Vulnerability Assessment Report

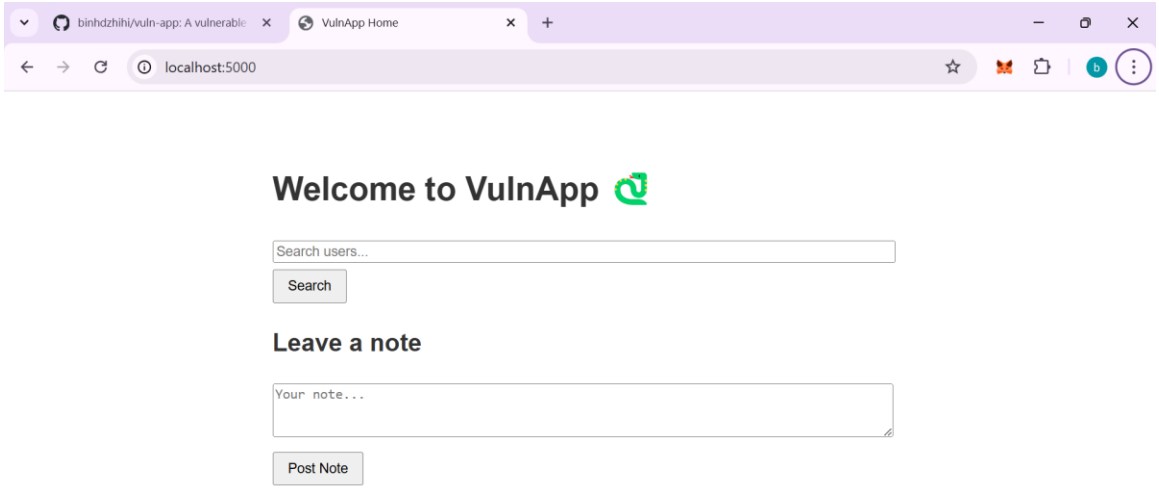
Le Nguyen Gia Binh - V202200678

Project: https://github.com/binhdzhihi/vuln-app

Stack: Flask + MySQL (Dockerized)

1. Web Application Features

- Home page (/) with user search and comment forms
- /search endpoint displays users matching the query
- /comment endpoint stores and reflects notes



---

2. Identified Vulnerabilities

- **SQL Injection in app.py search():** query = f"SELECT ... WHERE username LIKE '%{name}%'"
- **Reflected XSS in app.py comment():** return f" Your note: {note} "

3. Black-Box Testing Methodology

- 1. Reconnaissance: Browse to `/`, inspect forms and view requests in DevTools.
- 2. Tools & Techniques: Use Burp Suite, sqlmap, and manual payload injection.
- 3. Indicators: SQL errors or full-table results; raw <script> tags in responses.

Observations:

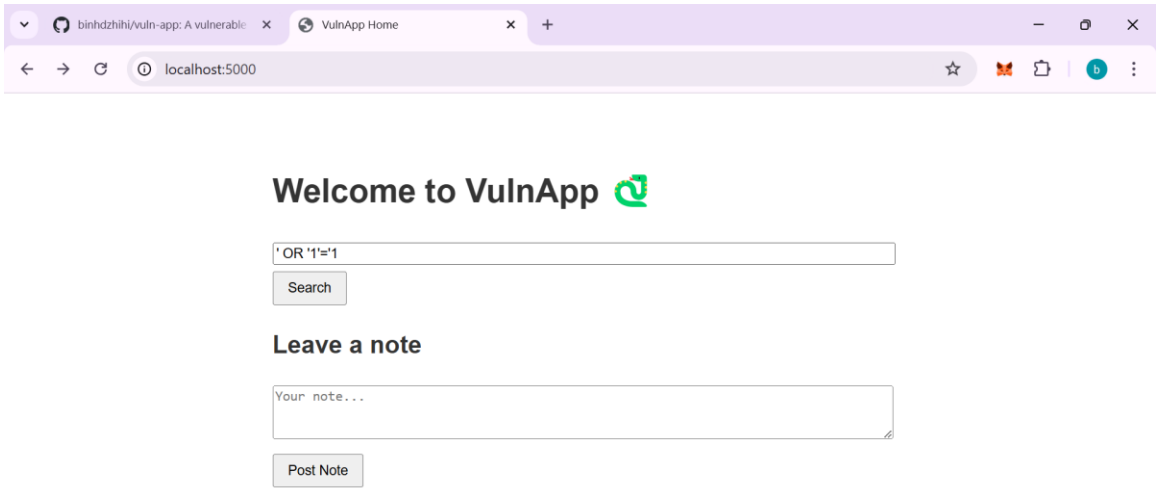
When testing the live app without source code, I observed:

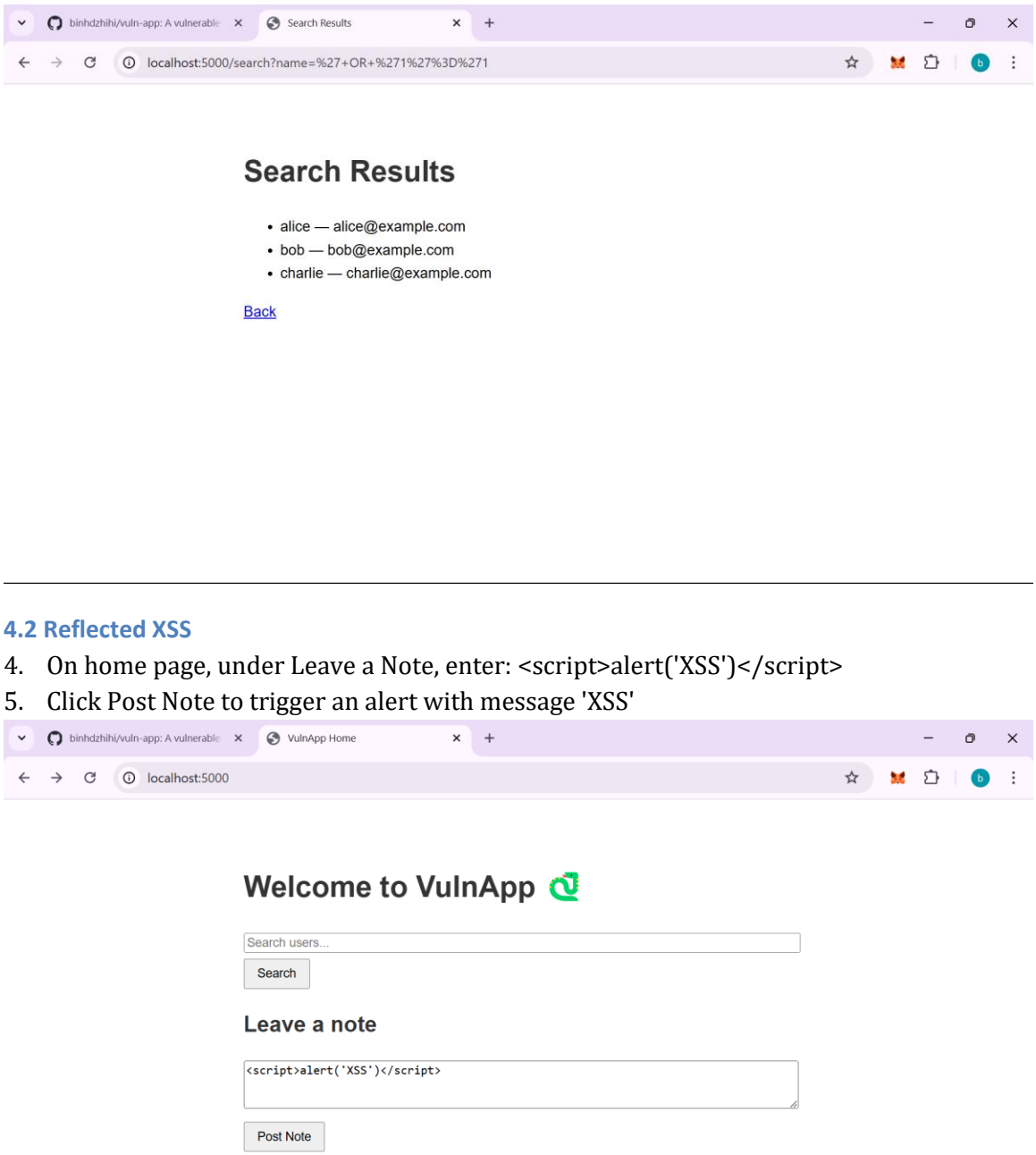
- All Records Returned: Submitting `**' OR '1'='1**` in the search form caused every user (alice, bob, charlie) to appear — proof of SQL Injection.
- Syntax Error Revealed: Entering a single quote (`'`) in the search box produced a Flask debug error page showing the raw SQL statement and traceback, exposing internal details.
- Immediate XSS Execution: Posting `**<script>alert('XSS')</script>**` in the comment form triggered an alert popup, confirming the input was reflected without HTML-escaping.
- Payload in HTML Source: Viewing page source showed my exact `<script>` tag embedded in the `<p>` element, with no sanitization.
- Consistent Behavior Across Requests: Both endpoints consistently echoed back unfiltered input, making it easy to test different SQLi and XSS payloads.

