

Simple Encryptor

Capture the Flag Challenge.

Link: Challenge can be found [here](#).

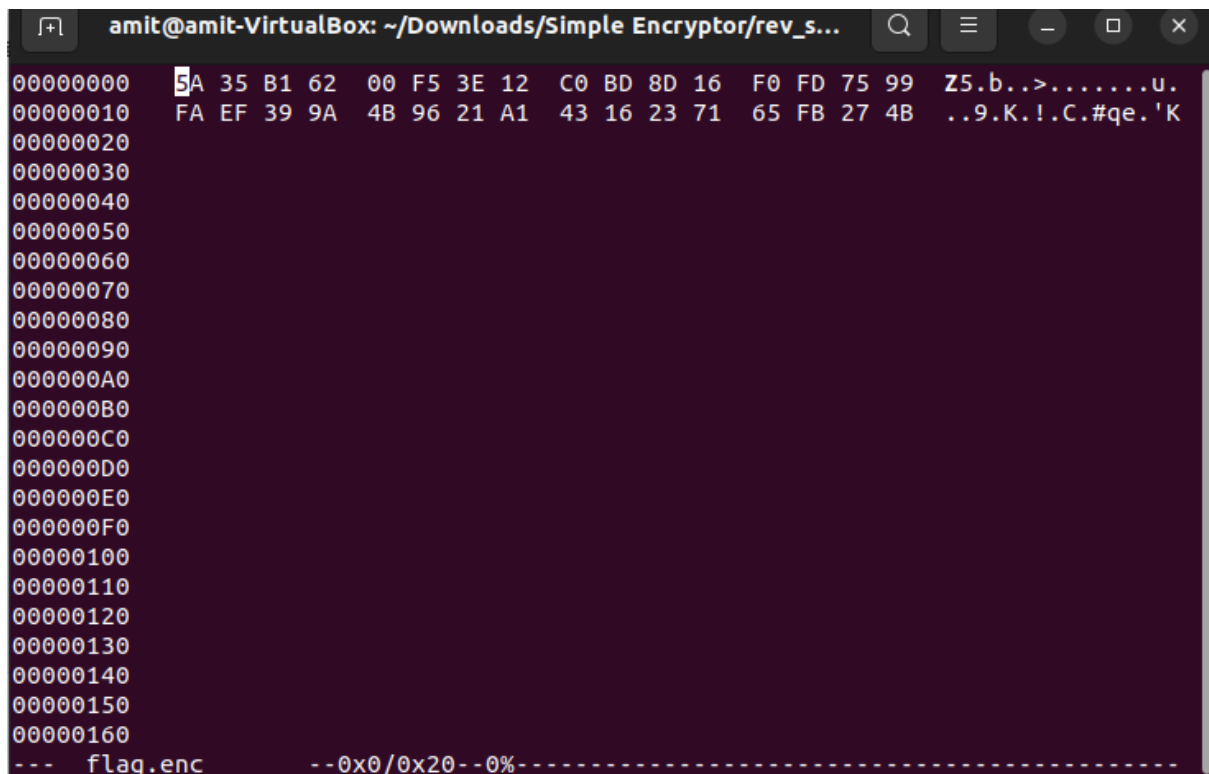
Overview: The task of this CTF is to decrypt an encrypted password to extract the flag.

In my possession is an encrypted binary file – ‘Flag.enc’, and the encryption algorithm – ‘encrypt’.

I had to do Inspection on both the file, and the algorithm in order to find a method to retrieve the original string.

Method: in order to decrypt the binary code – I needed to understand both it, and the encryption algorithm.

I will start investigating with the encrypted file:



```
amit@amit-VirtualBox: ~/Downloads/Simple Encryptor/rev_s...
00000000  5A 35 B1 62 00 F5 3E 12 C0 BD 8D 16 F0 FD 75 99 Z5.b..>.....u.
00000010  FA EF 39 9A 4B 96 21 A1 43 16 23 71 65 FB 27 4B ..9.K.!..C.#qe.'K
00000020
00000030
00000040
00000050
00000060
00000070
00000080
00000090
000000A0
000000B0
000000C0
000000D0
000000E0
000000F0
00000100
00000110
00000120
00000130
00000140
00000150
00000160
--- flag.enc ---0x0/0x20---0%-----
```

An hexedit view of the ‘flag.enc’ shows that we are dealing with 32 Bytes file.

