

Encryption Bot

Capture the Flag Challenge.

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

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

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

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

Upon running the encrypting program with some arbitrary string – we get the message that the program will encrypt only with specific length:

```

amit@amit-VirtualBox: ~/Downloads/capture the flags/Encr...
amit@amit-VirtualBox:~/Downloads/capture the flags/Encryption Bot$ ./chall

#####
# # # #### ##### # # ##### ##### # #### # # # #####
# #####
# ## # # # # # # # # # # # # # # # # # # # #
# #
##### # # # # # # # # # # # # # # # # # # # # ##### #
# #
# # # # ##### # ##### # # # # # # # # # # # #
# #
# # # # # # # # # # # # # # # # # # # # # #
##### # # ##### # # # # # # # # # # # # # # #####
# #

```

Enter the text to encrypt : somestringtoencrypt

I'm encrypt only specific length of character.
 (._.) Find it (._.)

```

amit@amit-VirtualBox:~/Downloads/capture the flags/Encrption Bot$

```

So we have to inspect closely the encryption program and the encrypted flag to extract the flag.

Method:

The first action that I did is to run the 'file' command on the program:

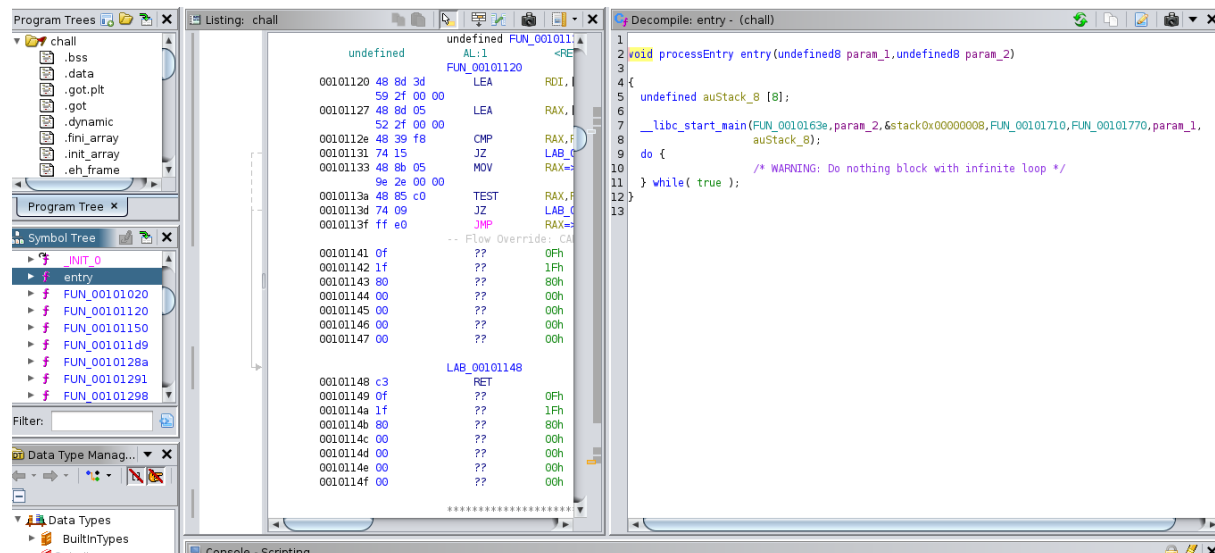
```

amit@amit-VirtualBox:~/Downloads/capture the Flags/Encryption Bot$ file chall
chall: ELF 64-bit LSB pie executable, x86_64, version 1 (SYSV), dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2,
BuildID[sha1]=581371f680358611dc4e8db77b03858bd4c780174, for GNU/Linux 3.2.0, stripped

```

We are dealing here with stripped 64bit ELF, little endian. Meaning the program is not built for debugging. So I will use 'Ghidra' in order to statically analyze the file.

I launched Ghidra and decompiled the program, the first thing I looked is for the entry function:



It can be observed that the function 'FUN_0010163e' is being executed by the entry, that's our main function.

