SQL INJECTION PLAYGROUND WITH DETECTION ENGINE

Author:

Kummithi Prabhat Obula Reddy

Project:

SQL INJECTON PLAYGROUND WITH DETECTION ENGINE

1. Abstract

This project builds an educational SQL Injection (SQLi) playground consisting of (1) a deliberately vulnerable Flask web application and (2) a Python-based detection engine that probes the app for SQLi vulnerabilities. The playground demonstrates common SQLi scenarios (search and login endpoints), the differences between vulnerable and parameterized (safe) implementations, and basic detection techniques (error-based and boolean-difference).

2. Objectives

- Create a small, self-contained vulnerable web app to demonstrate SQL injection risks.
- Build a lightweight detector that automatically tests endpoints with common SQLi payloads.
- Log and present scan results so learners can observe how injections behave.
- Provide a secure implementation using parameterized queries to demonstrate mitigation.

3. Architecture & Components

- **Flask web app (vuln_app)**
  - app.py — vulnerable endpoints:
    - /search?q= — builds SQL via string formatting (vulnerable).
    - /login — authentication query built via unsafe string interpolation (vulnerable).
  - secure_app.py — same endpoints but using parameterized queries.
  - templates/ — simple Jinja2 HTML pages for interaction.
  - schema.sql → used to initialize app.db (SQLite).
- **Detector (detector)**
  - detector.py — sends HTTP requests to target endpoints with a list of SQLi payloads, records baseline responses, detects anomalies (error messages and response-length differences), and logs results to results.csv and results.db (SQLite).

- payloads.txt — optional custom payloads file.

- **Storage / Output**
  - vuln_app/app.db — seeded sample DB.
  - detector/results.csv, detector/results.db — scan outputs and logs.

4. Implementation Details

- Language & libraries: Python 3, Flask, SQLite (via Python sqlite3), requests.
- Vulnerable code example (simplified):

```
sql = "SELECT id FROM users WHERE username='{}' AND
password='{}';".format(user, pwd)
```

This is intentionally insecure because user input is directly concatenated into SQL.

- Secure example uses parameterized queries:

```
cur = db.execute("SELECT id FROM users WHERE username=? AND
password=?", (user, pwd))
```

Parameterized queries prevent SQL parsing/concatenation vulnerabilities.

5. Detector Logic

1. **Baseline** — request the endpoint with a benign query (e.g., normalquery) and record response text and length.
2. **Payload testing** — iterate through payloads such as ' OR '1'='1, UNION SELECT ..., and others supplied in payloads.txt or built-in defaults.
3. **Detection heuristics**
   a. **Error-based**: search response text for DB error signatures (e.g., syntax error, sqlite3.OperationalError).
   b. **Boolean-diff**: compare response length to baseline; a significant difference (heuristic threshold) flags potential SQLi.

4. **Logging** — write results with payload, HTTP status, response length, detected flag, and reasons into CSV and SQLite DB for audit.

6. How to Reproduce (summary)

1. Create a Python virtual environment and install dependencies:

```
python -m venv venv
source venv/bin/activate    # or .\venv\Scripts\Activate.ps1 on Windows
pip install flask requests
```

2. Create DB:

```
import sqlite3
conn = sqlite3.connect('vuln_app/app.db')
with open('vuln_app/schema.sql') as f: conn.executescript(f.read())
conn.commit(); conn.close()
```

3. Run vulnerable app:

```
cd vuln_app
flask run --port 5000   # or python app.py
```

4. In a separate terminal, run the detector:

```
cd detector
python detector.py --target "http://ipaddress/search?q="
```

5. Review detector/results.csv and detector/results.db.

7. Sample Findings (example)

Sample detector output (illustrative)
Baseline length: 1024
Testing payload: ' OR '1'='1 → detected: True [length-diff]
Testing payload: ' UNION SELECT 1, sqlite_version(), 3 -- → detected: True [error-string: sqlite3.OperationalError]

The vulnerable /search endpoint responded differently to injection payloads and returned database error strings in some cases; the secure /secure_search did not exhibit these behaviors.

Note: The above is a sample. Actual output will vary per environment and payload set.

8. Mitigations & Recommendations

- Always use parameterized queries (prepared statements) for all DB access.
- Use least-privilege database accounts (avoid superuser access for the web application).
- Apply input validation where appropriate and use allow-lists for structured inputs.
- Integrate automated security scans into CI/CD pipelines, and run periodic dynamic scans (in controlled environments).
- Use Web Application Firewalls (WAFs) as an additional layer, not a substitute for secure code.
- Log and monitor suspicious requests and apply rate limiting to mitigate automated attacks.

9. Limitations & Future Work

- The detector uses simple heuristics (error and length diff). It does not implement time-based blind SQLi detection (which requires DB functions like SLEEP() and may need MySQL/Postgres).
- Future improvements:

- Add time-based / blind injection techniques and response fingerprinting.
- Add a web UI to run scans from the browser and visualize results.
- Containerize application stack with Docker and provide a docker-compose.yml.
- Expand payload library and add concurrency/throttling controls.