And its content is encrypted, and unreadable with Ascii encoding.

So in order to continue the investigation – I turned toward the ‘encrypt’ program.

The first action that I tried is to run it directly with ‘./encrypt’.

Any attempt to run it directly, resulted in Segmentation fault:

```
amit@amit-VirtualBox: ~/Downloads/Simple Encryptor/rev_s...  
amit@amit-VirtualBox:~/Downloads/Simple Encryptor/rev_simpleencryptor$ ./encrypt  
Segmentation fault (core dumped)  
amit@amit-VirtualBox:~/Downloads/Simple Encryptor/rev_simpleencryptor$
```

So I had to look further how to make the program works.

First – I’ve checked its type with ‘objdump -s encrypt’, and its ELF x86-64 binary.

```
amit@amit-VirtualBox:~/Downloads/Simple Encryptor/rev_simpleencryptor$ objdump -s encrypt  
encrypt:      file format elf64-x86-64
```

Then I checked for files names in ‘keywords’ of the file.:

```
amit@amit-VirtualBox: ~/Downloads/Simple Encryptor/rev_s...  
13c0 488b45d0 483b45e0 7c86488d 353b0c00 H.E.H;E.|.H.5;..  
13d0 00488d3d 370c0000 e893fdff ff488945 .H.=7.....H.E  
13e0 f0488b55 f0488d45 c84889d1 ba040000 .H.U.H.E.H.....  
13f0 00be0100 00004889 c7e882fd ffff488b .....H.....H.  
1400 55e0488b 4df0488b 45e8be01 00000048 U.H.M.H.E.....H  
1410 89c7e869 fdffff48 8b45f048 89c7e8dd ...i...H.E.H....  
1420 fcffffb8 00000000 488b7df8 6448333c .....H.}.dH3<  
1430 25280000 007405e8 d4fcffff c9c36690 %(...t.....f.  
1440 f30f1efa 41574c8d 3d1b2900 00415649 ...AWL.=.)..AVI  
1450 89d64155 4989f541 544189fc 55488d2d ..AUI..ATA..UH.-  
1460 0c290000 534c29fd 4883ec08 e88ffbff ..)..SL).H.....  
1470 ff48c1fd 03741f31 db0f1f80 00000000 .H...t.1.....  
1480 4c89f24c 89ee4489 e741ff14 df4883c3 L..L..D..A...H..  
1490 014839dd 75ea4883 c4085b5d 415c415d .H9.u.H...[A\A]  
14a0 415e415f c366662e 0f1f8400 00000000 A^A_.ff.....  
14b0 f30f1efa c3  
Contents of section .fini:  
14b8 f30f1efa 4883ec08 4883c408 c3 ...H...H....  
Contents of section .rodata:  
2000 01000200 72620066 6c616700 77620066 ....rb.flag.wb.f  
2010 6c61672e 656e6300 lag.enc.  
Contents of section .eh_frame_hdr:  
2018 011b033b 44000000 07000000 08f0ffff ...;D.....  
2028 78000000 c8f0ffff a0000000 d8f0ffff x.....
```

And I observed that its input file is ‘flag’, and out file is ‘flag.enc’.

So I created a 'flag' file with some custom text in it. (lets say my name – 'amit'):

```
amit@amit-VirtualBox: ~/Downloads/Simple Encryptor/rev_simpleencr...  
amit@amit-VirtualBox:~/Downloads/Simple Encryptor/rev_simpleencryptor$ touch flag  
amit@amit-VirtualBox:~/Downloads/Simple Encryptor/rev_simpleencryptor$ echo amit > flag  
amit@amit-VirtualBox:~/Downloads/Simple Encryptor/rev_simpleencryptor$
```

I examined it again with hexedit:

```
amit@amit-VirtualBox: ~/Downloads/Simple Encryp...  
00000000  61 6D 69 74  0A               amit.  
0000000C  
00000018  
00000024  
00000030  
--- flag          --0x0/0x5--0%-----
```

The 'flag' file now contains the string: "amit\n".