4. Exploitation Steps

4.1 SQL Injection

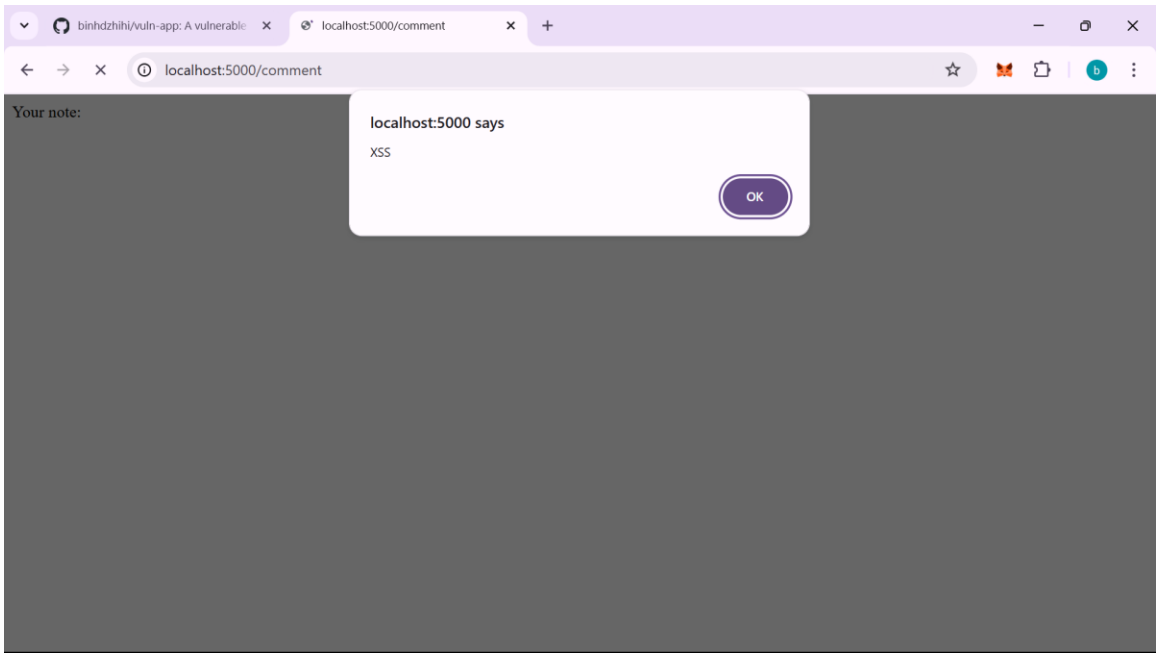
1. Open browser at http://localhost:5000
2. In Search users, type: ' OR '1'='1
3. Click Search to return all users (alice, bob, charlie)





4.2 Reflected XSS

4. On home page, under Leave a Note, enter: <script>alert('XSS')</script>
5. Click Post Note to trigger an alert with message 'XSS'



5. Root-Cause Analysis

- **SQL Injection Cause:** User input is directly interpolated into SQL without parameterization. Because there's no separation between code and data, an attacker can inject additional SQL syntax (e.g. **' OR '1'='1**) that the database will execute. This zero-parameter interpolation bypasses the database driver's protections and allows arbitrary SQL to run.
- **Reflected XSS Cause:** User-supplied text is rendered without HTML-escaping. Here, any **<script>...</script>** tags in the note string become part of the page's DOM and execute in the victim's browser. There is no HTML-encoding, input sanitization, or use of a templating filter to neutralize markup, so malicious scripts run immediately.

6. Proposed Mitigations

- **SQL Injection Mitigation:** Replace string interpolation with parameterized queries (prepared statements). This ensures the database treats user input purely as data, never as executable code. For example: **cursor.execute('SELECT ... WHERE username LIKE %s', (f'%{name}%',)).** Making the driver automatically escapes any special characters in pattern, preventing injected SQL from altering query logic.
- **Reflected XSS Mitigation:** Always HTML-escape any user input before adding to the page. For example: ** Your note: {{ note | e }} .** This converts < to <, > to > so any <script> tags appear as harmless text.