Shuffle Me:

This challenge was rewarded 200 points at the beginning of the CTF.
There is no description except “find the correct entries”.

First thing first, we run `file` on the binary and find out it is an ELF 64-bit, dynamically linked, stripped binary.

Running it for the first time we see it expects a “secret” arg and then waits for an input.

Ok, time to throw it in Ghidra.

```
nb = atoi((char *)argv[1]);
memset(buffer,0,0x50);
__isoc99_scanf(&70s,buffer);
local_18 = check(buffer,(ulong)nb);
if (local_18 == 0x460000) {
  puts("Yay!");
}
else {
  puts("Nope.");
}
```

We discover that the secret arg must be a number and that our input string may contain a maximum of 70 characters.

Unfortunately, Ghidra has a hard time decompiling the function I named “check” so we will have to use GDB. But we can clearly see that our goal is to return 0x460000 from it.

When calling this function our args are
rsi = our argument number
rdi = our string

Here are the few instructions it is executing:

```
0x401245:    pop    r8
0x401247:    add    r8,0x38
0x40124b:    push   rax
0x40124c:    mov    rax,rsi
0x40124f:    mov    rcx,0xb5
0x401256:    mul    rcx
0x401259:    add    rax,0xd9
0x40125f:    mov    rcx,0x200
0x401266:    div    rcx
0x401269:    mov    rsi,rdx
0x40126c:    mov    rcx,rsi
0x40126f:    shl    rcx,0x4
0x401273:    add    rcx,r8
0x401276:    pop    rax
0x401277:    sub    r11,0x7
0x40127b:    jmp    rcx
```

We see that our number is moved to *rax* and some calculus is done with it.
Here is a more C way of writing the mathematics behind:

```
rsi = (rsi * 0xb5 + 0xd9) % 0x200
rcx = (rsi << 4) + 0x40127d
```

And then we jump on *rcx*.

Once I had a good feeling about what the function was doing, I jumped on *rcx.*
Only one instruction was here, followed by many NOPs and a jump back to our function doing calculus on *rsi*.

So, this function controls the flow of the program with *rsi*. From this function we can jump into addresses between 0x40127d and 0x40327d (0x40127d + (0x200 << 4)).
So I dumped all the instructions in this range with Ghidra.

```
004027cd 4d 01 e3        ADD        R11,R12
004027d0 90              NOP
004027d1 90              NOP
004027d2 90              NOP
004027d3 90              NOP
004027d4 90              NOP
004027d5 90              NOP
004027d6 90              NOP
004027d7 90              NOP
004027d8 e9 6e ea        JMP        LAB_0040124b
         ff ff
004027dd 49 09 d9        OR         R9,RBX
004027e0 90              NOP
004027e1 90              NOP
004027e2 90              NOP
004027e3 90              NOP
004027e4 90              NOP
004027e5 90              NOP
004027e6 90              NOP
004027e7 90              NOP
004027e8 e9 5e ea        JMP        LAB_0040124b
```

An extract of these instructions

I removed NOP instructions with `*cat instructions | grep -v NOP*`

And then I did a lot of checks with lots of CTRL+F. Here is why I verified:

Is there a jump not going back to our function controlling the flow?
> No.
How many RET instructions?
>Only one hopefully.
Is RSI modified in these instructions?
>No, which is great because it means that only the calculus we disassembled earlier are the instructions controlling the flow.
Is there always only one instruction executed before jumping back?
>Yes

Those precious information were positive, but we are still left with 512 *shuffled* instructions (without the jumps!).

I observed these instructions a lot but nothing useful to reduce the possibilities came to me.
I then went back to our calculus:

```
rsi = (rsi * 0xb5 + 0xd9) % 0x200
rcx = (rsi << 4) + 0x40127d
```

I started having a feeling:
There is only one RET instruction that is always the last instruction we execute in the list.
Because of the modulus, *rsi* can take up to 512 (0x200) values, which is exactly the number of instructions we have.
With all the information we have (and a bit of confirmation by a python script I admit), we are sure that these 512 instructions will be executed in the same order every time. We are only controlling where we start in that sequence.

So first, I searched the input that directly leads to the RET function. And then I searched the one that executes all the 512 instructions until the RET. The input we have to give to the program is “474”. But the question is “Do we want those 512 instructions to be all executed?”. I mean, we can start where we want in the chain so nothing proves that these 512 instructions must be executed. Doing a bit of testing, when executing all instructions with a random string, we get a value pretty close to our goal (0x460000):

```
$rax   : 0x467f7f
```

Now, having a feeling that all 512 instructions must be executed in sequence, we can reorder them.
So, I made a file containing one instruction per line, then got the indexes order in an array, and a script to reorder the lines.
Here is a quick look:

```
lines = []
with open('instructions') as my_file:
    for line in my_file:
        lines.append(line)

arr = [474, 507, 336, 105, 278, 359, 172, 1
print(len(arr), len(lines))
test = []
f = open('ordered', 'w')
for i in range(512):
    f.write(lines[arr[i]])
f.close()
```

The array is hardcoded on the screen but was actually calculated with the calculus we dumped earlier.

And then the execution flow became clear.

```
004024bd 8a 1f          MOV     BL,byte ptr [RDI]
0040274d 88 df          MOV     BH,BL
0040171d 66 81 f3       XOR     BX,0x476d
0040252d 48 31 f3       XOR     RBX,RSI
0040167d 49 09 d9       OR      R9,RBX
0040140d 4d 01 d1       ADD     R9,R10
00401add 4d 01 e3       ADD     R11,R12
00402bed 8a 5f 01       MOV     BL,byte ptr [RDI + 0x1]
00401c3d 88 df          MOV     BH,BL
004024cd 66 81 f3       XOR     BX,0x4341
0040129d 48 31 f3       XOR     RBX,RSI
004016ad 49 09 d9       OR      R9,RBX
004015fd 4d 01 d1       ADD     R9,R10
0040198d 4d 01 e3       ADD     R11,R12
00401e5d 8a 5f 02       MOV     BL,byte ptr [RDI + 0x2]
0040256d 88 df          MOV     BH,BL
```

There is a clear block pattern of instructions checking each character of our string (remember it is in *rdi*).

The `MOV BH, BL` transforms for example 0xab into 0xabab.
Then it is XORed to a value, unique for each character.
Finally, it is XORed to *rsi*, our magic number controlling the flow.

We know each flag starts with “FCSC{“, I checked in gdb and it appears that the results of these XORs give 0 for each good character.

Time to solve the challenge.

We have to parse instructions to get every XORed value associated to each char.
I have already calculated the array of *rsi* values in the right order as seen above.

```python
lines = []
with open('ordered') as my_file:
    for line in my_file:
        lines.append(line)

# Values taken by rsi for each instruction, in right order
arr = [474, 507, 336, 105, 278, 359, 172, 117, 402, 275, 328,

for i in range(70):
    xored_value = int(lines[16 + 7 * i].split(',')[1], 16)
    rsi_value = arr[17+7 * i]
    char_value = xored_value ^ rsi_value
    # Undoing MOV BH,BL
    char_value = char_value % 0x100
    print(chr(char_value), end='')
```

And done!

```
FCSC{af22cfdd46bfc8105cb28e04988f904049dd60be1245718ffef5b9bbf9b509d5}
```