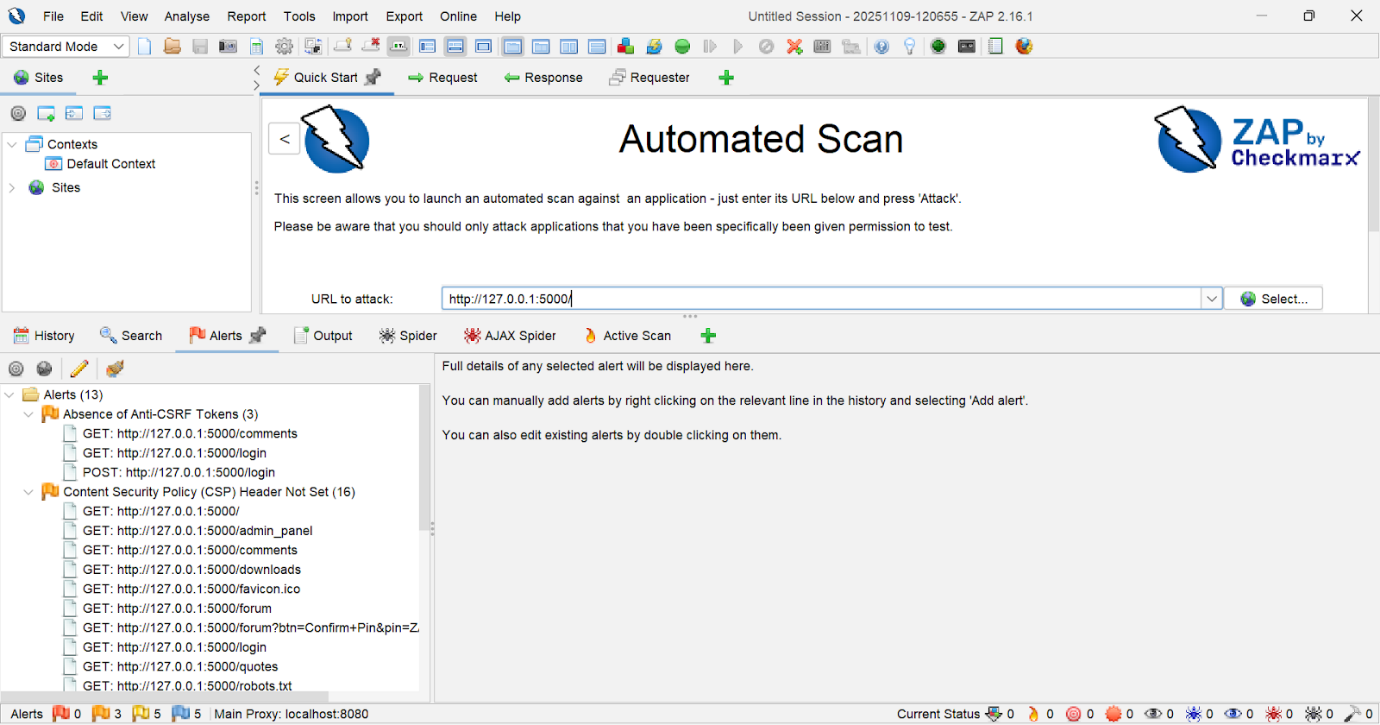
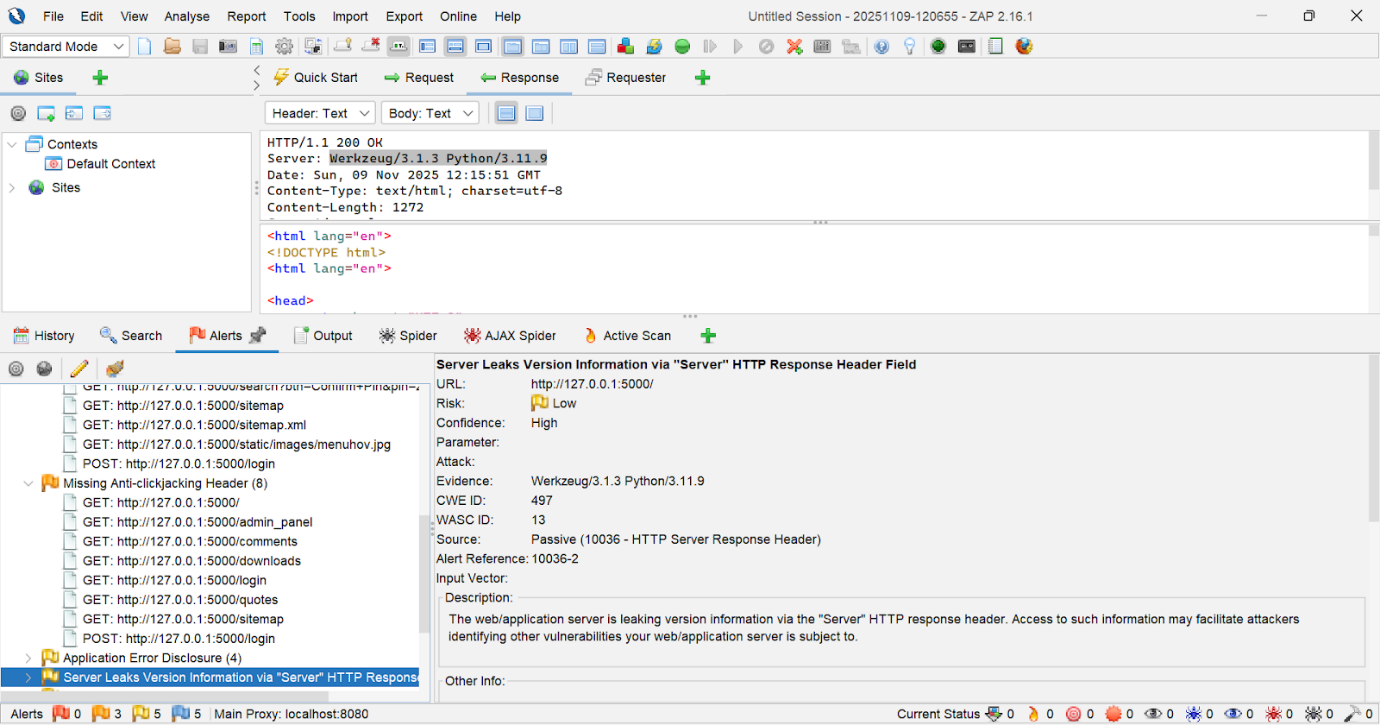
Errors i found looking at [app.py](http://app.py) code