Before running, just as a precaution – I Kept the original 'flag.enc' as backup – as I expected running the 'encrypt' will modify the 'flag.enc' file.

And I ran 'encrypt' again:

```
amit@amit-VirtualBox: ~/Downloads/Simple Encryptor/rev_simpleencryptor  
amit@amit-VirtualBox:~/Downloads/Simple Encryptor/rev_simpleencryptor$ ./encrypt  
amit@amit-VirtualBox:~/Downloads/Simple Encryptor/rev_simpleencryptor$ cat flag.enc  
-K.ds...&amit@amit-VirtualBox:~/Downloads/Simple Encryptor/rev_simpleencryptor$
```

The current attempt to run 'encrypt' succussed. And 'flag.enc' now contains encrypted binary of 'amit\n' string.

Hexedit reveals that the encrypted binary consisted of 9 bytes:

```
amit@amit-VirtualBox: ~/Downloads/Simple Encryptor/rev_simpleencryptor  
00000000  2D 4B FB 64  73 B6 10 9C  26               -K.ds...&  
00000010  
00000020  
00000030  
00000040  
00000050  
--- flag.enc      --0x0/0x9--0%-----
```

Which are 5 bytes of the original string. And 4 extra bytes whose purpose is currently unknown.

Further runs of 'encrypt' on different strings reveals consistency with the output size – the input string size + 4 extra bytes.

That is a start. Now it is time to investigate the 'encrypt' further with inspection on its assembly, and GDB debugging.

With running 'objdump -d encrypt', and observing on the assembly of 'main' I can divide the code into these sections:

- a. Setting the variables: Binary reading from a file. Storing its content in a char array variable and its size in an int variable, and setting timestamp with time(), and using its seed for srand().

```
0000000000001289 <main>:
1289: f3 0f 1e fa      endbr64
128d: 55              push %rbp
128e: 48 89 e5        mov %rsp,%rbp
1291: 48 83 ec 40     sub $0x40,%rsp
1295: 64 48 8b 04 25 28 00 mov %fs:0x28,%rax
129c: 00 00
129e: 48 89 45 f8     mov %rax,-0x8(%rbp)
12a2: 31 c0          xor %eax,%eax
12a4: 48 8d 35 59 0d 00 00 lea 0xd59(%rip),%rsi # 2004 <_IO_stdin_used+0x4>
12ab: 48 8d 3d 55 0d 00 00 lea 0xd55(%rip),%rdi # 2007 <_IO_stdin_used+0x7>
12b2: e8 b9 fe ff ff  call 1170 <fopen@plt>
12b7: 48 89 45 d8     mov %rax,-0x28(%rbp)
12bb: 48 8b 45 d8     mov -0x28(%rbp),%rax
12bf: ba 02 00 00 00  mov $0x2,%edx
12c4: be 00 00 00 00  mov $0x0,%esi
12c9: 48 89 c7       mov %rax,%rdi
12cc: e8 8f fe ff ff  call 1160 <fseek@plt>
12d1: 48 8b 45 d8     mov -0x28(%rbp),%rax
12d5: 48 89 c7       mov %rax,%rdi
12d8: e8 53 fe ff ff  call 1130 <ftell@plt>
12dd: 48 89 45 e0     mov %rax,-0x20(%rbp)
12e1: 48 8b 45 d8     mov -0x28(%rbp),%rax
12e5: ba 00 00 00 00  mov $0x0,%edx
12ea: be 00 00 00 00  mov $0x0,%esi
12ef: 48 89 c7       mov %rax,%rdi
12f2: e8 69 fe ff ff  call 1160 <fseek@plt>
12f7: 48 8b 45 e0     mov -0x20(%rbp),%rax
12fb: 48 89 c7       mov %rax,%rdi
12fe: e8 4d fe ff ff  call 1150 <malloc@plt>
1303: 48 89 45 e8     mov %rax,-0x18(%rbp)
1307: 48 8b 75 e0     mov -0x20(%rbp),%rsi
130b: 48 8b 55 d8     mov -0x28(%rbp),%rdx
130f: 48 8b 45 e8     mov -0x18(%rbp),%rax
1313: 48 89 d1       mov %rdx,%rcx
1316: ba 01 00 00 00  mov $0x1,%edx
131b: 48 89 c7       mov %rax,%rdi
131e: e8 cd fd ff ff  call 10f0 <fread@plt>
1323: 48 8b 45 d8     mov -0x28(%rbp),%rax
1327: 48 89 c7       mov %rax,%rdi
132a: e8 d1 fd ff ff  call 1100 <fclose@plt>
132f: bf 00 00 00 00  mov $0x0,%edi
1334: e8 07 fe ff ff  call 1140 <time@plt>
1339: 89 45 c8       mov %eax,-0x38(%rbp)
133c: 8b 45 c8       mov -0x38(%rbp),%eax
133f: 89 c7         mov %eax,%edi
1341: e8 da fd ff ff  call 1120 <srand@plt>
1346: 48 c7 45 d0 00 00 00 movq $0x0,-0x30(%rbp)
134d: 00
```