Let's open the decompiled function, now renamed to main:

```
1
2 Undefined8 main(void)
3
4 {
5     undefined local_38 [32];
6     FILE *local_18;
7     undefined4 local_c;
8
9     FUN_00101628();
10    local_c = 0;
11    printf("\n\nEnter the text to encrypt : ");
12    __isoc99_scanf(&DAT_0010230f, local_38);
13    local_18 = fopen("data.dat", "r");
14    if (local_18 != (FILE *)0x0) {
15        system("rm data.dat");
16    }
17    putchar(10);
18    FUN_001015dc(local_38);
19    FUN_0010128a();
20    FUN_0010131d(local_38);
21    FUN_001014ba();
22    system("rm data.dat");
23    putchar(10);
24    return 0;
25 }
26
```

We can see that among file handling, some variables initializations and printf, there are some other nameless functions. Let's inspect them:

The first function (line 9) is the 'drawing' printing:

[illegible]

Which is irrelevant.

In line 12 we can observe the scanf function, that puts the user's input into 'local_38' variable, lets call it 'user_input'.

In addition there is also file pointer variable that points to file called 'data.dat', at this point we don't know the purpose of which. So lets call it for now 'workfile'.

```

1
2 undefined8 main(void)
3
4 {
5     undefined user_input [32];
6     FILE *workfile;
7     undefined4 local_c;
8
9     irrelevant_drawing();
10    local_c = 0;
11    printf("\n\nEnter the text to encrypt : ");
12    _isoc99_scanf(&AT_0010230f, user_input);
13    workfile = fopen("data.dat", "r");
14    if (workfile != (FILE *)0x0) {
15        system("rm data.dat");
16    }
17    putchar(10);
18    FUN_001015dc(user_input);
19    FUN_0010128a();
20    FUN_0010131d(user_input);
21    FUN_001014ba();
22    system("rm data.dat");
23    putchar(10);
24    return 0;
25 }

```

That's better. Now lets take a look at the function at line 18, that takes in the user's input:

```

1 | Decompile: FUN_001015dc - (chall)
2 |
3 |
4 | void FUN_001015dc(char *param_1)
5 |
6 | {
7 |     size_t sVar1;
8 |
9 |     sVar1 = strlen(param_1);
10 |    if ((int)sVar1 != 0x1b) {
11 |        puts("I\'m encrypt only specific length of character.");
12 |        puts("(-_-) Find it (-_-)");
13 |        /* WARNING: Subroutine does not return */
14 |        exit(1);
15 |    }
16 |    return;
17 | }

```

We can clearly see that it checks for the string's length. And if the string's length is not 0x1b, or 27 characters long, we get the wrong length message and exit, as we did in the initial run.

Let's confirm that 27 is indeed the required string length:

[illegible]

When 27 characters long string was given – we indeed got the encryption of which, and we can observe it is 36 characters long.

Now lets proceed and see how the encryption works:

The function on line 19 seems to be irrelevant:

```
1 void FUN_0010128a(void)
2 {
3     return;
4 }
```

Now let's take a look at the function in line 20, that takes in the user's input:

```

1
2 undefined8 FUN_0010131d(char *param_1)
3
4 {
5     size_t sVar1;
6     ulong uVar2;
7     undefined local_868 [2000];
8     int aiStack_98 [31];
9     int local_1c;
10
11     local_1c = 0;
12     while( true ) {
13         uVar2 = (ulong)local_1c;
14         sVar1 = strlen(param_1);
15         if (sVar1 <= uVar2) break;
16         aiStack_98[local_1c] = (int)param_1[local_1c];
17         FUN_001011d9(aiStack_98[local_1c], local_868);
18         local_1c = local_1c + 1;
19     }
20     FUN_0010129f();
21     return 0;
22 }

```

It can be observed that:

param_1 is the user's string.

sVar1 is the string length.

uVar2 and local_1c are some counters, from 0 to string's length.

aiStack_98 that holds the ascii values of the input characters.

After some renaming:

```
Decompile: characters_to_ascii_array - (chall)
1
2 undefined8 characters_to_ascii_array(char *param_1)
3
4 {
5     size_t str_length;
6     ulong counter;
7     undefined local_868 [2000];
8     int ascii_array_of_input [31];
9     int counter2;
10
11     counter2 = 0;
12     while( true ) {
13         counter = (ulong)counter2;
14         str_length = strlen(param_1);
15         if (str_length <= counter) break;
16         ascii_array_of_input[counter2] = (int)param_1[counter2];
17         FUN_001011d9(ascii_array_of_input[counter2], local_868);
18         counter2 = counter2 + 1;
19     }
20     FUN_0010129f();
21     return 0;
22 }
```

Every ascii value of the input's characters goes in to other function in line 17,