Im using owasp zap to find vulnerabilities.





Known Vulnerability:

1. **Login route — raw SQL with user input, space for sql injection**

//reference: [SQL Injection Prevention - OWASP Cheat Sheet Series](https://cheatsheetseries.owasp.org/cheatsheets/SQL_Injection_Prevention_Cheat_Sheet.html)

Sql injection used : ' OR '1'='1

Faulty code and why:

query = text(f"SELECT \* FROM users WHERE username = '{username}' AND password = '{password}'")

Why this is risky:

* Attackers were able to insert SQL commands into the username or password fields because user input was dropped directly into the SQL command.
* This allows hacker to change the database's functionality like avoiding the password check and logging in under a different identity.
* They can read private information (emails, profiles, and possibly sensitive files) if they can log in or run queries.
* also, they might alter or remove data (erase records or break the app).
* Certain attacks have the potential to escalate and execute database-level commands, granting the attacker command over the server.
* It relied on insecure password handling and exposed more data than necessary due to the use of SELECT \* and plaintext password checks.

What i fixed it into

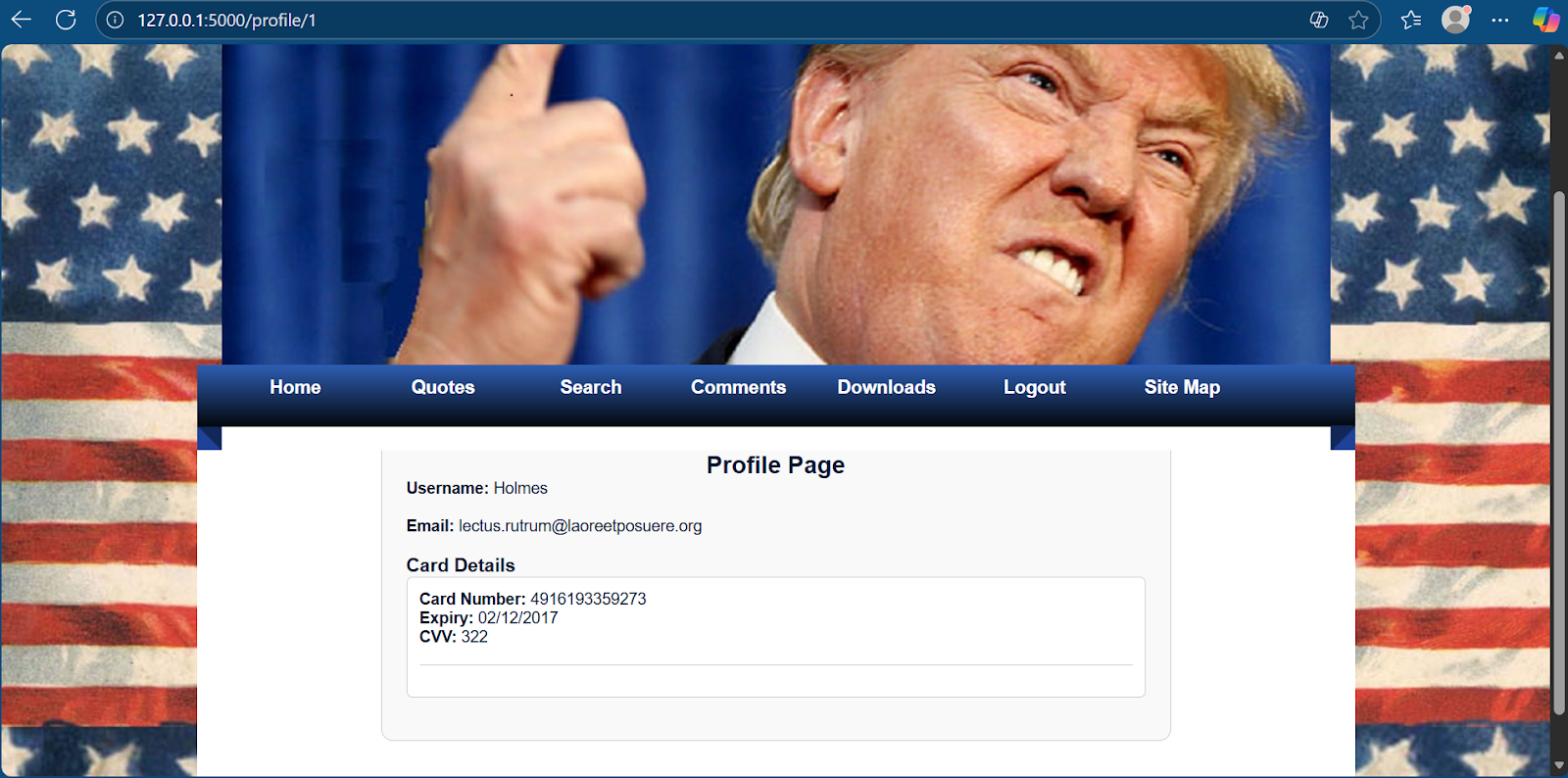
query = text("SELECT \* FROM users WHERE username = :username AND password = :password")

user = db.session.execute(query, {'username': username, 'password': password}).fetchone()