- b. The encryption process itself – taking each byte in the string, and encrypting it with xor with random byte. and the result of which, perform left rotation a random amount of times between 0 to 7.

The random values that were used in the encryption process – they were generated with the seed that was created on the first stage.

```

134e: eb 70          jmp     13c0 <main+0x137>
1350: e8 3b fe ff ff call    1190 <rand@plt>
1355: 0f b6 c8       movzbl %al,%ecx
1358: 48 8b 55 d0     mov     -0x30(%rbp),%rdx
135c: 48 8b 45 e8     mov     -0x18(%rbp),%rax
1360: 48 01 d0       add     %rdx,%rax
1363: 0f b6 00       movzbl (%rax),%eax
1366: 89 c2          mov     %eax,%edx
1368: 89 c8          mov     %ecx,%eax
136a: 89 d1          mov     %edx,%ecx
136c: 31 c1          xor     %eax,%ecx
136e: 48 8b 55 d0     mov     -0x30(%rbp),%rdx
1372: 48 8b 45 e8     mov     -0x18(%rbp),%rax
1376: 48 01 d0       add     %rdx,%rax
1379: 89 ca          mov     %ecx,%edx
137b: 88 10          mov     %dl,(%rax)
137d: e8 0e fe ff ff call    1190 <rand@plt>
1382: 83 e0 07       and     $0x7,%eax
1385: 89 c1          mov     %eax,%ecx
1387: 48 8b 55 d0     mov     -0x30(%rbp),%rdx
138b: 48 8b 45 e8     mov     -0x18(%rbp),%rax
138f: 48 01 d0       add     %rdx,%rax
1392: 0f b6 00       movzbl (%rax),%eax
1395: 0f b6 c0       movzbl %al,%eax
1398: 48 8b 75 d0     mov     -0x30(%rbp),%rsi
139c: 48 8b 55 e8     mov     -0x18(%rbp),%rdx
13a0: 48 01 f2       add     %rsi,%rdx
13a3: 88 45 c7       mov     %al,-0x39(%rbp)
13a6: 89 4d cc       mov     %ecx,-0x34(%rbp)
13a9: 0f b6 45 c7     movzbl -0x39(%rbp),%eax
13ad: 89 c6          mov     %eax,%esi
13af: 8b 45 cc       mov     -0x34(%rbp),%eax
13b2: 89 c1          mov     %eax,%ecx
13b4: 40 d2 c6       rol     %cl,%sll
13b7: 89 f0          mov     %esi,%eax
13b9: 88 02          mov     %al,(%rdx)
13bb: 48 83 45 d0 01 addq    $0x1,-0x30(%rbp)
13c0: 48 8b 45 d0     mov     -0x30(%rbp),%rax
13c4: 48 3b 45 e0     cmp     -0x20(%rbp),%rax
13c8: 7c 86          jle     1350 <main+0xc7>

```

In this section the 'xor' action can be found from line 1350 with the random value key (1 byte in size) generating, to line 136e with the xor operation itself, and storing it in the i index of the byte array, replacing the original char.

