Flask API Security Simulation Report

BATCH-12
21I204 - Afrith Ahamed AN
21I212 - Dharaneesh C.k.v
21I213 - ESAKKI AKASH.E
21I234 - Pragadesh Arun T
21I222 - jayaasurya R.D

Overview

This Flask-based application simulates various security vulnerabilities such as SQL Injection and Cross-Site Scripting (XSS), while demonstrating protection techniques like input sanitization and rate limiting. The goal is to showcase common attack vectors in web applications and how to log and mitigate them. This report details the code functionality, potential security vulnerabilities, attack logging, and prevention mechanisms.

Table of Contents

1. Application Overview

The web application is built using Flask, a lightweight Python web framework. It simulates two common attack vectors: **SQL Injection** and **Cross-Site Scripting (XSS)**. Additionally, it implements **rate limiting** to mitigate Denial of Service (DoS) attacks. The system logs attempted attacks and provides an interface to view and clear these logs.

2. Database Configuration

The database uses **SQLite** to store user credentials and attack logs. The database URI is configured as follows:

```
app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///api_security.db'
app.config['SQLALCHEMY_TRACK_MODIFICATIONS'] = False
```

- **api_security.db** is the SQLite database file.
- **SQLALCHEMY_TRACK_MODIFICATIONS** is set to False to disable event notifications.

3. User Model

The User model stores basic user information such as username, email, and password. It simulates a basic user login system:

python
Copy code

```
class User(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    username = db.Column(db.String(100), nullable=False)
    email = db.Column(db.String(100), nullable=False)
    password = db.Column(db.String(100), nullable=False)
```

Fields:

- id: Primary key for the table.
- username: The user's login name.
- email: The user's email address.
- password: The user's password.

4. Log Model

The Log model keeps track of any attempted attacks and records them in the database:

python
Copy code

```
class Log(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    attack_type = db.Column(db.String(100), nullable=False)
    timestamp = db.Column(db.DateTime, default=db.func.now())
```

Fields:

- id: Primary key for the log.
- attack_type: The type of attack detected (e.g., SQL Injection, XSS).
- timestamp: When the attack was detected.

5. Attack Logging

The log_attack function is a helper method that logs any detected attacks into the Log table:

```
def log_attack(attack_type):

    log = Log(attack_type=attack_type)
    db.session.add(log)
    db.session.commit()
```

Whenever a potential attack is detected (such as an SQL injection or XSS attempt), this function is called to log the attack.

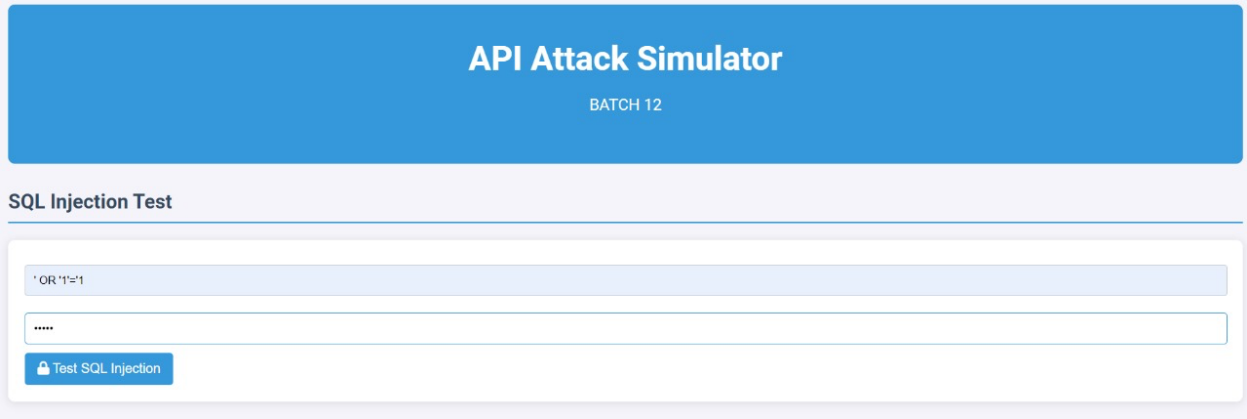
6. SQL Injection Vulnerability Simulation

The /vulnerable_sql route simulates an SQL Injection vulnerability. It accepts POST requests with username and password inputs. If an attack attempt is detected, it logs it in the Log table and returns a response indicating the presence of an SQL injection attempt:

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

    # Simulate SQL Injection
    if username == "' OR '1'='1":
        log_attack("SQL Injection Attempt")
        return jsonify({"message": "SQL Injection detected!"}), 400
    elif username == "admin" and password == "password":
        return jsonify({"message": "Login successful!"})
    return jsonify({"message": "Login failed!"}), 401
```

If the username input contains an SQL injection string (' OR '1'='1), the system detects it, logs the attack, and returns a **400 Bad Request** error.

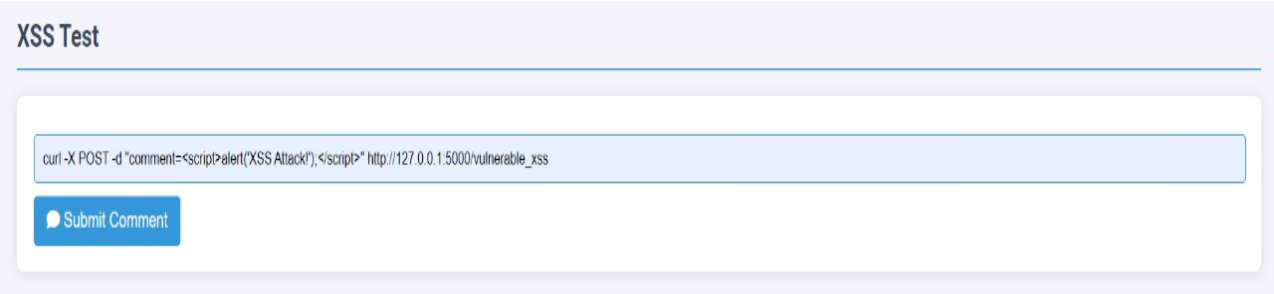


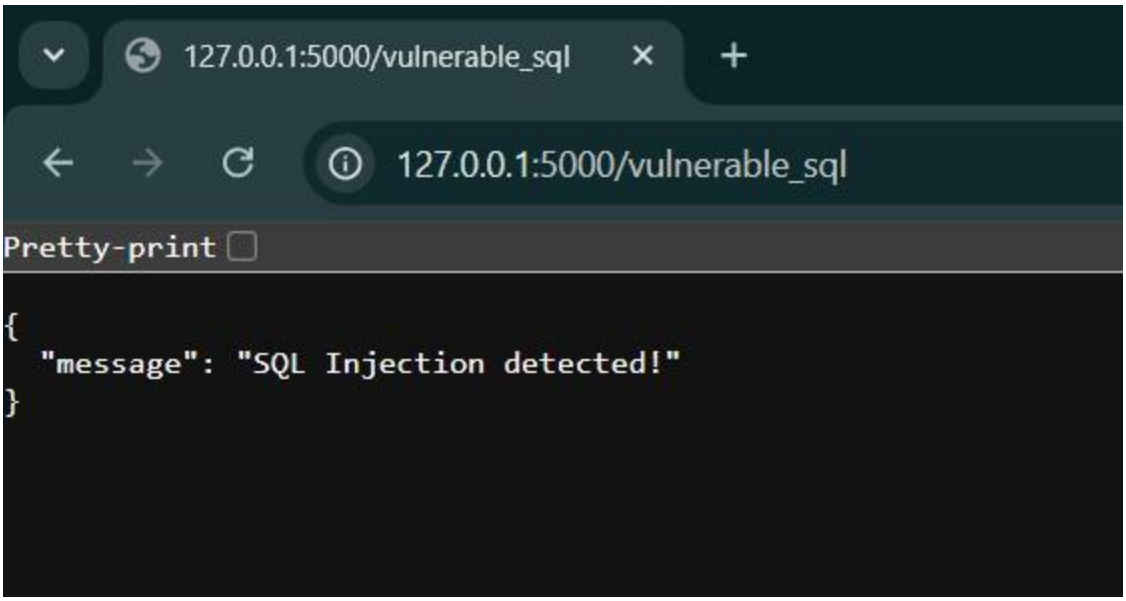
7. XSS Vulnerability Simulation

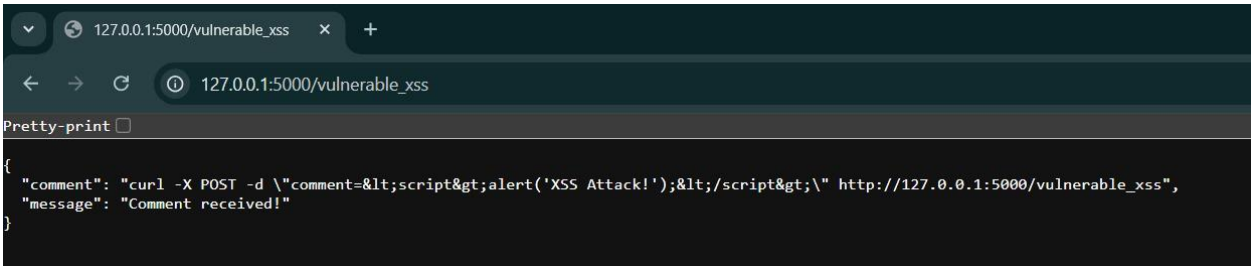
The /vulnerable_xss route simulates a Cross-Site Scripting (XSS) vulnerability. It processes a user-submitted comment, cleans it with the **Bleach** library to remove any malicious input, and logs any XSS attempts:

```
@app.route('/vulnerable_xss', methods=['POST'])
def vulnerable_xss():
    comment = request.form['comment']

    # Clean input to prevent XSS
    clean_comment = bleach.clean(comment)
    log_attack("XSS Attempt")
    return jsonify({"message": "Comment received!", "comment": clean_comment})
```





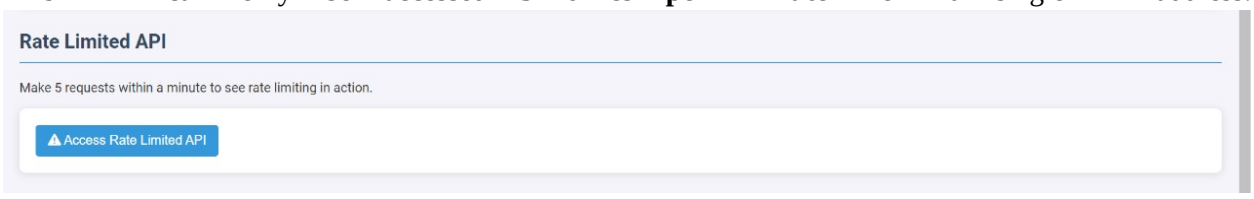


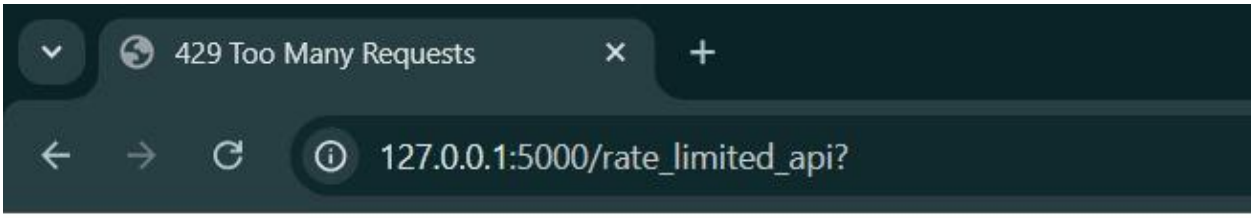
8. Rate Limiting for Denial of Service (DoS) Protection

The /rate_limited_api route uses the **Flask-Limiter** library to enforce rate limits. This protects against DoS attacks by limiting how often a client can access the endpoint:

```
@app.route('/rate_limited_api', methods=['GET'])
@limiter.limit("5 per minute")
def rate_limited_api():
    return jsonify({"message": "This is a rate-limited API."})
```

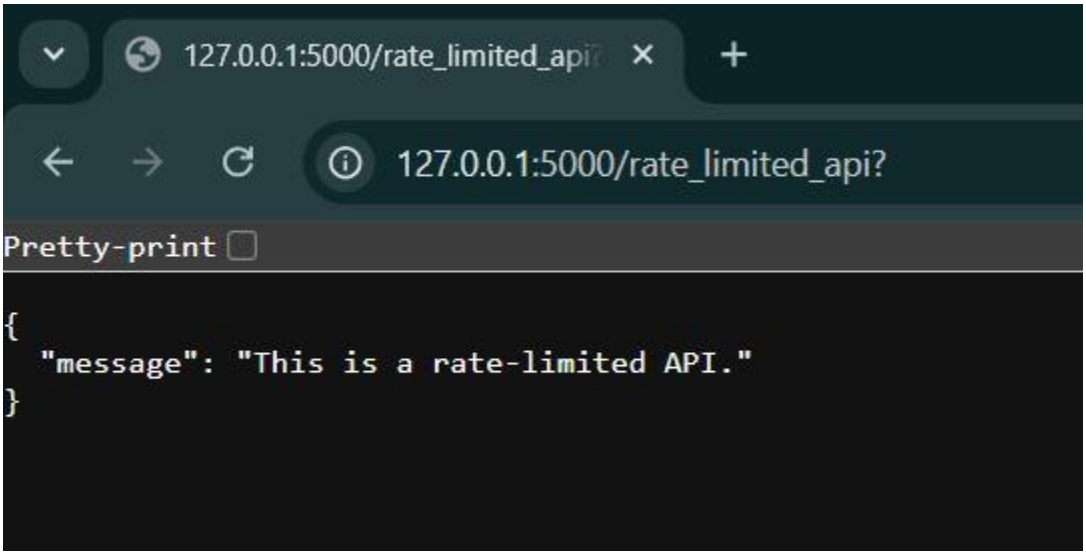
This API can only be accessed **5 times per minute** from a single IP address.





Too Many Requests

5 per 1 minute




