filename, "%s/%s_temp");
25    sub_401E00*((char **)&filename + 1), &stat_buf);
26    v13 = stat_buf.st_size;
27    v4 = *((_QWORD *)&filename + 1);
28    stream = fopen(*((const char **)&filename + 1), "r");
29    sub_401593(v4, "r");
30    v5 = filename;
31    v11 = fopen((const char *)filename, "w");
32    sub_401593(v5, "w");
33    v6 = v13;
34    v7 = stream;
35    sub_40169A(stream, v13, v3, v11);
36    sub_401593(v7, v6);
37    fclose(stream);
38    v8 = v11;
39    fclose(v11);
40    sub_4015F7(v8);
41    (*(void (__fastcall **)(const char *, _QWORD))word_401842)(_directoryName, filename);
42    sub_4015F7(_directoryName);
43    //
44    //
```

Observam ca in functia *encrypt* se apeleaza functia *sub_401593* de mai multe ori, astfel ne uitam in ea sa vedem ce face si aflam ca este un mecanism antidebug. Din aceasta

cauza vom redenumi functia in *AntiDebug*.

Pentru a trece cu usurinta peste aceasta metoda, putem seta variabila *dword_4040E4* sa fie egala cu 1.

Revenim la functia *encrypt* pentru a o analiza in continuare. Astfel, observam ca se creaza un fisier temporar cu path-ul *directoryName/fileName_temp* si este deschis in modul de scriere. Fisierul care nu este temporar (proper file) este deschis tot in modul de citire.

Cu aceste informatii, vom face rename si retype:

```
1 void __fastcall encrypt(const char *directoryName, const char *fileName)
2 {
3     char *anotherFileName; // ST00_8
4     __int128 temporaryFile; // [rsp+10h] [rbp-C0h]
5     struct stat stat_buf; // [rsp+20h] [rbp-B0h]
6     FILE *temporaryFileDescriptor; // [rsp+B8h] [rbp-18h]
7     FILE *properFileDescriptor; // [rsp+C0h] [rbp-10h]
8     __int64 properFileLength; // [rsp+C8h] [rbp-8h]
9
10    AntiDebug();
11    seed = (unsigned __int64)time(0LL) ^ 0xDEADBEEF;
12    srand(seed);
13    temporaryFile = 0uLL;
14    asprintf((char **)&temporaryFile + 1, "%s/%s", directoryName, fileName, fileName);
15    AntiDebug();
16    asprintf((char **)&temporaryFile, "%s/%s_temp", directoryName, anotherFileName);
17    AntiDebug();
18    sub_401E00(*((char **)&temporaryFile + 1), &stat_buf);
19    properFileLength = stat_buf.st_size;
20    properFileDescriptor = fopen(*((const char **)&temporaryFile + 1), "r");
21    AntiDebug();
22    temporaryFileDescriptor = fopen((const char *)temporaryFile, "w");
23    AntiDebug();
24    sub_40169A(properFileDescriptor, properFileLength, anotherFileName, temporaryFileDescriptor);
25    AntiDebug();
26    fclose(properFileDescriptor);
27    fclose(temporaryFileDescriptor);
28    sub_4015F7();
29    (*(void (__fastcall **)(const char *, _QWORD))word_401842)(directoryName, temporaryFile);
30    sub_4015F7();
31    //
32    //
33    unlink(*((const char **)&temporaryFile + 1));
34    unlink((const char *)temporaryFile);
35    free(*((void **)&temporaryFile + 1));
36    free((void *)temporaryFile);
37 }
```

In continuare, ne vom axa pe functiile ramase nedenumite. Cea mai de interes este functia *sub_40169A* deoarece ea se foloseste atat de fisierul normal cat si de cel creat temporar.

Dupa ce facem rename si retype si mai comentam putin codul ca sa fie usor de inteles ulterior ajungem la urmatoarea forma:

```

IDA View-A  Pseudocode-A  Stack of sub_40169A  Stack of encrypt  Hex View-1  Structures  Enums
1 size_t __fastcall createTempFile(FILE *properFileDescriptor, signed __int64 properFileLength, char *fileName, FILE *temporaryFileDescriptor)
2 {
3     size_t indexCopy2; // rbx
4     size_t indexCopy; // rbx
5     size_t length; // rax
6     FILE *temporaryFileDescriptorCopy; // [rsp+0h] [rbp-50h]
7     const char *fileNameCopy; // [rsp+8h] [rbp-48h]
8     char fileNameCharacter; // [rsp+2Ch] [rbp-24h]
9     char properFileContent_ptr; // [rsp+2Dh] [rbp-23h]
10    char arr[8]; // [rsp+2Eh] [rbp-22h]
11    int i; // [rsp+3Ch] [rbp-14h]
12
13    fileNameCopy = fileName;
14    temporaryFileDescriptorCopy = temporaryFileDescriptor;
15    fseek(properFileDescriptor, -1LL, 2); // pointer close to EOF
16    for ( i = 0; i < properFileLength; ++i ) // write first position in file
17    {
18        fread(&properFileContent_ptr, 1uLL, 1uLL, properFileDescriptor);
19        AntiDebug();
20        fseek(properFileDescriptor, -2LL, 1); // move pointer one position back; right to left reading
21        AntiDebug();
22        properFileContent_ptr += rand();
23        AntiDebug();
24        fwrite(&properFileContent_ptr, 1uLL, 1uLL, temporaryFileDescriptorCopy);
25        AntiDebug();
26    }
27    //
28    strcpy(arr, "fmi_re_course");
29    for ( i = 0; ; ++i ) // write string to second position in file
30    {
31        indexCopy2 = i;
32        if ( indexCopy2 >= strlen(arr) )
33            break;
34        fwrite(&arr[i], 1uLL, 1uLL, temporaryFileDescriptorCopy);
35    }
36    //
37    for ( i = 0; ; ++i ) // write to last position in file
38    {
39        indexCopy = i;
40        length = strlen(fileNameCopy);
41        if ( indexCopy >= length )
42            break;
43        fileNameCharacter = fileNameCopy[i];
44        fileNameCharacter += rand();
45        fwrite(&fileNameCharacter, 1uLL, 1uLL, temporaryFileDescriptorCopy);
46    }
47    return length;
48 }

```

Vom grupa cele 3 for-uri si le vom discuta pe rand ce fac si ce scriu in fisier:

1. in prima parte se face un fseek in fisier, doar ca se pozitioneaza capul de citire la un byte inainte de finalul fisierului. Apo se itereaza pana la numarul de bytes continuti (fiecare byte se citeste pe rand intr-o variabila ptr), apoi se muta capul de citire la -2 relativ la SEEK_CUR(1) pentru a trece peste byte-ul citit curent (acesta urmeaza sa fie folosit la urmatoarea iteratie). Apoi la byte-ul din ptr se adauga un rand(). Seed-ul pentru rand este folosit in functia anterioara (cea cu xor 0xDEADBEEF). Apoi ptr este scris in fisierul temporar creat anterior;
2. in a doua parte (al doilea for), se completeaza intr-o variabila deunmita buffer, sirul de caractere "fmi_re_course" si este scris in fisierul temporar byte cu byte;
3. in cea de a treia parte se aplica aceiasi alterare de bytes cum a fost la prima parte, doar ca acum se foloseste numele fisierului propriu-zis (nu cel temporar) si este scris apoi in fisierul temporar.

Exercitiu 3

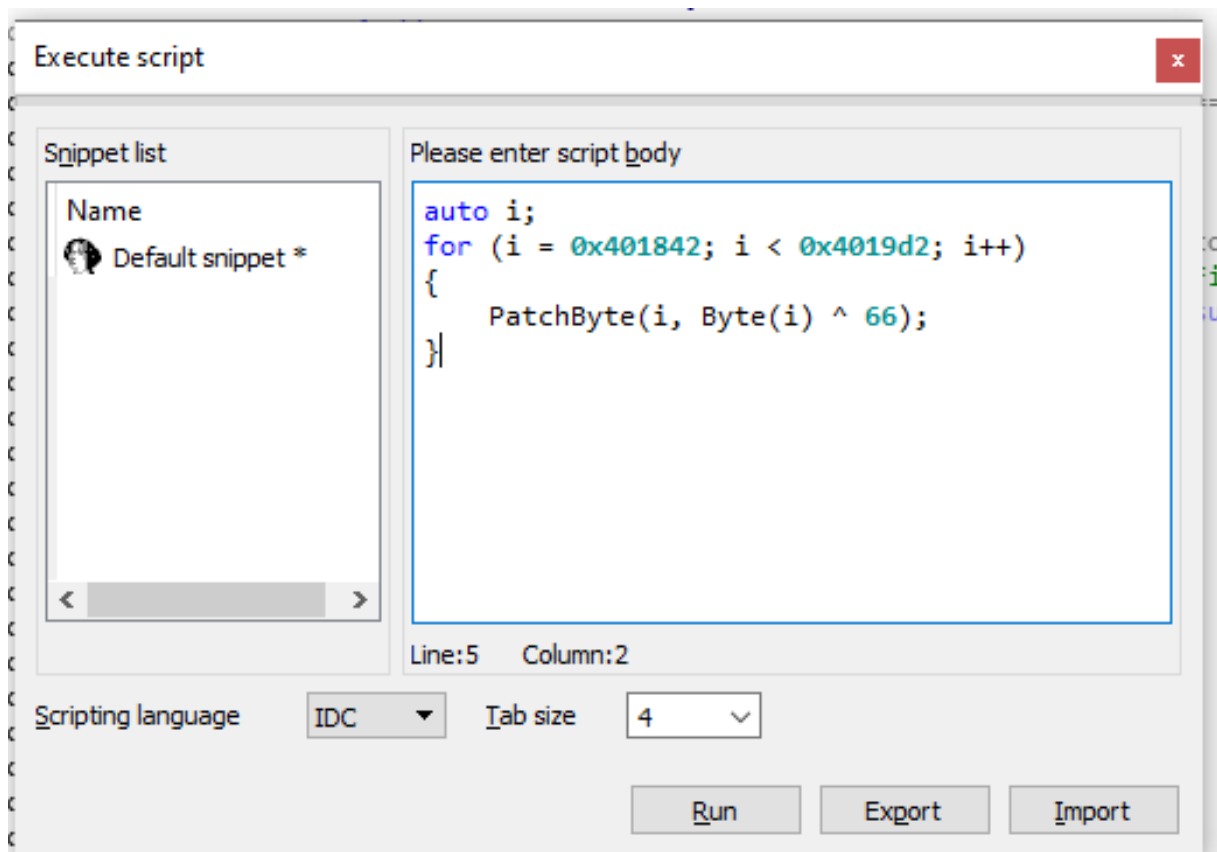
Pe parcursul analizei am dat peste un apel de functie care se ocupa de renaming, doar ca el este o functie criptata:

```
28 | sub_4015F7();  
29 | (*(void (__fastcall **)(const char *, _QWORD))word_401842)(directoryName, temporaryFile);  
30 | sub_4015F7();
```

Dam dublu click pe word_401842 ca sa gasim unde se afla in memorie si apoi cautam referintele unde este folosit si aflam ca este folosit in 2 locuri: cel de unde am plecat (cel care apeleaza functia criptata) si locul in care se decripteaza functia noastra.

Continuam prin analizarea celui de al doilea loc unde este folosit si aflam ca acel cod din memorie este de fapt xorat byte cu byte cu valoarea **0x42** in hexadecimal sau 66 in decimal. Xorarea este o functie care este reversibila foarte usor prin rexorarea cu aceiasi valoare. Stim adresa de inceput (0x401842) si de final (0x4019d2) pentru codul criptat, deci putem sa decriptam cu urmatorul program scris in IDC:

```
1 auto i;  
2 for (i = 0x401842; i < 0x4019d2; i++)  
3 {  
4     PatchByte(i, Byte(i) ^ 66);  
5 }
```



Astfel, dupa ce am decriptat si dupa ce am facut renaming, functia arata astfel:

```

1 int __fastcall renameFile(__int64 directoryName, const char *temporaryFileDescriptor)
2 {
3     char *v2; // ST00_8
4     char *old; // [rsp+0h] [rbp-430h]
5     char *ptr; // [rsp+18h] [rbp-418h]
6     __int64 result; // [rsp+20h] [rbp-410h]
7     char v7; // [rsp+28h] [rbp-408h]
8     unsigned __int128 seed2; // [rsp+410h] [rbp-20h]
9     int i; // [rsp+42Ch] [rbp-4h]
10
11     old = (char *)temporaryFileDescriptor;
12     result = 47LL;
13     memset(&v7, 0, 0x3E0uLL);
14     i = 1;
15     seed2 = seed * (unsigned __int128)seed * seed * seed;
16     while ( seed2 != 0 )
17     {
18         sprintf((char *)&result + i, "%02x", (unsigned __int8)seed2, old);
19         seed2 >>= 8;
20         i += 2;
21     }
22     *((_BYTE *)&result + i) = 0;
23     ptr = 0LL;
24     sub_401593();
25     asprintf(&ptr, "%s%s", directoryName, &result, old);
26     return rename(v2, ptr);
27 }

```