The left rotation can be found from line 137b with the call to the second random number generator. Then we slice the last 3 bits of which with 'and' operation in order to create a random key in value range of 0 to 7. The reason 7 is used for the 'and' operation is because a byte can be rotated for maximum 8 different ways. From 0 steps rotation to 7 steps rotation. 8 steps rotation would be equal to 0 steps rotation, due its circular nature. The rotation itself is being performed on line 13b4 and its storage in the byrearray in the relative index in the subsequencing lines, up to line 13c8. After this action – the byte array containing the encrypted string.

- c. Writing up the encrypted data, and 4 bytes timestamp to 'flag.enc' output file.

```

13ca: 48 8d 35 3b 0c 00 00 lea 0xc3b(%rip),%rsi # 200c <_IO_stdin_used+0xc>
13d1: 48 8d 3d 37 0c 00 00 lea 0xc37(%rip),%rdi # 200f <_IO_stdin_used+0xf>
13d8: e8 93 fd ff ff call 1170 <fopen@plt>
13dd: 48 89 45 f0 mov %rax,-0x10(%rbp)
13e1: 48 8b 55 f0 mov -0x10(%rbp),%rdx
13e5: 48 8d 45 c8 lea -0x38(%rbp),%rax
13e9: 48 89 d1 mov %rdx,%rcx
13ec: ba 04 00 00 00 mov $0x4,%edx
13f1: be 01 00 00 00 mov $0x1,%esi
13f6: 48 89 c7 mov %rax,%rdi
13f9: e8 82 fd ff ff call 1180 <fwrite@plt>
13fe: 48 8b 55 e0 mov -0x20(%rbp),%rdx
1402: 48 8b 4d f0 mov -0x10(%rbp),%rcx
1406: 48 8b 45 e8 mov -0x18(%rbp),%rax
140a: be 01 00 00 00 mov $0x1,%esi
140f: 48 89 c7 mov %rax,%rdi
1412: e8 69 fd ff ff call 1180 <fwrite@plt>
1417: 48 8b 45 f0 mov -0x10(%rbp),%rax
141b: 48 89 c7 mov %rax,%rdi
141e: e8 dd fc ff ff call 1100 <fclose@plt>
1423: b8 00 00 00 00 mov $0x0,%eax
1428: 48 8b 7d f8 mov -0x8(%rbp),%rdi
142c: 64 48 33 3c 25 28 00 xor %fs:0x28,%rdi
1433: 00 00
1435: 74 05 je 143c <main+0x1b3>
1437: e8 d4 fc ff ff call 1110 <__stack_chk_fail@plt>
143c: c9 leave
143d: c3 ret
143e: 66 90 xchg %ax,%ax

```

First – the timestamp is being written in the lines 13d8 to 13f9.

It can be observed that in line 13ec 4 bytes were loaded to the appropriate register, indicating the writing was for 4 bytes.

Then – in line 13fe to 1412 the writing of the encrypted byte array took place. Giving us the output file structure that we observed.

In order to illuminate further the ‘encryptor’ process:

I’ve attached pseudo c code of the reversed engineered assembly:

```

2 int main(){
3     File *f = fopen("flag", "rb");
4
5     fseek(f, 0, 2);
6     int str_size = ftell(f);
7     fseek(f,0,0); //return file reader to its initial position
8     m = malloc(str_size);
9     fread(m, str_size, 1, f);
10    fclose(f);
11    time_t time = time(NULL);
12    srand(time);
13
14    while(int i = 0; i < str_size; i++){
15        int rand_val = rand();
16        byte lowval = rand_val.get low byte) //possible values = 0-255
17        int some_xor = xor(lowval, m[i])..
18        m[i] = some_xor.lowByte;
19
20        int val2 = rand();
21        val2 = val2 & 7 -> last 3 bits (values 0 to 7);
22        rf = rotate_left(some_xor, val2) //rotate left m[i].lowByte, val2 times
23        m[i] = rf;
24    }
25
26    File *output = fopen("flag.enc", "wb");
27    fwrite(&time,4,1,output);
28    fwrite(m,str_size,1,output);
29    fclose(output);
30 }

```

Decryption: Now that we know how the ‘encrypt’ works – its time to build the ‘decryptor’.