```
PS C:\Users\Dell> curl http://127.0.0.1:5000/rate_limited_api
curl : Too Many Requests
5 per 1 minute
At line:1 char:1
+ curl http://127.0.0.1:5000/rate_limited_api
+ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    + CategoryInfo          : InvalidOperation: (System.Net.HttpWebRequest:HttpWebRequest) [Invoke-WebRequest],
   WebException
    + FullyQualifiedErrorId : WebCmdletWebResponseException,Microsoft.PowerShell.Commands.InvokeWebRequestComman
   d
```

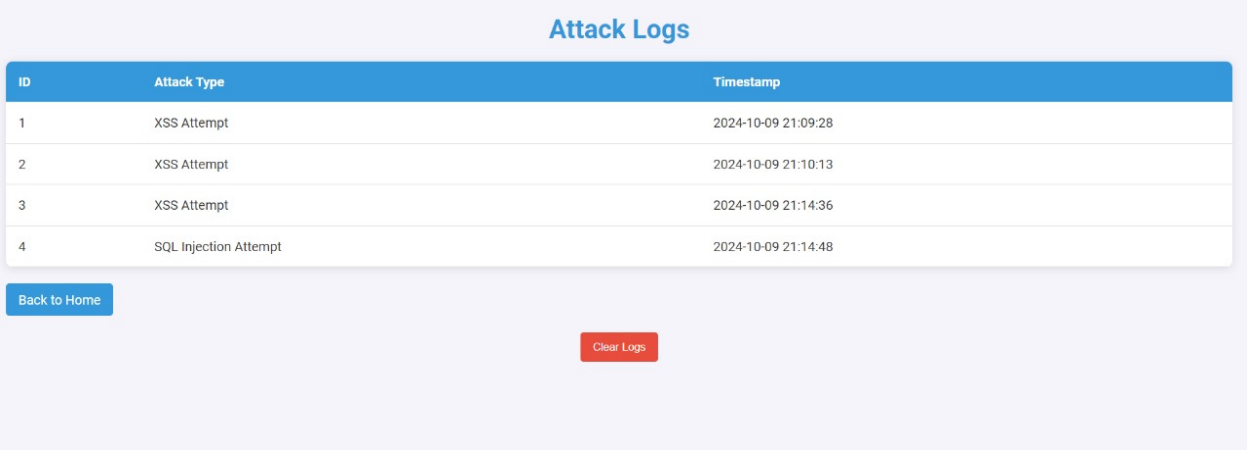
9. Viewing and Clearing Logs

Viewing Logs

The /logs route allows viewing of all attack logs:

@app.route('/logs', methods=['GET'])

```
def view_logs():
    logs = Log.query.all()
    return render_template('logs.html', logs=logs)
```



Clearing Logs

The /clear_logs route allows clearing all attack logs:

```
@app.route('/clear_logs', methods=['POST'])
def clear_logs():
    db.session.query(Log).delete()
    db.session.commit()
    return redirect(url_for('view_logs'))
```

10. Bleach for Input Sanitization

The **Bleach** library is used to sanitize user inputs in the XSS vulnerability simulation. It strips out any potentially dangerous tags and attributes from user-submitted content.

11. Flask-Limiter for Rate Limiting

The **Flask-Limiter** library is integrated to enforce rate limits and prevent abuse of certain routes. It uses **in-memory storage** by default for managing limits.

12. Detailed Code Explanation

The code simulates a real-world scenario of handling SQL Injection, XSS, and DoS attacks. It integrates the following libraries:

- **Flask**: Web framework.
- **SQLAlchemy**: ORM for database interaction.

- **Bleach**: Input sanitization for XSS prevention.
- **Flask-Limiter**: Rate limiting for DoS protection.

13. Future Improvements

- **User Authentication**: Implement a proper authentication system using hashed passwords and sessions.
- **Advanced Logging**: Add more details to logs, such as IP addresses and request headers.
- **Testing**: Add unit tests to ensure the application's robustness.

14. Conclusion

This application demonstrates how to identify and mitigate common web application vulnerabilities such as SQL Injection, XSS, and DoS attacks. The logging system enables tracking of malicious activities, and the rate limiting mechanism offers protection against traffic-based attacks.