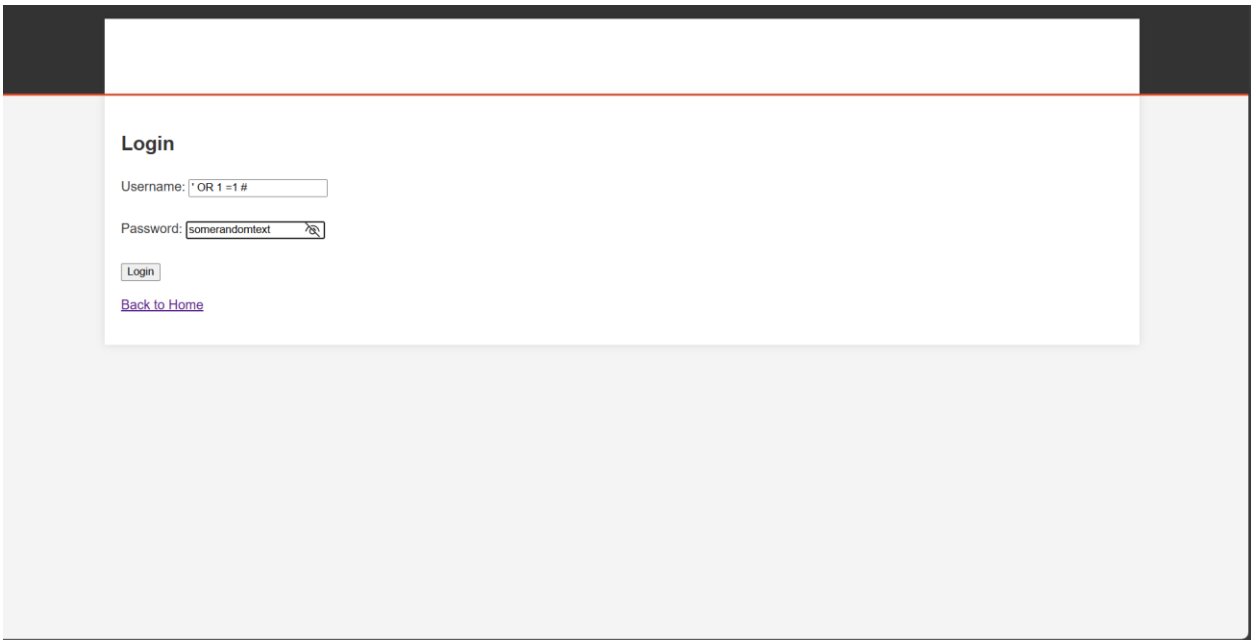
VULNERABLE LOGIN

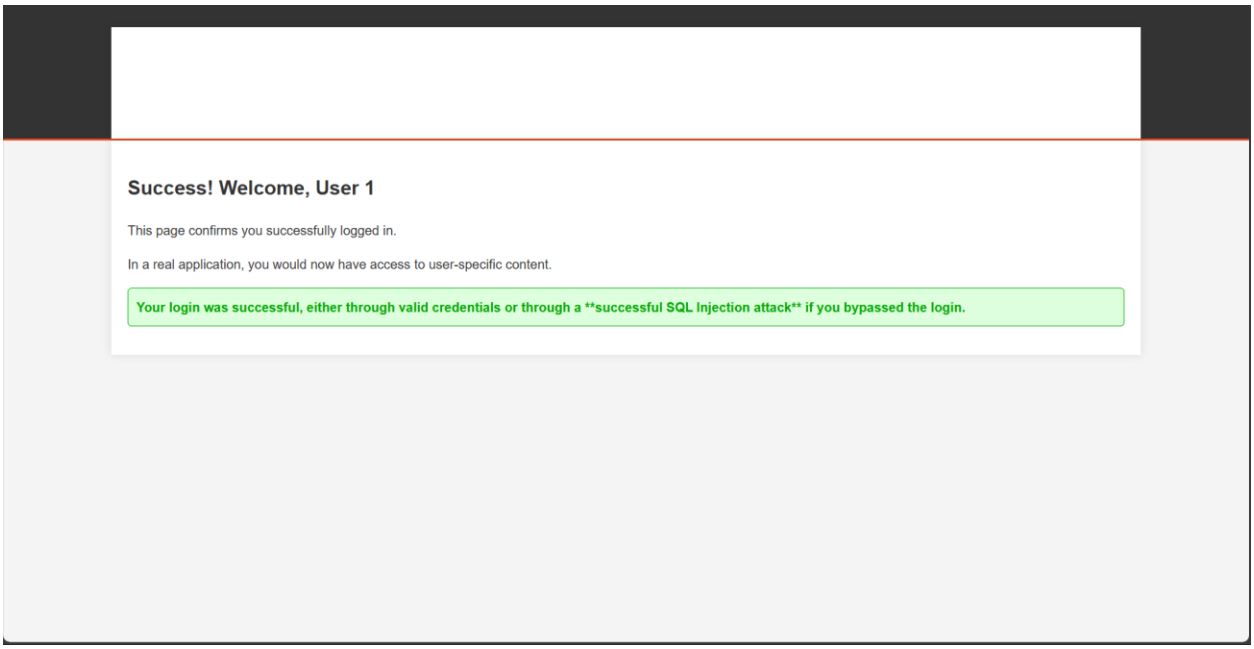
Welcome to the SQL Injection Demonstration Lab!

This application uses a MySQL backend.

The Goal

Demonstrate how malicious user input can change the meaning of a query (Vulnerable App) and how Parameterized Queries prevent this (Secure App).




Success! Welcome, User 1
This page confirms you successfully logged in.
In a real application, you would now have access to user-specific content.
Your login was successful, either through valid credentials or through a **successful SQL Injection attack** if you bypassed the login.