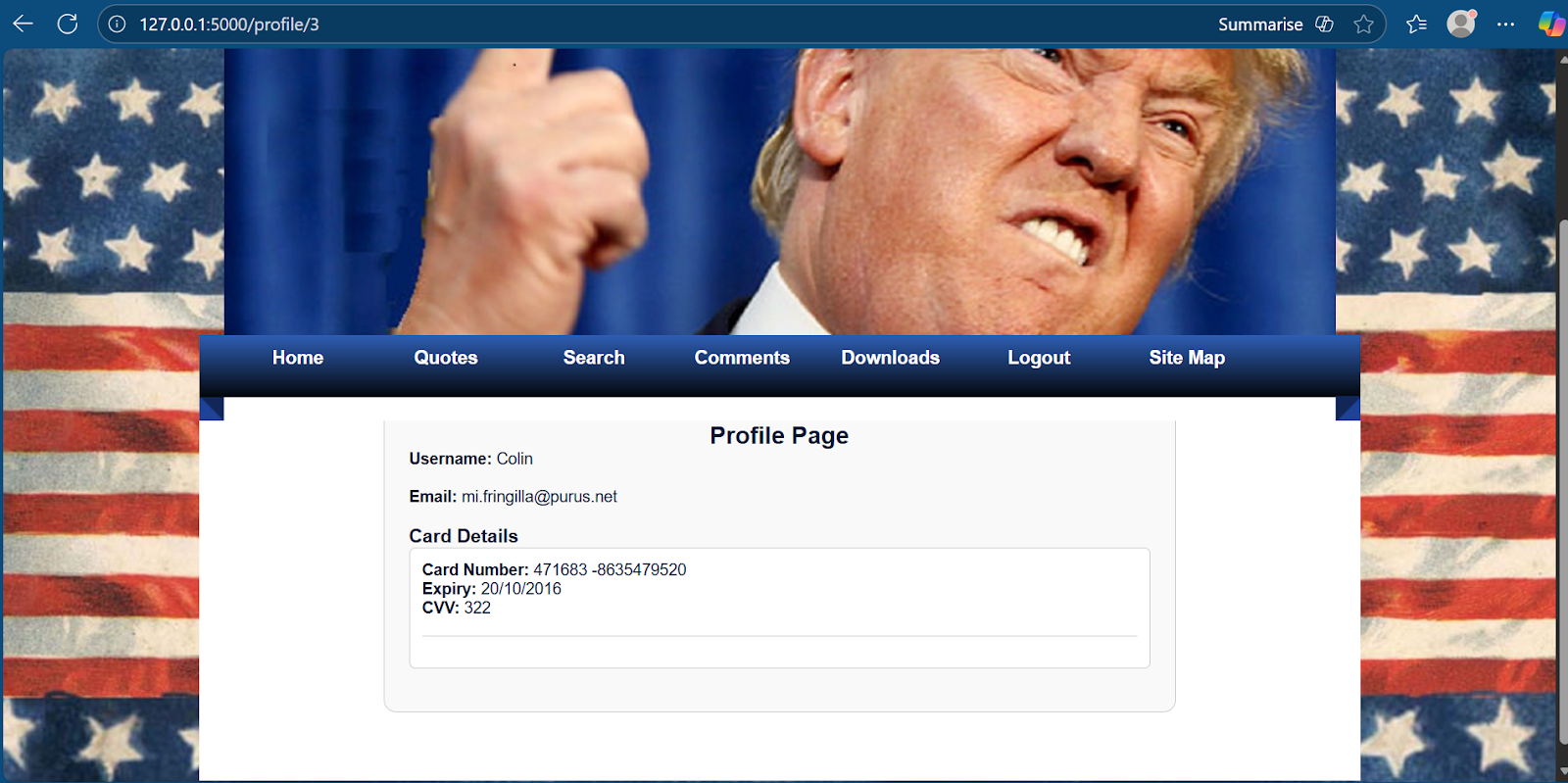
Why its safer:

* The code i added instead uses parameterized queries (placeholders) so the database treats username/password as **data**, not code.
* That prevents user input from changing the SQL structure — classic SQL injection is blocked.
* I’m no longer building the SQL string by hand, which removes the easy injection vector.
* The fixed code is also clearer and less error-prone cos it’s harder to accidentally introduce another injection later.

Result of sql injection done in comments page:



CHANGING THE NUMBER AT THE END OF \ SHOWS U DIFFERENT USERS



Unknown Vulnerability:

1. **Clickjacking -**

tricking a user into clicking something on a hidden or framed page by placing a fake UI over it, so the user thinks they clicked the attacker’s button but actually clicked a button on the victim site.

Example scenario:

* Victim site has a “Delete account” button and you’re logged in.
* Attacker page places a big, shiny “Click me!” button over the framed “Delete account” button.
* usernclick the shiny button but the browser delivered the click into the victim iframe and your account gets deleted!!

