King Saud University
College of Computer and Information Sciences
Computer Science Department

CSC429: Computer Security
1st Semester 1447
Project

Building a Secure Web Application -Detection and Mitigation of Security Vulnerabilities

Prepared by:

Student name	ID
Kholod Almutairi	443200467
Leen Alobaidi	443201010
Razan Bagazi	441200280
Ragad Alrowili	443200747

Instructor:
Dr. Arwa Alawajy

Introduction:

This report presents the main security vulnerabilities in our application and the steps we took to fix them. We focused on issues such as SQL injection, Cross-Site Scripting (XSS), access control weaknesses, password protection, and secure communication. For each vulnerability, we explain how it appeared in our code, how it could be exploited, and the changes we implemented to improve the overall security of the application.

1. SQL Injection:

1.1 Vulnerable Version :

The original login used a raw SQL query constructed through string concatenation
An attacker can inser malicious SQL code in the username field, for example
' OR 1=1 --
Since `1=1` is always true, the query returns the first user in the database.

```
sql = text(f"SELECT * FROM user WHERE username = '{username}' AND password = '{password}' ")
```

1.2 Secure Version :

We fixed this vulnerability by replacing raw string-based SQL queries with SQL Alchemy's.
This ensures that user input is safely handled as data rather than executable SQL, whitch will prevent attackers from injecting malicious commands such as ' OR 1=1 --

```
user = User.query.filter_by(username=username).first()
```

2. Weak Password Storage

2.1 Vulnerable Version

In the vulnerable implementation, user passwords were stored in plaintext directly in the database. This exposes all user credentials if the database is compromised. Attackers can read, reuse, or crack the passwords easily.

Security Risks

- Immediate credential exposure
- Enables credential-stuffing across other services

Vulnerable Code:

```
# Vulnerable version
        #sql = text(f"INSERT INTO user (username, password) VALUES ('{username}','{password}')")
    #db.session.execute(sql)
    #db.session.commit()
```

2.2 Secure Version

The secure implementation uses **bcrypt** to hash passwords before storing them, because it is widely trusted and slows down brute-force attacks.

Secure Code:

```
# Secure version
    hashed_password = bcrypt.hashpw(password.encode(), bcrypt.gensalt()).decode('utf-8')
    new_user = User(username=username, password=hashed_password)
    db.session.add(new_user)
    db.session.commit()
```

Why This is Secure

- Passwords are never stored in recoverable form
- Salt is automatically included
- Bcrypt resists brute-force attacks

3. Cross-Site Scripting (XSS)

3.1 Vulnerable Version

User comments were accepted and displayed without sanitization. Since the comment was rendered using |safe, any JavaScript injected by a user would execute in the browser.

Vulnerable Code:

```
#Vulnerable version
    #comments.append(content)
```

Security Risks

Attackers can inject scripts to:

- Steal session cookies
- Perform unauthorized actions

3.2 Secure Version

Sanitize comments to mitigate XSS vulnerabilities.

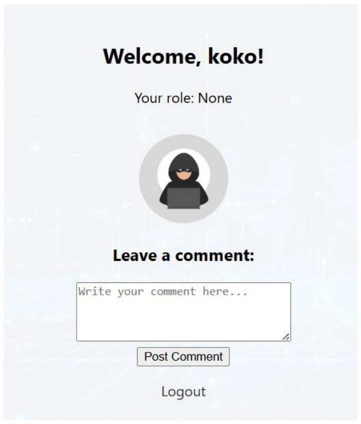
Secure Code:

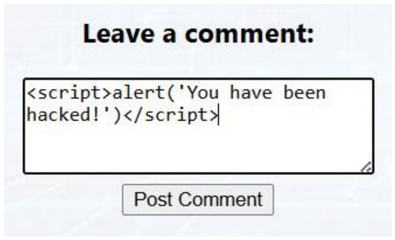
```
#Secure version
```

```
    #Used to sanitize comments to mitigate XSS vulnerabilities.
    sanitized = html.escape(content)
    comments.append(sanitized)
```

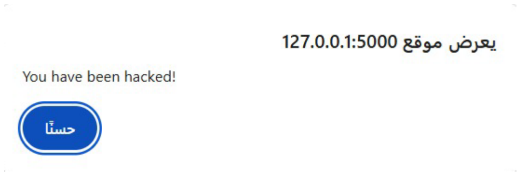
Why This is Secure

- Converts <script> tags into harmless text
- Prevents execution of injected JavaScript





```
<script>alert('You have been hacked!')</script>
```



You have been hacked!

4. Access Control

4.1 Vulnerable Version

The original route allowed anyone to access the admin panel without checking authentication or roles.

Vulnerable Code:

```
# Vulnerable version
    #return render_template('admin.html', username=session.get('username'), role=session.get('role'), users=[])
```

Security Risks

- Unauthorized access to user data
- Full exposure of the system's administrative interface

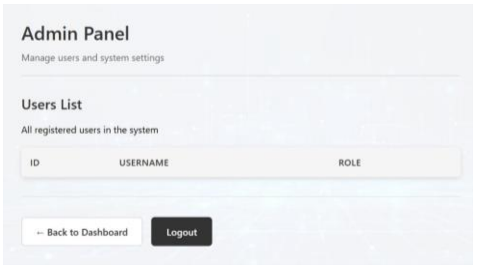
4.2 Secure Version

Secure Code:

```
# Secure version
   if 'username' not in session:
      return redirect(url_for('login'))

   if session.get('role') != 'admin':
    return '❌ Access denied. You are not an admin.', 403
```

Why This is Secure
- Enforces role-based authorization
- Restricts admin-only actions


Admin Panel
Manage users and system settings

Users List
All registered users in the system

ID	USERNAME	ROLE

5. Encryption :

To secure communication in our application, we configured Flask to run over HTTPS using the SSL/TLS encryption protocol. This was implemented in our code by providing Flask with a self-signed certificate and private key .

```
    app.run(debug=True, ssl_context=('cert.pem', 'key.pem'))
```

Challenges faced:

A significant challenge our group faced was the lack of a graphical user interface (GUI) for the database, which made it difficult to visually verify our security patches. Since the application relies on a file-based SQLite database, we could not simply open a dashboard to inspect stored data like user credentials. We overcame this by learning how to navigate the database directly through the terminal using sqlite3 commands.

Acknowledgement:

We acknowledge the use of OpenAI's ChatGPT as a supportive tool during the development of this project, particularly for assisting us in generating and refining segments of the code. All final design decisions, implementations, and evaluations were carried out and verified by the group.