Let's take a look in it:

```
Decompile: FUN_001011d9 - (chall)
1
2 undefined8 FUN_001011d9(int param_1)
3
4 {
5     int local_6c;
6     uint auStack_68 [20];
7     FILE *local_18;
8     int local_10;
9     int local_c;
10
11     local_18 = fopen("data.dat", "a");
12     local_6c = param_1;
13     for (local_c = 0; local_c < 8; local_c = local_c + 1) {
14         auStack_68[local_c] = local_6c % 2;
15         local_6c = local_6c / 2;
16     }
17     for (local_10 = 7; -1 < local_10; local_10 = local_10 + -1) {
18         fprintf(local_18, "%d", (ulong)auStack_68[local_10]);
19     }
20     fclose(local_18);
21     return 0;
22 }
23
```

param_1 parameter is the input ascii value, and local_6c takes its value.

local_18 is the workfile.

Local_c and local_10 are iterators i and j.

au_Stack68 is unsigned integer array.

Lets rename the variables:

```
Decompile: FUN_001011d9 - (chall)
1
2 undefined8 FUN_001011d9(int ascii_input_parameter)
3
4 {
5     int ascii_input;
6     uint auStack_68 [20];
7     FILE *workfile;
8     int j;
9     int i;
10
11     workfile = fopen("data.dat", "a");
12     ascii_input = ascii_input_parameter;
13     for (i = 0; i < 8; i = i + 1) {
14         auStack_68[i] = ascii_input % 2;
15         ascii_input = ascii_input / 2;
16     }
17     for (j = 7; -1 < j; j = j + -1) {
18         fprintf(workfile, "%d", (ulong)auStack_68[j]);
19     }
20     fclose(workfile);
21     return 0;
22 }
```

In line 13 there is a loop of 8 iterations, that stores in the array in each iteration at index i, the value of the ascii input modulo 2. Then the ascii input value is being halved.

This is of course the standard procedure to get the binary value in reverse.

For example, the value 135 will store in the array the values:

[1, 1, 1, 0, 0, 0, 0, 1], which are the reverse of its binary: 10000111.

In line 17 we see writing to the workfile the array values, in descending order.

Which means – this functions writes to the workfile the binary of the ascii value, so it will be renamed 'write_binary_of_ascii_to_workfile'.

In function 'characters_to_ascii_array' there is also a function in line 20.

That function is also irrelevant with the same structure is the other irrelevant function, so we ignore it.

```
Decompile: FUN_0010128a
1
2 void FUN_0010128a(void)
3
4 {
5     return;
6 }
7
```

So, at this point what we know is that the input string is being converted to binary and written into a file.

Now we will proceed to inspect the function in line 21 in the main function:

```

Decompile: FUN_001014ba - (chall)
1
2 void FUN_001014ba(void)
3
4 {
5     int iVar1;
6     int aiStack_668 [400];
7     int local_28;
8     char local_21;
9     FILE *local_20;
10    int local_18;
11    int local_14;
12    int local_10;
13    int local_c;
14
15    local_20 = fopen("data.dat","r+");
16    FUN_00101291();
17    for (local_c = 1; local_c < 0xd9; local_c = local_c + 1) {
18        iVar1 = fgetc(local_20);
19        local_21 = (char)iVar1;
20        if (local_21 == '0') {
21            aiStack_668[local_c + -1] = 0;
22        }
23        else if (local_21 == '1') {
24            aiStack_668[local_c + -1] = 1;
25        }
26        if (local_c != 0) {
27            if (local_c % 6 == 0) {
28                local_14 = 0;
29                local_10 = local_c;
30                for (local_18 = 0; local_10 = local_10 + -1, local_18 < 6; local_18 = local_18 + 1) {
31                    local_28 = FUN_001013ab(local_18);
32                    local_14 = local_14 + aiStack_668[local_10] * local_28;
33                }
34                FUN_001013e9(local_14);
35            }
36        }
37    }
38    fclose(local_20);
39    return;
40 }

```

The function in line 16 is also one of the irrelevant function that does nothing, we ignore it.

We can see the variable `local_c` which is iterator from 1 to 0xd9 (217)

However, the loop does not includes the value 217 itself. So the loop iterates 216 times, which is 27×8 – so basically for every bit in the file.

From line 18 to line 25 we read from the file a character, which is either '0' or '1', and depend on the result – we store in the integer array 'aiStack_668' the integer cast of which, 0 or 1.

Before we examine the second part of this loop – lets take a look in the function in line 31:

```

Decompile: FUN_001013ab - (chall)
1
2 int FUN_001013ab(int param_1)
3
4 {
5     int local_1c;
6     int local_c;
7
8     local_c = 1;
9     for (local_1c = param_1; local_1c != 0; local_1c = local_1c + -1) {
10         local_c = local_c * 2;
11     }
12     FUN_0010129f();
13     return local_c;
14 }
15

```

That function returns 2 to the power of param_1.

Back to the encryption loop – if the iterator 'local_c' value is devisable by 6 - an internal loop of 6 iterations sums the values of the previous 6 indexes, multiplied with 2 to the power of the backward iterator.

Ok it will be easier to explain with example:

Lets say part of the array is [0,1,1,0,0,1], with the array indexes 0 to 5.

When i (or local_c) equals 6 – what we do is take the value of index 5 (1) and multiply it with 2^0 , The result is 1.

Then we take the value of index 4 (0), and multiply it with 2^1 , the result is 0.

Then we take the value of index 3 (0), and multiply it with 2^2 , the result is 0.

Then we take the value of index 2 (1), and multiply it with 2^3 , the result is 8.

Then we take the value of index 1 (1), and multiply it with 2^4 , the result is 16.

Then we take the value of index 0 (0), and multiply it with 2^5 , the result is 0.

Let's sum it all together: $16+8+1 = 25$.

Basically, what the code does is convert binary to integers. However with 6 bits representation instead of the traditional 8 bits.

Now as there are 217 bits in the workfile, the output is array of 36 integers.

every integer generated from the 6 bits binary – is being inserted to a function in line 34:

```
Decompile: FUN_001013e9 - (chall)
1
2 undefined8 FUN_001013e9(int param_1)
3
4 {
5     long lVar1;
6     undefined8 *puVar2;
7     undefined8 local_198;
8     undefined8 local_190;
9     undefined8 local_188;
10    undefined8 local_180;
11    undefined8 local_178;
12    undefined8 local_170;
13    undefined8 local_168;
14    undefined8 local_160;
15    undefined8 local_158 [42];
16
17    local_198 = 0x5958575655545352;
18    local_190 = 0x363534333231305a;
19    local_188 = 0x4544434241393837;
20    local_180 = 0x4d4c4b4a49484746;
21    local_178 = 0x6463626151504f4e;
22    local_170 = 0x6c6b6a6968676665;
23    local_168 = 0x74737271706f6e6d;
24    local_160 = 0x7a7978777675;
25    puVar2 = local_158;
26    for (lVar1 = 0x2a; lVar1 != 0; lVar1 = lVar1 + -1) {
27        *puVar2 = 0;
28        puVar2 = puVar2 + 1;
29    }
30    putchar((int)*(char *)((long)&local_198 + (long)param_1));
31    return 0;
32 }
33
```


We can observe that there are 7 long (8 bytes) integers, represented with hexadecimal, and one more 6 bytes integer. All with consecutive addresses.

Then there is a loop and another variable that we will ignore at the moment,

And in line 30 – we have the char print of some value – lets break it apart:

We take the address of local 198, and add to it the 6 bits integer input.

The result would be a new address.

Example: lets say the address of local_198 is 0x0000000000000000

And local_198 is 8 bytes long type.

So this is how it would be stored in little endian system:

0x00..00	0x00..01	0x00..02	0x00..03	0x00..04	0x00..05	0x00..06	0x00..07
0x52	0x53	0x54	0x55	0x56	0x57	0x58	0x59

The next long number address would be in this example 0x0000000000000008:

This is how memory would look with both integers:

0x00..00	0x00..01	0x00..02	0x00..03	0x00..04	0x00..05	0x00..06	0x00..07
0x52	0x53	0x54	0x55	0x56	0x57	0x58	0x59
0x00..08	0x00..09	0x00..0a	0x00..0b	0x00..0c	0x00..0d	0x00..0e	0x00..0f
0x5a	0x30	0x31	0x32	0x33	0x34	0x35	0x36

So, if we for example add 0x01 to the address of 0x0000000000000000:

The long that we will get would be:

0x5a59585756555453 – as we read from address 0x00...01.

The Byte that we will get in such a case would be 0x53.

By the same principal: 0x00...02 would give us 0x54.

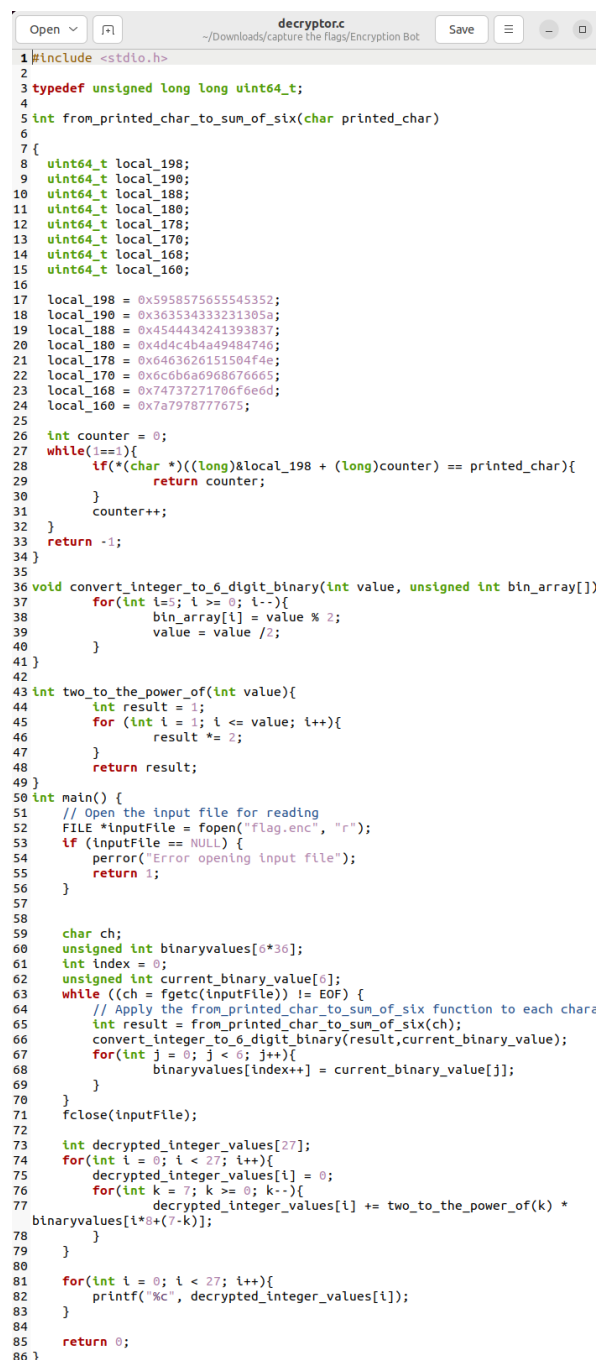
To summarize: the parm_1, the 6 bits ascii value we got from the workfile – is the number of steps we should walk from the base address.

Then – we take the Byte within the memory address, and print the character that this byte represents by ascii.

So if the function input is 2. we walk 2 steps from the base address (local_198 address), which means. we got 0x54. So the printed value would be 'T'.

That, is our encrypted output.

Now that encryption mechanism is clear – Lets build the decryption algorithm:



```
1 #include <stdio.h>
2
3 typedef unsigned long long uint64_t;
4
5 int from_printed_char_to_sum_of_six(char printed_char)
6
7 {
8     uint64_t local_198;
9     uint64_t local_190;
10    uint64_t local_188;
11    uint64_t local_180;
12    uint64_t local_178;
13    uint64_t local_170;
14    uint64_t local_168;
15    uint64_t local_160;
16
17    local_198 = 0x5958575655545352;
18    local_190 = 0x3635343332313050;
19    local_188 = 0x4544434241393837;
20    local_180 = 0x4d4c4b4a49484746;
21    local_178 = 0x6463626151504f4e;
22    local_170 = 0x6c6b6a6968676665;
23    local_168 = 0x74737271706f6e6d;
24    local_160 = 0x7a7978777675;
25
26    int counter = 0;
27    while(1){
28        if(*(char *)((long)&local_198 + (long)counter) == printed_char){
29            return counter;
30        }
31        counter++;
32    }
33    return -1;
34 }
35
36 void convert_integer_to_6_digit_binary(int value, unsigned int bin_array[])
37 {
38     for(int i=5; i >= 0; i--){
39         bin_array[i] = value % 2;
40         value = value / 2;
41     }
42 }
43
44 int two_to_the_power_of(int value){
45     int result = 1;
46     for (int i = 1; i <= value; i++){
47         result *= 2;
48     }
49     return result;
50 }
51
52 int main() {
53     // Open the input file for reading
54     FILE *inputFile = fopen("flag.enc", "r");
55     if (inputFile == NULL) {
56         perror("Error opening input file");
57         return 1;
58     }
59
60     char ch;
61     unsigned int binaryvalues[0*30];
62     int index = 0;
63     unsigned int current_binary_value[0];
64     while ((ch = fgetc(inputFile)) != EOF) {
65         // Apply the from_printed_char_to_sum_of_six function to each char
66         int result = from_printed_char_to_sum_of_six(ch);
67         convert_integer_to_6_digit_binary(result, current_binary_value);
68         for(int j = 0; j < 6; j++){
69             binaryvalues[index++] = current_binary_value[j];
70         }
71         fclose(inputFile);
72
73         int decrypted_integer_values[27];
74         for(int i = 0; i < 27; i++){
75             decrypted_integer_values[i] = 0;
76             for(int k = 7; k >= 0; k--){
77                 decrypted_integer_values[i] += two_to_the_power_of(k) *
78                 binaryvalues[i*8+(7-k)];
79             }
80         }
81         for(int i = 0; i < 27; i++){
82             printf("%c", decrypted_integer_values[i]);
83         }
84
85         return 0;
86 }
```

The decryption algorithm does the reverse of the encryption program:

It reads the encrypted string from 'flag.enc', which contains the encrypted characters.

For every character in it:

It takes encrypted char as an input, and counts the number of steps it took to walk over the addresses until the character whose ascii value on the long variable matches the encrypted character, which in decryption is the input.

Then, the number of steps output is being converted to 6 bits binary and stored in its place in 216 bits long array.

Then, we take the 216 bits array, and extract from it the characters value, this time for every 8 bits. We use the same method used by the encrypting program.

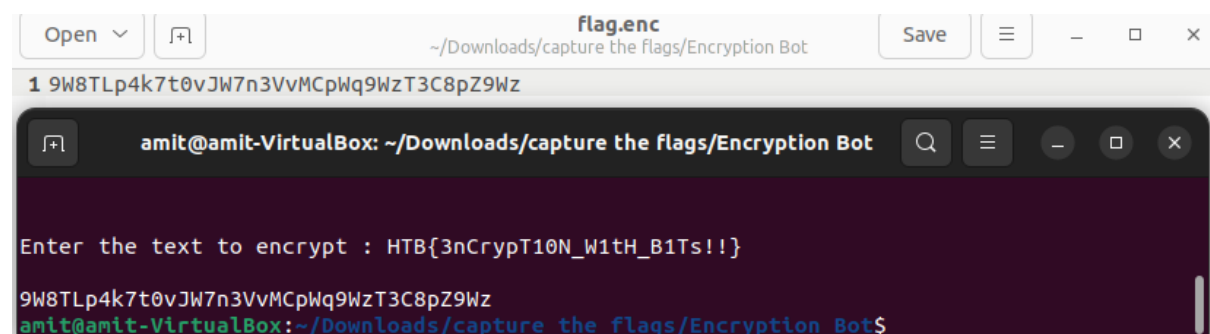
The output of which, is the original string – what should be the flag.

Let's run the decryptor:

```
amit@amit-VirtualBox:~/Downloads/capture the flags/Encryption Bot$ gcc decryptor.c -o d
amit@amit-VirtualBox:~/Downloads/capture the flags/Encryption Bot$ ./d
HTB{3nCrypT10N_W1tH_B1Ts!!}amit@amit-VirtualBox:~/Downloads/capture the flags/Encryption
amit@amit-VirtualBox:~/Downloads/capture the flags/Encryption Bot$
```

Success! 'flag.enc' had been successfully decrypted, and we got the flag!

Let's confirm the decrypted flag by running it on the encryptor and compare the result to 'flag.enc':



The screenshot shows a terminal window titled 'flag.enc' with the path '~/Downloads/capture the flags/Encryption Bot'. The terminal output shows the encryption of the decrypted flag 'HTB{3nCrypT10N_W1tH_B1Ts!!}' resulting in '9W8TLp4k7t0vJW7n3VvMCpWq9WzT3C8pZ9Wz'. The terminal prompt is 'amit@amit-VirtualBox:~/Downloads/capture the flags/Encryption Bot\$'.

Perfect match, that is indeed the flag.

Conclusions: The main thing I learned from this challenge is the importance of 'Ghidra' and decompilation. As the program is stripped, attempting to solve it with 'GDB' and manual 'objdump' was extremely hard.

Also, the logic of the encryption was a fair challenge. Not too difficult, but many small details that it was important to observe.

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