In order to Decrypt the binary string – I need to take all the bytes from index 4 onward – meaning from the fifth element.

For every byte – I need to use the exact same random values that were generated for this byte operations, The first one (lets call it v1) that was generated in the encryptor for the xor string, and the second one (lets call it v2) that was used for the left rotation.

The decrypting, for every byte will first rotate the byte v2 times to the opposite direction – right. And then will xor it with v1, giving the original value, as xor is reversible operation.

The main issue I've encountered is how to deal with the random values that were generated as keys.

So, the first approach I've tried is to build a python script – that for every byte – cover the reverse process with all possible keys: right rotation for every 0-7 values, then 'xor' with every 0-255 values. And picking only the result bytes which are Ascii readable. However, that took too long, as we are dealing with 8x256xstring length combinations. And there were too many valid Ascii combinations of them. So that first approach was abandoned.

In order to figure out the random seed challenge – I tried a different approach with the provided timestamp.

At the time, I was not sure it will work – as I wasn't certain that even with recreating the seed with the given timestamp – The exact same random values will indeed generate the same random values that were used in the original encrypting.

However I haven't got at the time good alternatives, some approaches of inspecting further the 'rand()' function, searching its implementation online or reverse engineering it were considered but abandoned.

So I have decided to try it – I've built a decryptor in c that takes the flag.enc content, does the actions that were mentioned as necessary to decrypt the bytes, with the provided timestamp as seed. The output of which were saved in the original 'flag' file (the same 'flag' that is used as input for encrypt).

However it didn't work – the output of the decrypted was 'Gibberish' – non readable ASCII.

I wasn't sure what caused it – after examination the of the File read/write, seed creation and xor/and/rotate operation – everything worked as expected.

So I suspected the problem was with the rand() – maybe the timestamp didn't work after all and the random values generated were not dependent of which.

After a long time and arduous effort of examination – I've came to the conclusion that in order for the rand() to generate the exact same values as the 'encryptor' – a vital part of the decryption, they have to be generated in the same order.

In the original implementation of the decryption – the 'v2' variable used for the 'and' operation key creation for the rotation – is being initialized before 'v1' – the 'xor' random key.

It seemed obvious as in the decryption the rotation operation is being performed before the xor operation, so it would seem natural to initialize 'v2' first.

However as mentioned – the order of 'rand()' initialization for every byte matters - and 'v1' still has to be initialized before v2.

```
val2 = rand();  
val1 = rand() & 7;  
  
m[i] = rotate_right(m[i], val1);  
m[i] = val2 ^ m[i];
```

After fixing the issue – I've tried to run it again, and got the decrypted flag:



Decrypting is Finished. Challenge completed.

Conclusions: This capture the flag challenge was indeed challenging.

The reverse engineering the code took its fair share of time and effort, but I've got it through. The main challenge was to deal with the 'rand()' function – which I took me a while to figure out how to utilize it, and the time based seed properly for the decryption.

The main conclusion is that same seed will yield the same random values. And the order of which matters – the order of 'rand()' generation in the encryption algorithm, has to be the same order for the decryption algorithm.

Another conclusion from the CTF is the behavior of the rotation operations, this operation I knew scarcely about before the CTF, so I had to take a fair share of research in order to understand its behavior and properties properly.

Another valuable aspect is Thanks to this CTF I've deepened my skill in file handling in C, especially regarding to its many operations ('fseek', 'ftell'...)

I will make sure to take forward everything I've learned.

*Relevant materials used for solving the challenge can be found [here](#).