Observam ca seed-ul se ridica la puterea a 4-a si este memorat in variabila "seed2" apoi se parcurc bitii din seed2 pana cand se ajunge la 0, se memoreaza in result[i] cate un

byte, iar apoi se elimina cei mai nesemnificativi 8 biti (mai precis, un singur byte din seed2). Indexul i este marit cu 2 de fiecare data deoarece un byte este format din doua cifre hexadecimale.

Asta inseamna ca numele fisierului ne va oferi seed-ul pentru decriptare. Deci daca luam procedeul invers vom ajunge sa aflam care este seed-ul folosi:

1. citim byte cu byte numele fisierului, doar ca de la finalul fisierului. Citirea se va face cu cate 2 caractere;
2. pentru fiecare 2 caractere, se inverseaza intre ele (ca la little vs big endian);
3. convertim sirul final in baza 10;
4. facem radical de ordin 4 din rezultatul din baza 10 (pentru ca seed-ul este ridicat la a 4) pentru a afla seed-ul generat initial.

Acum ca avem seed-ul calculat, putem sa facem diferenta `byte - rand()` (ca sa fie procedeul invers adunarii despre care am discutat anterior). Trebuie sa tinem cont de faptul ca in fisierul generat, noi de fapt avem 3 chestii: mesajul, textul hardcodat si numele fisierului.

Exercitiu 4

Pentru a putea decripta, vom folosi pasii de la task-ul 3. Adica:

1. aflam seed-ul pe baza numelui fisierului prin transformarea in hexa, apoi byte cu byte de la finalul sirului cate 2 caractere, facem ca la big vs little endian cu fiecare grup de caractere, apoi il transformam in intreg si facem radical de ordin 4;
2. spargem continutul fisierului creat in 3 (fisierul e de forma: `continut_hardcoded-String_numeleFisieruluiOriginal`. Textul hardcodat este "fmi_re_course");
3. luam prima parte si ultima parte a continutului spart si le parcurgem byte cu byte si facem transformarea inversa criptarii, adica `byte - rand()`;
4. facem un fisier nou cu numele original si scriem in el continutul de dupa criptare.

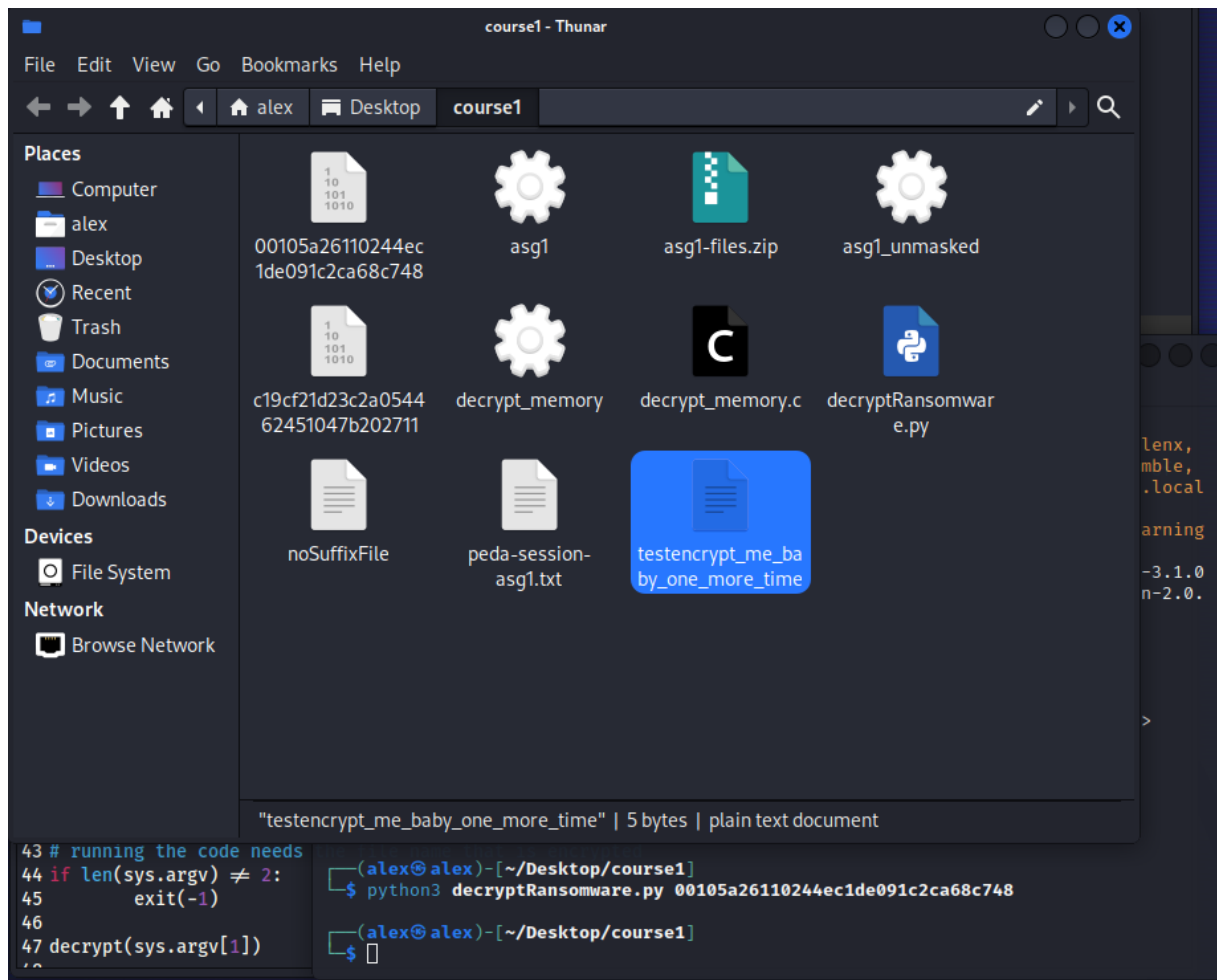
Codul python rezultat este:

```

6
7 libc = CDLL("libc.so.6")
8
9 def decrypt(filename):
10     # Calculate initial seed from the seed that is powered by 4
11     little_endian_conversion = codecs.encode(codecs.decode(filename, 'hex')[::-1], 'hex').decode()
12     seedToPowerOf4 = int(little_endian_conversion, 16)
13     initialSeed = math.floor(math.sqrt(math.sqrt(seedToPowerOf4)))
14     libc.srand(initialSeed)
15
16     encryptedFileDescriptor = open(filename, 'rb')
17     encryptedFileContent = encryptedFileDescriptor.read()
18     encryptedFileDescriptor.close()
19
20     # Split the crypted file information into 3 parts and use the first and the last one
21     splitIndex = encryptedFileContent.index(b"fmi_re_course")
22     message = encryptedFileContent[0:splitIndex]
23     originalFileName = encryptedFileContent[splitIndex + 13:]
24
25     decryptedMessage = []
26     decryptedOriginalFileName = ""
27     for b in bytes(message):
28         # Reverse encryption the initial message by using subtraction and then concatenate to the list of
29         # decrypted bytes
30         decryptedMessage += [(b - libc.rand()) % 0x100]
31         decryptedMessage = decryptedMessage[::-1]
32
33     # Reverse encryption for the original file name
34     for b in bytes(originalFileName):
35         decryptedOriginalFileName += chr((b - libc.rand()) % 0x100)
36
37     # write byte with byte the decrypted message
38     f = open(decryptedOriginalFileName, "wb")
39     for b in decryptedMessage:
40         f.write(bytes([b]))
41     f.close()
42
43 # running the code needs the file name that is encrypted
44 if len(sys.argv) != 2:
45     exit(-1)
46
47 decrypt(sys.argv[1])

```

Vom folosi codul pe un fisier criptat denumit conform taskului 3 si continutul va fi un text simplu: "test". Vom rula executabilul ca sa ne cripteze fisierul (fisierul criptat se cheama: 00105a26110244ec1de091c2ca68c748), iar apoi vom rula codul nostru python sa vedem daca functioneaza si observam ca fisierul decriptat este exact asa cum a fost cel initial, inainte de criptare.



Asa ca acum suntem convinsi ca merge cum trebuie decriptarea si putem sa o rulam pe fisierul primit ca sa vedem ce continea fisierul initial.

Rezultatul este:

