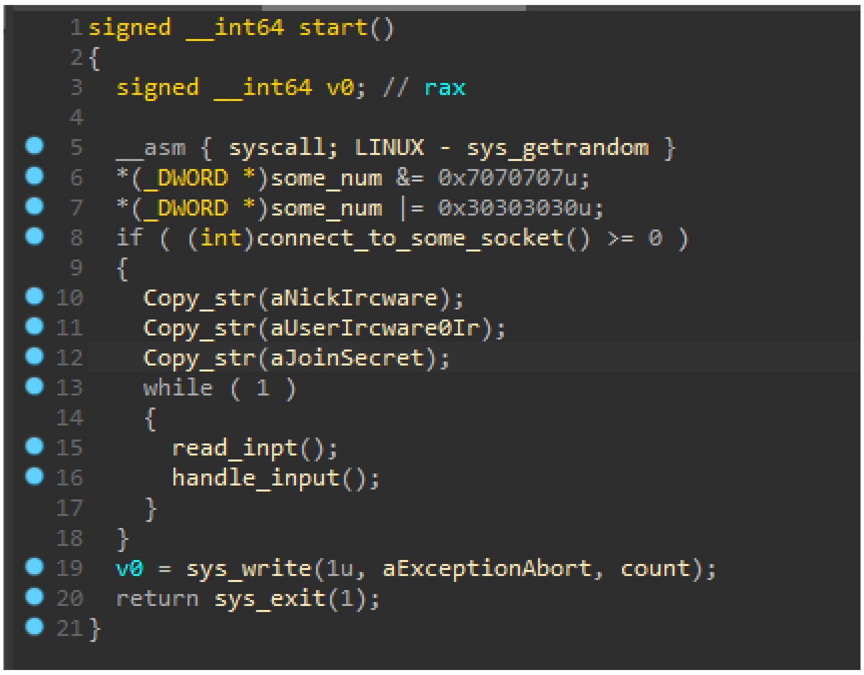
What is it?

Big chungus that won't run on purpose. We are forced to reverse it statically in order to get the flag. Effectively, this program is a sort-of simulator for IRC communication over a socket that can send us a flag. I had to utilize my 300iq guess god skills because I did a speedrun of decompilation reading.

How to solve it?

We begin our journey at the entry point:



```
signed __int64 start()
{
  signed __int64 v0; // rax

  __asm { syscall; LINUX - sys_getrandom }
  *(_DWORD *)some_num &= 0x7070707u;
  *(_DWORD *)some_num |= 0x30303030u;
  if ( (int)connect_to_some_socket() >= 0 )
  {
    Copy_str(aNickIrcware);
    Copy_str(aUserIrcware0Ir);
    Copy_str(aJoinSecret);
    while ( 1 )
    {
      read_inpt();
      handle_input();
    }
  }
  v0 = sys_write(1u, aExceptionAbort, count);
  return sys_exit(1);
}
```

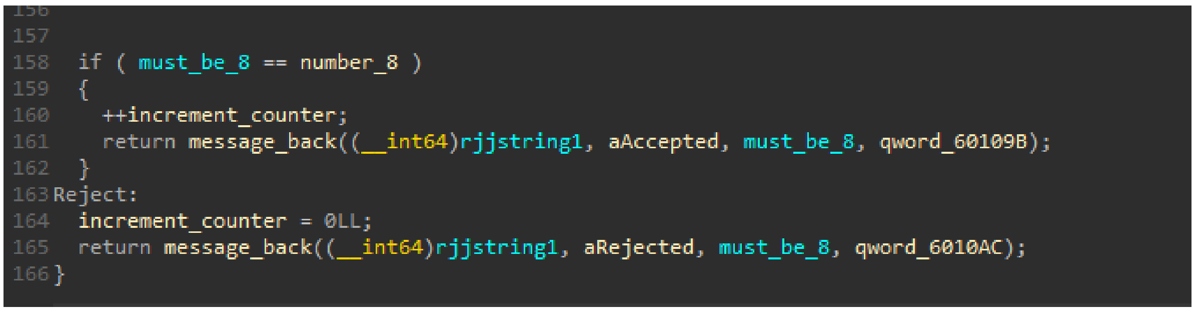
I've renamed some functions to describe what they do. In short, we set up a socket, connect to it, copy some strings into memory, and run the core functionality of the program. At every step it reads our input and does something to it :O

handle_input() handles our input, and does various things depending on what we send it!

Similarly to IRC, we can do a couple of things:

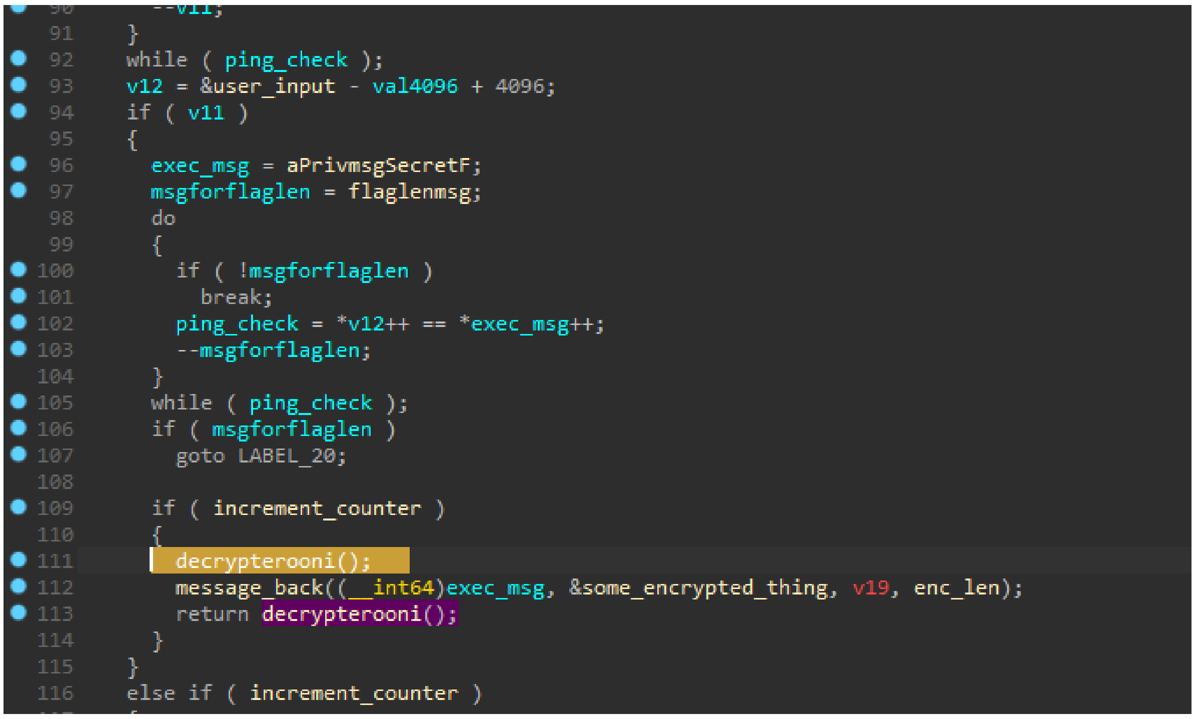
- Send a PING message which tells us to "Join #secret", a channel in irc we are simulating
- PRIVMSG (private message) #secret with specific parameters
  - :@pass -> We can supply a password that is checked against something
  - :@exec -> No clue, ask admins
  - :@flag -> Decrypts flagerooni and sends it as a msg :)

So we need some sort of password that can get Accepted or Rejected:



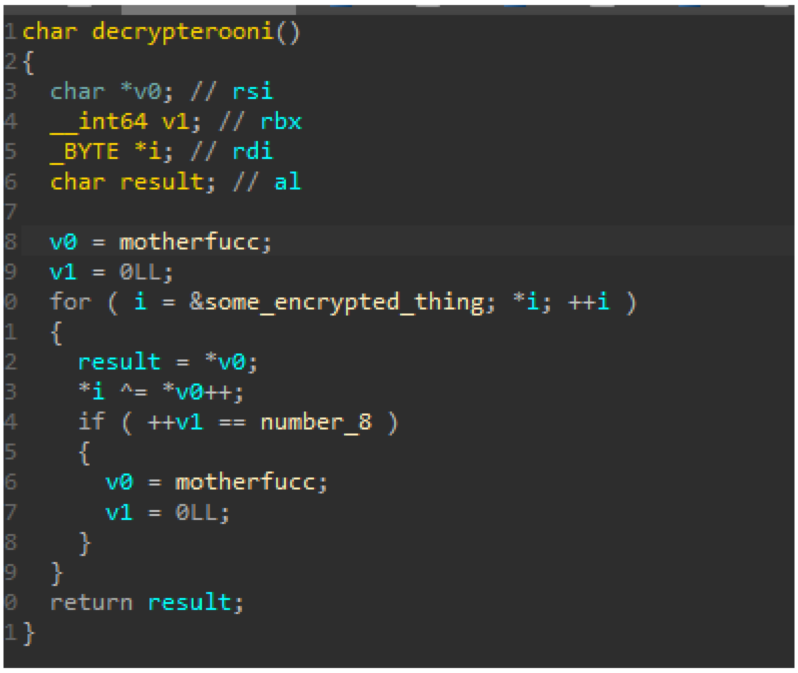
```

  if ( must_be_8 == number_8 )
  {
    ++increment_counter;
    return message_back((__int64)rjjstring1, aAccepted, must_be_8, qword_60109B);
  }
Reject:
  increment_counter = 0LL;
  return message_back((__int64)rjjstring1, aRejected, must_be_8, qword_6010AC);
}
```

Without looking further, I also noticed that asking for the flag decrypts it using this:



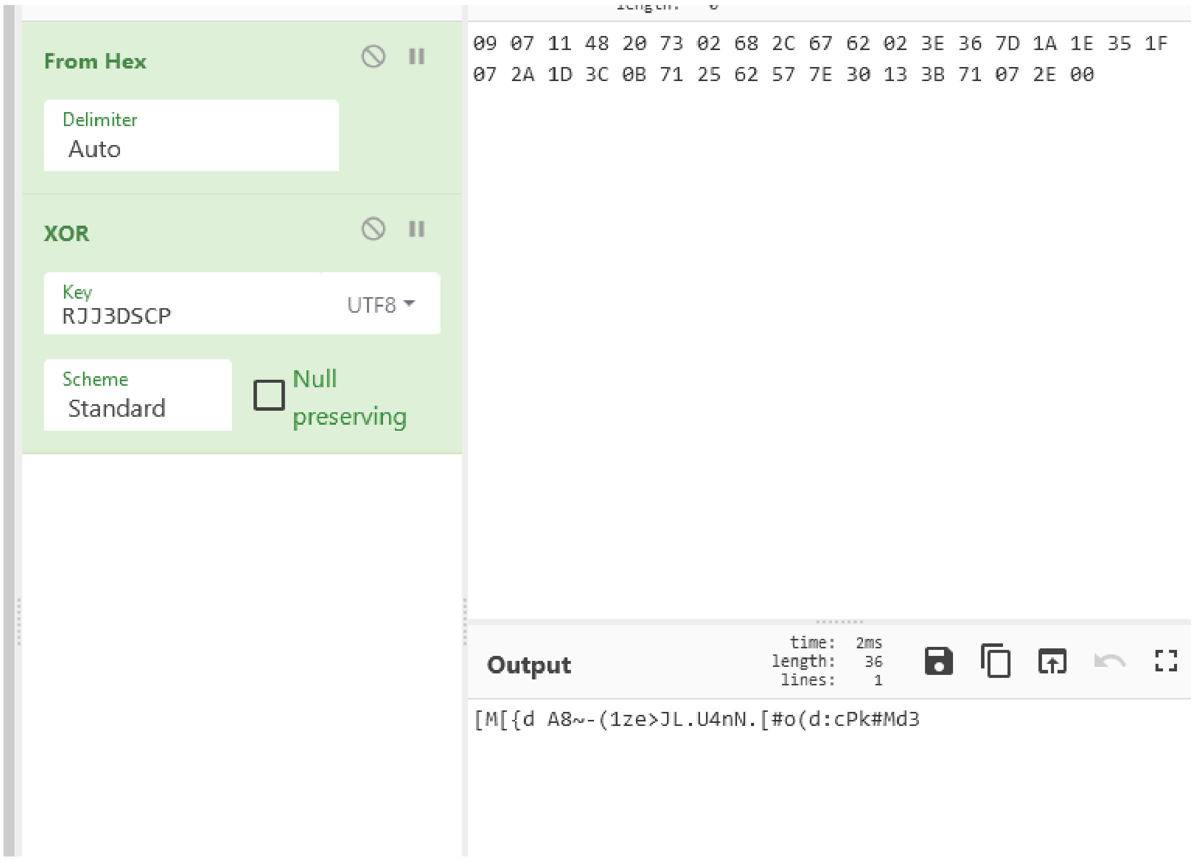
```
    --v11;
  }
  while ( ping_check );
  v12 = &user_input - val4096 + 4096;
  if ( v11 )
  {
    exec_msg = aPrivmsgSecretF;
    msgforflaglen = flaglenmsg;
    do
    {
      if ( !msgforflaglen )
        break;
      ping_check = *v12++ == *exec_msg++;
      --msgforflaglen;
    }
    while ( ping_check );
    if ( msgforflaglen )
      goto LABEL_20;

    if ( increment_counter )
    {
      decrypterooni();
      message_back((__int64)exec_msg, &some_encrypted_thing, v19, enc_len);
      return decrypterooni();
    }
  }
  else if ( increment_counter )
```

Where decrypterooni is just simple XOR using the variable *motherfucc* as the xor key:



```
char decrypterooni()
{
  char *v0; // rsi
  __int64 v1; // rbx
  _BYTE *i; // rdi
  char result; // al

  v0 = motherfucc;
  v1 = 0LL;
  for ( i = &some_encrypted_thing; *i; ++i )
  {
    result = *v0;
    *i ^= *v0++;
    if ( ++v1 == number_8 )
    {
      v0 = motherfucc;
      v1 = 0LL;
    }
  }
  return result;
}
```

Now, *motherfucc* is specified to be **RJJ3DSCP** by default, so let's try to XOR the hex bytes in *some_encrypted_thing* using that:



From Hex

Delimiter: Auto

XOR

Key: RJJ3DSCP (UTF8)

Scheme: Standard

Null preserving

```
09 07 11 48 20 73 02 68 2C 67 62 02 3E 36 7D 1A 1E 35 1F
07 2A 1D 3C 0B 71 25 62 57 7E 30 13 3B 71 07 2E 00
```

Output

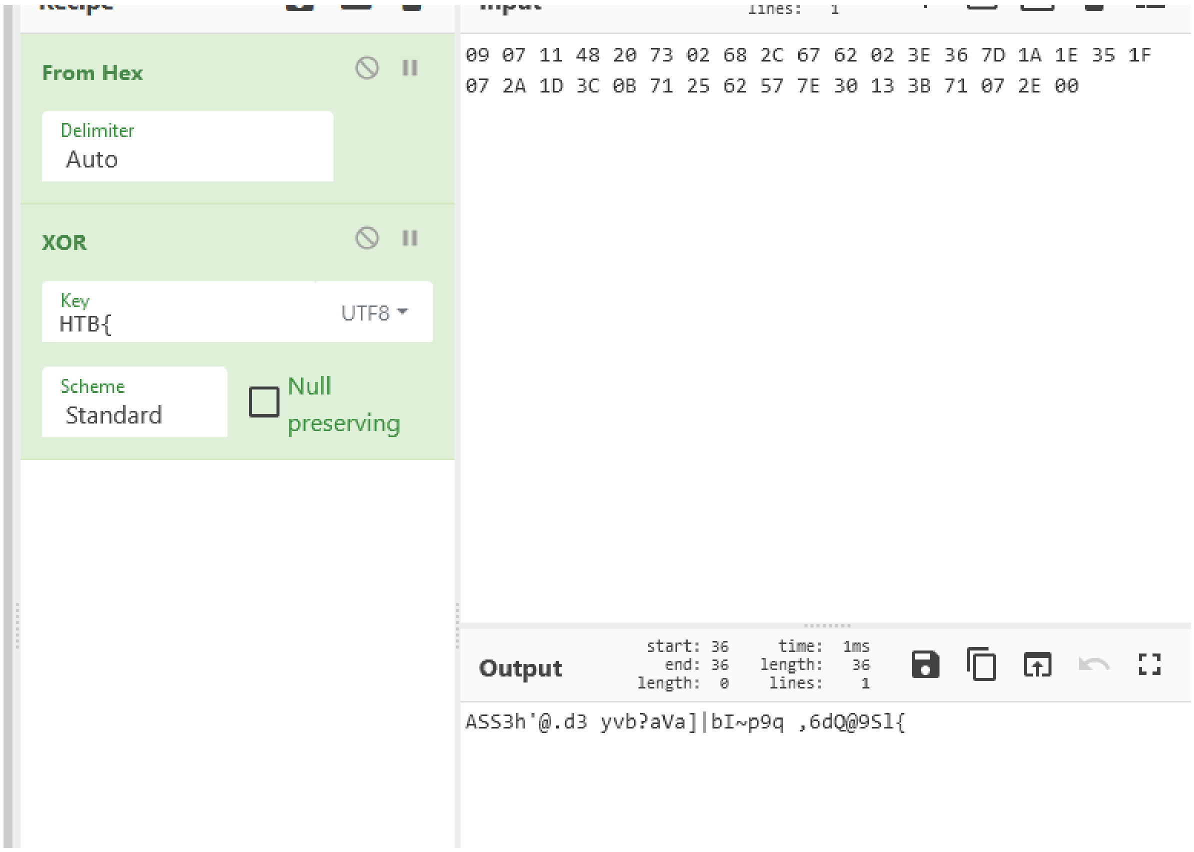
```
time:    2ms
length:   36
lines:     1
```

```
[M[{d A8~-(1ze>JL.U4nN.[#o(d:cPk#Md3
```

Uhhhh....

motherfucc indeed.

My next tactic, instead of actually carefully reversing the program, was to try and guess the password by xoring the encrypted hex bytes using the first part of the expected flag **HTB{**. The results were astounding, we got: **ASS**3.


From Hex
Delimiter: Auto

XOR
Key: HTB{ (UTF8)
Scheme: Standard
Null preserving: ☐

Input

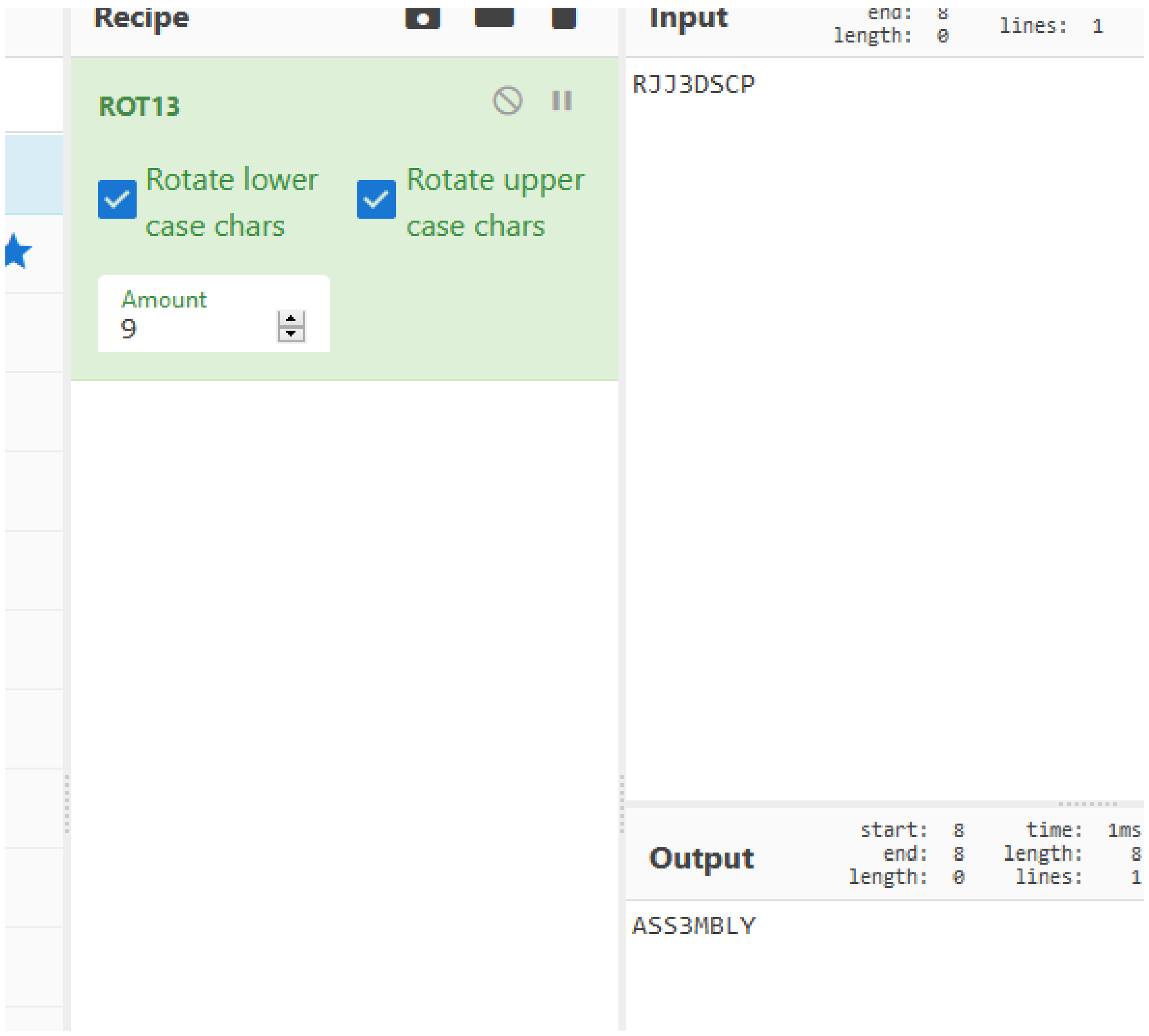
```
09 07 11 48 20 73 02 68 2C 67 62 02 3E 36 7D 1A 1E 35 1F
07 2A 1D 3C 0B 71 25 62 57 7E 30 13 3B 71 07 2E 00
```

Output

```
 start: 36     time:  1ms
   end: 36   length:   36
length:  0    lines:    1
```

```
ASS3h'@.d3 yvb?aVa]|bI~p9q ,6dQ@9Sl{
```

But wait! My spidey senses are tingling. Notice the resemblance between **ASS** and **RJJ**?! Let's now see if we can convert **RJJ3DSCP** to something starting with **ASS3**. My two braincells managed (by failing multiple times) to guess that ROT13 could be the solution, and by trying every possible combination we get our lord and saviour to be ROT9:



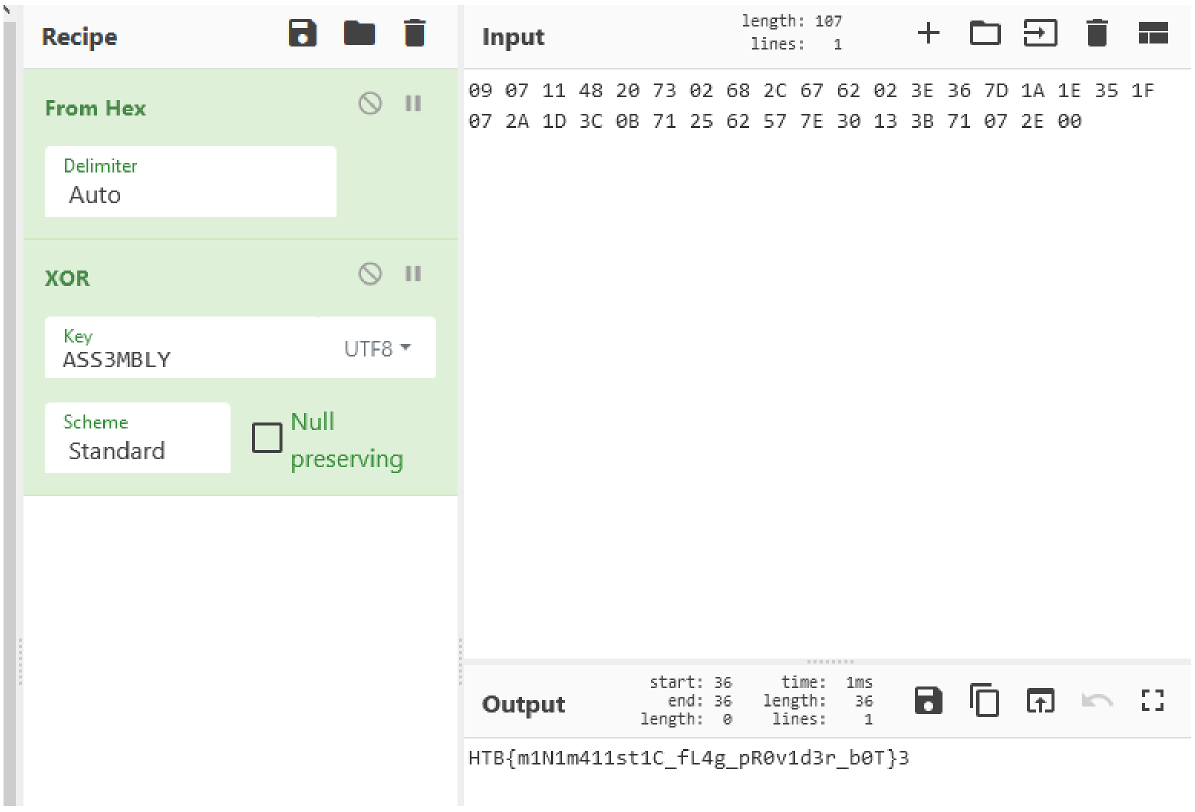
ROT13
Rotate lower case chars [✓] Rotate upper case chars [✓]
Amount 9

```
end: 8
length: 0
lines: 1

RJJ3DSCP
```

```
start: 8    time: 1ms
  end: 8  length:   8
length: 0   lines:   1

ASS3MBLY
```

So **ASS** was **ASS3MBLY** all along! Haha get it? Because Assembly is ASS? No?? Ok...

Either way, because its a readable string I try to xor the encrypted bytes with **ASS3MBLY** and wowowowowowo:



From Hex
Delimiter Auto

XOR
Key ASS3MBLY UTF8
Scheme Standard
Null preserving

```
length: 107
lines:   1

09 07 11 48 20 73 02 68 2C 67 62 02 3E 36 7D 1A 1E 35 1F
07 2A 1D 3C 0B 71 25 62 57 7E 30 13 3B 71 07 2E 00
```

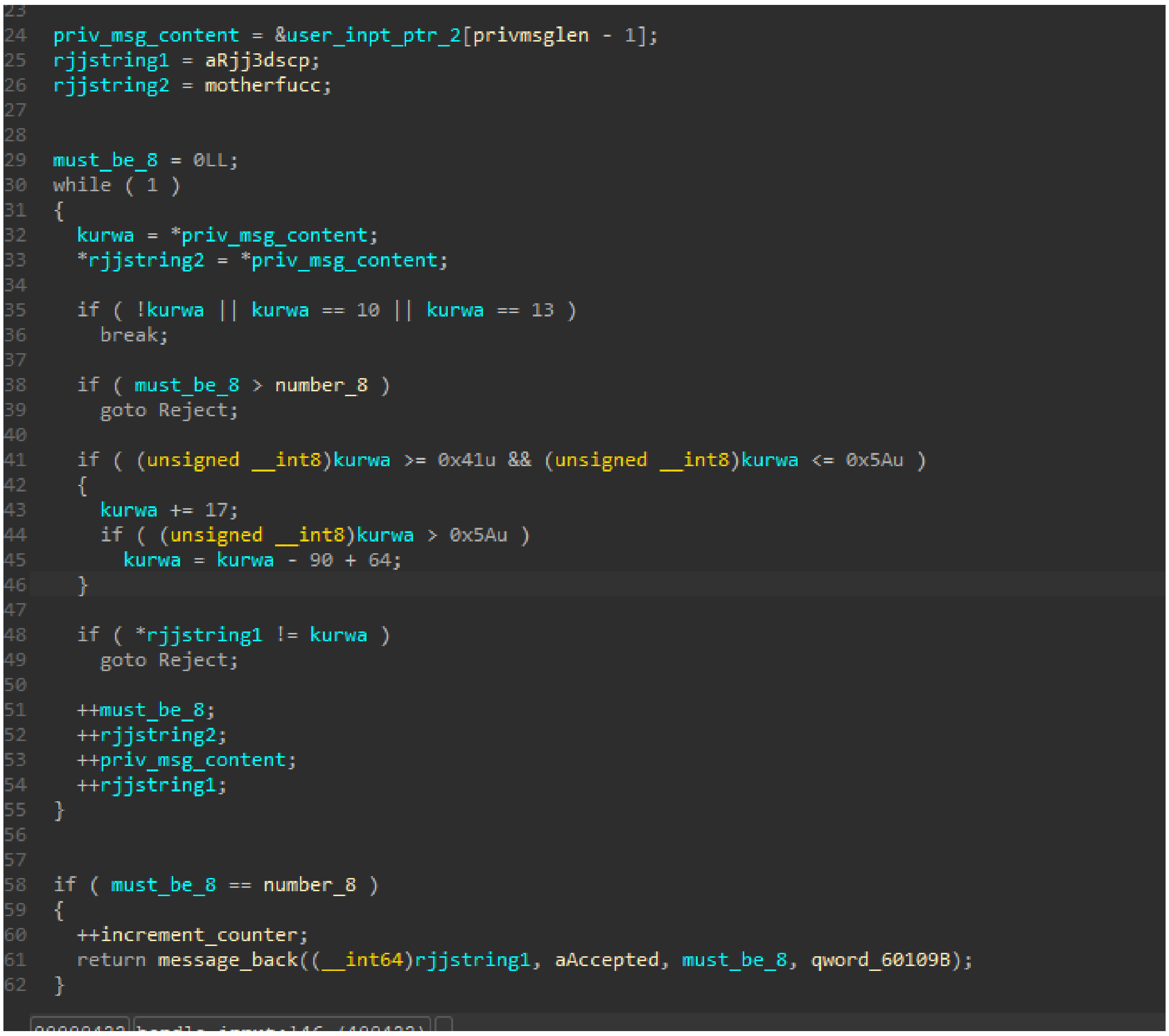
```
start: 36    time: 1ms
  end: 36  length:  36
length: 0   lines:   1

HTB{m1N1m411st1C_fL4g_pR0v1d3r_b0T}3
```

FLAG: HTB{m1N1m411st1C_fL4g_pR0v1d3r_b0T}

The morale of the story is that everyone needs ass.

SIDE NOTE:

We can actually see that ROT9 is applied to the password, by investigating how the password we send using PRIVMSG is actually handled, taking *kurwa* to be our input, we can see that the following happens:



```
23
24  priv_msg_content = &user_inpt_ptr_2[privmsglen - 1];
25  rjjstring1 = aRjj3dscp;
26  rjjstring2 = motherfucc;
27
28
29  must_be_8 = 0LL;
30  while ( 1 )
31  {
32    kurwa = *priv_msg_content;
33    *rjjstring2 = *priv_msg_content;
34
35    if ( !kurwa || kurwa == 10 || kurwa == 13 )
36      break;
37
38    if ( must_be_8 > number_8 )
39      goto Reject;
40
41    if ( (unsigned __int8)kurwa >= 0x41u && (unsigned __int8)kurwa <= 0x5Au )
42    {
43      kurwa += 17;
44      if ( (unsigned __int8)kurwa > 0x5Au )
45        kurwa = kurwa - 90 + 64;
46    }
47
48    if ( *rjjstring1 != kurwa )
49      goto Reject;
50
51    ++must_be_8;
52    ++rjjstring2;
53    ++priv_msg_content;
54    ++rjjstring1;
55  }
56
57
58  if ( must_be_8 == number_8 )
59  {
60    ++increment_counter;
61    return message_back((__int64)rjjstring1, aAccepted, must_be_8, qword_60109B);
62  }
```

The crucial parts are lines *25*, and *43-45*. *kurwa* becomes set to the memory that we use as the xor key for flag decryption, but it also gets 9 subtracted from each character (because -90+64+17=-9). And so, this implies that we need to ROT9 the **RJJ3DSCP** ;)