First and foremost I made this basic html attacker page with our website embedded into it

Reference :[<iframe>: The Inline Frame element - HTML | MDN](https://developer.mozilla.org/en-US/docs/Web/HTML/Reference/Elements/iframe)

<!doctype html>

<html>

<head>

  <meta charset="utf-8">

  <title>Attacker Demo (Local)</title>

  <style>

    body { font-family: sans-serif; padding: 20px; }

    .box { width: 600px; height: 300px; position: relative; border: 2px solid #ccc; }

    iframe { width: 200%; height: 200%; border: 0; opacity: 0.99; }

    .overlay {

      position: absolute;

      right: -160px; top: 399px;

      width: 150px; height: 44px;

      display:flex; align-items:center; justify-content:center;

      background: rgba(255,255,255,0.6);

      border: 2px dashed red;

      cursor: pointer;

      user-select: none;

      z-index: 2;

    }

  </style>

</head>

<body>

  <h2>Attacker Page :p</h2>

  <p>Click the fake button below! not suspicious at all.</p>

  <div class="box">

    <iframe id="victimFrame" src="http://localhost:5000/"></iframe>

    <div class="overlay" onclick="alert('You clicked the attacker overlay ');">

      Click me hehe

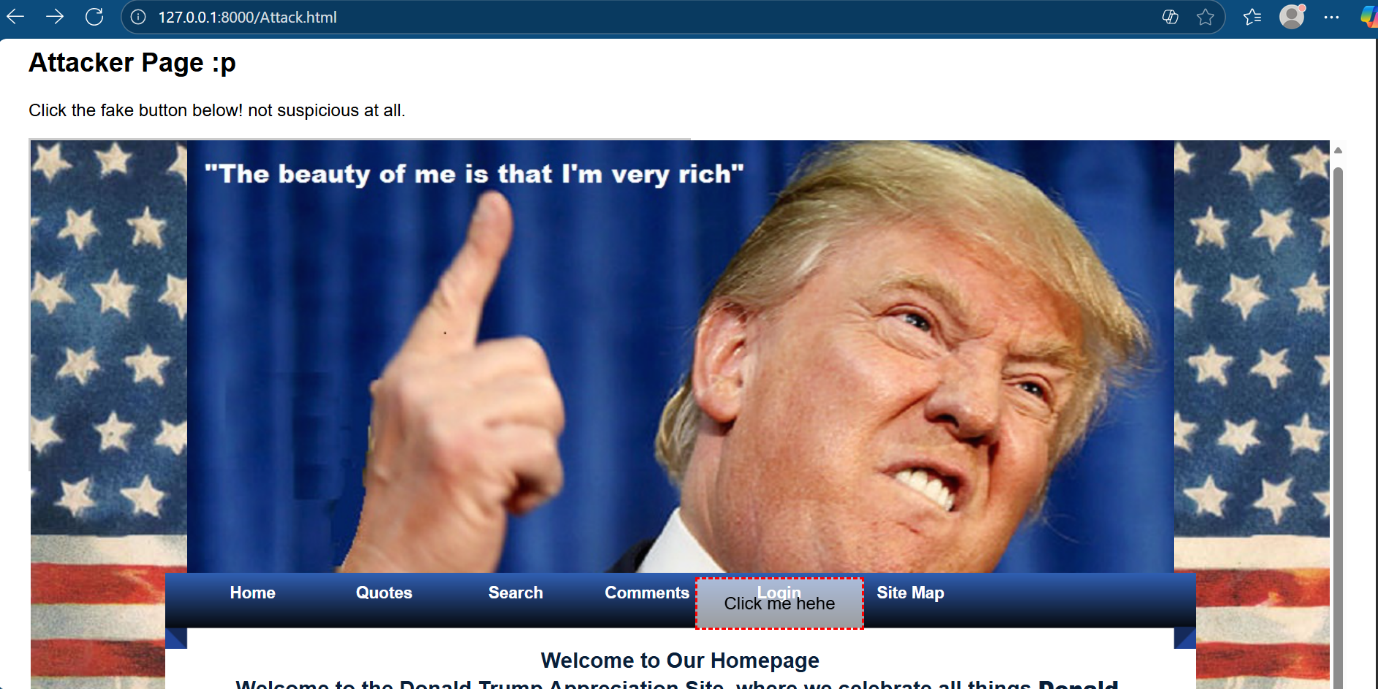
    </div>

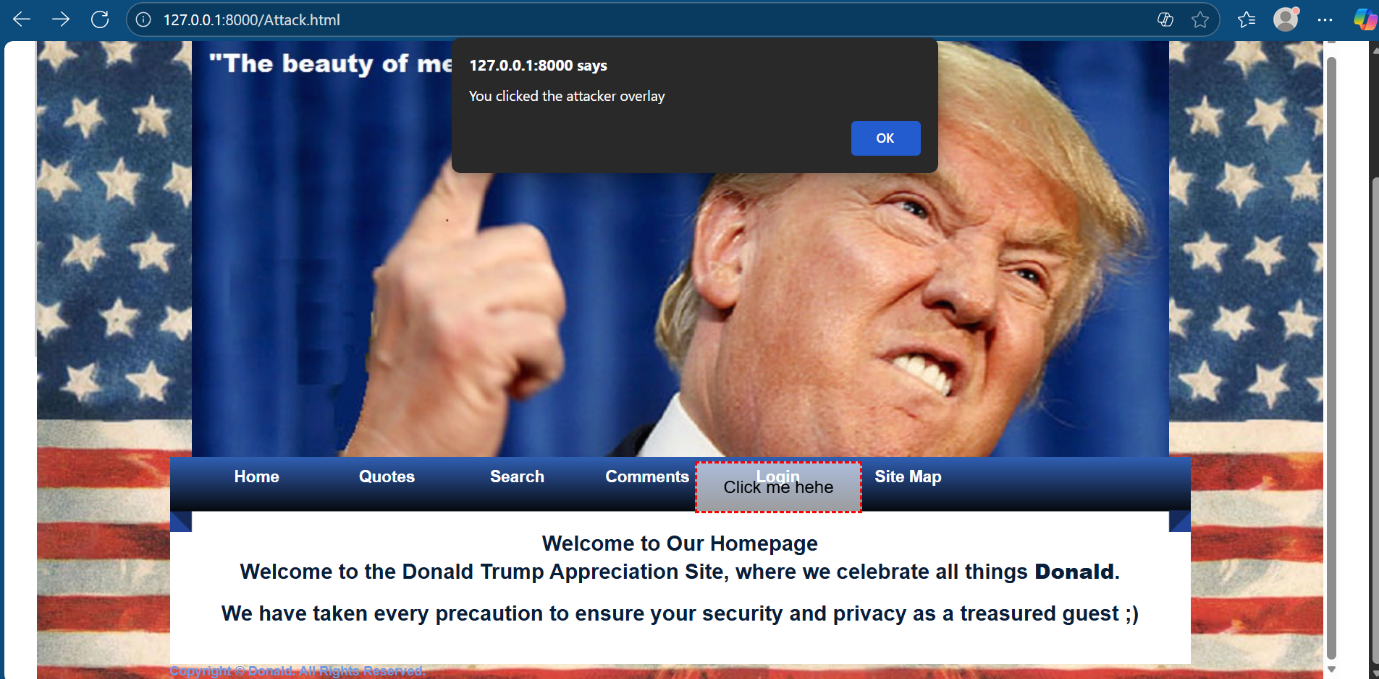
  </div>

</body>

</html>

And upon running it on a different server it looks like this





And so obviously we can see thus website is susceptible to clickjacking attacks.

My fix:

Just above  app = Flask(\_\_name\_\_) I added the code below

References :

[X-Frame-Options header - HTTP | MDN](https://developer.mozilla.org/en-US/docs/Web/HTTP/Reference/Headers/X-Frame-Options)

@app.after\_request

def set\_security\_headers(response):

    # Prevent our pages from being loaded in iframes (stop clickjacking)

    response.headers['X-Frame-Options'] = 'DENY'

    return response

* Prevents the app’s pages from being loaded inside <iframe> or other frame elements.
* This stops clickjacking attacks, where a malicious page tricks a user into clicking hidden elements on our site.

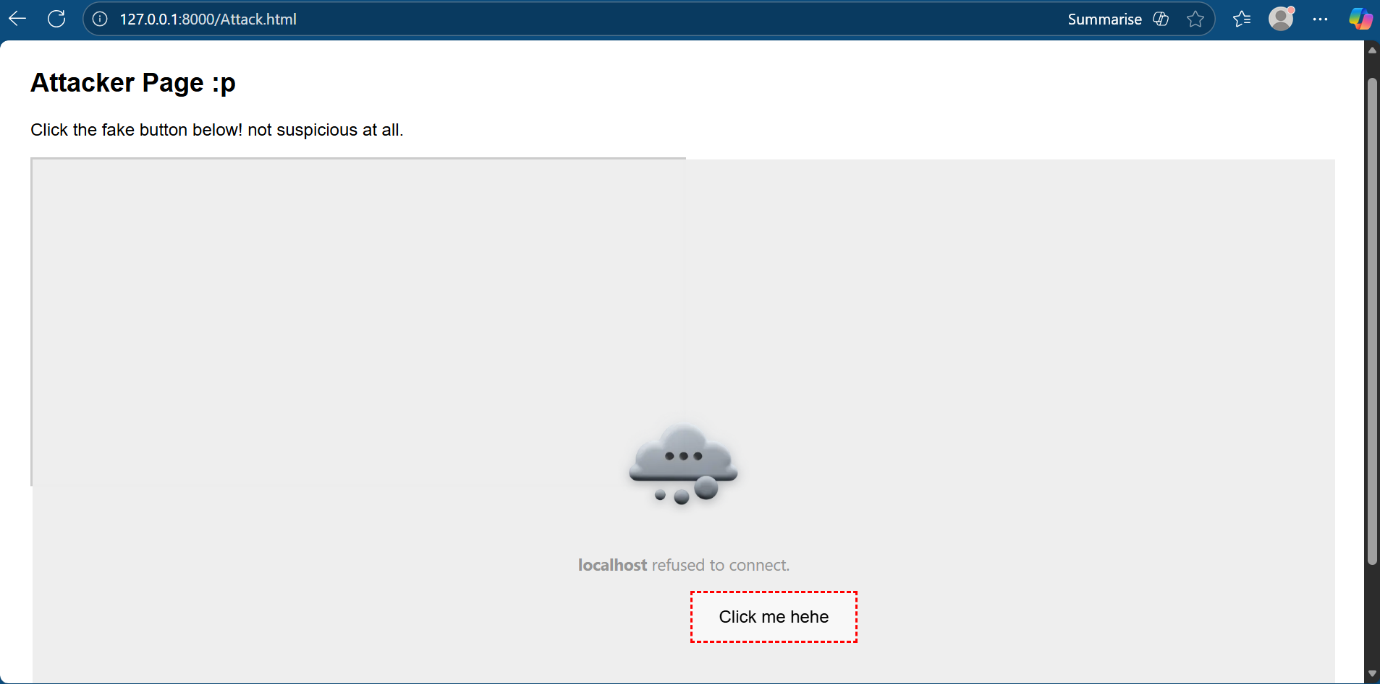
How it works:

* Adds the 'X-Frame-Options: DENY' header to every HTTP response.
* Browsers enforce this by refusing to render the page in any frame.

Why it’s safe:

* Does not execute any user input or external code .
* Applies globally to all routes via @app.after\_request.
* Only modifies headers so no impact on legitimate user interactions.

After the fix:



1. **Debug Mode Exposure(RCE) –**

What it is: The website was running in Flask’s debug mode. This turns on a built-in web console that lets anyone run Python code on the server. In other words debug mode accidentally gives visitors a web-based Python terminal. They can use it to read files, run commands, and take over the server.

Where the flawed code is:

Its right at the bottom

if \_\_name\_\_ == '\_\_main\_\_':

    initialize\_database()  # Initialize the database on application startup if it doesn't exist

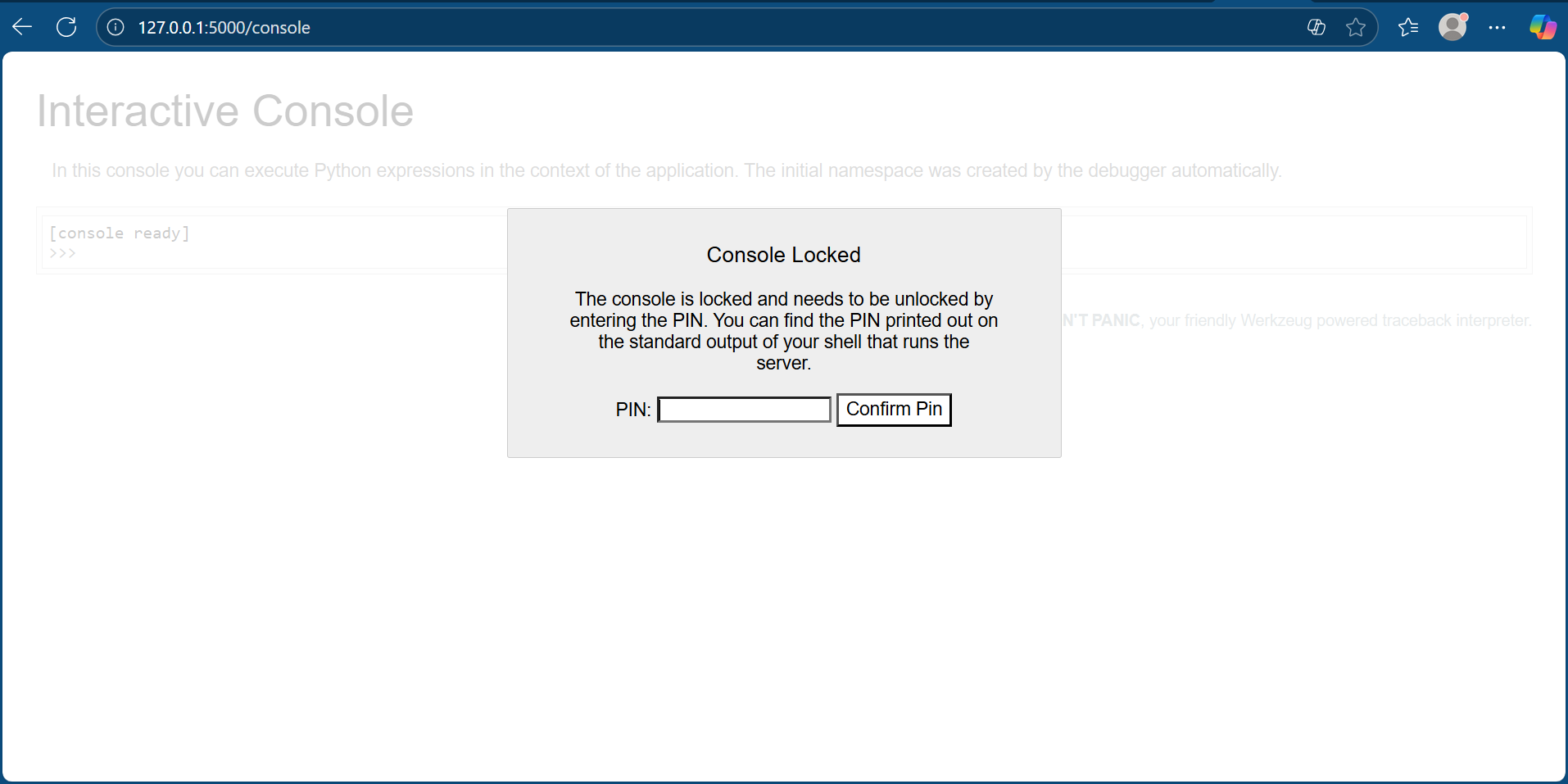
    with app.app\_context():

        db.create\_all()  # Create tables based on models if they don't already exist

    app.run(debug=True)

What I did to expose it:

Just after my url, I type console to pull up the console. Although it requires a pin, a console should never show up on a deployed app. Anyone brute force a pin and further have access to sensitive data like the db.



To fix this, I simply turn the debug setting to false instead of true.

app.run(debug=False)

After doing this, if I try go in and access console again, the page returns a 404.

