Security Vulnerability Report

I. Web Application Features

Purpose of the Application

This is a deliberately vulnerable Flask-based web application created for educational purposes to demonstrate common security vulnerabilities.

Key Features

- User search functionality with database backend
- User profile browsing
- File viewing capability
- MySQL database integration
- Basic frontend interface

II. Identifying Vulnerabilities

1. SQL Injection Vulnerability

Location: `/search` endpoint in `views.py`

```
@main.route('/search')
def search():
    name = request.args.get('name', '')
    # VULNERABLE: SQL Injection
    sql = f"SELECT id, name, bio FROM users WHERE name LIKE '%{name}%'"
    with conn.cursor(pymysql.cursors.DictCursor) as cursor:
        cursor.execute(sql)
```

2. Cross-Site Scripting (XSS) Vulnerability

Location: Templates that display unsanitized user input

Search results display user input without proper HTML escaping:

```
<h1>Search Results for: {% autoescape false %}{{ name }}{% endautoescape %
```

3. Local File Inclusion / Path Traversal

Location: `/show` endpoint in `views.py`

```
@main.route('/show')
def show_file():
    # VULNERABLE: Path Traversal
    filename = request.args.get('file', '')
    path = os.path.join('files', filename)

    try:
        if not os.path.exists(path):
            return f"File not found: {path}", 404

        return "<pre>" + open(path, 'r', errors='ignore').read() + "</pre>"
    except Exception as e:
        return f"Error: {str(e)}", 500
```

III. Black-Box Testing Methodology

Tools

- **Web Vulnerability Scanners**:
  - OWASP ZAP
  - Burp Suite
- **Specialized Testing Tools**:

- SQLmap (SQL injection testing)
- XSSer (XSS testing)
- Dirbuster (directory traversal)

Techniques

1. **Parameter Manipulation**:
   - Modifying URL parameters
   - Adding special characters to inputs
   - Testing boundary conditions
2. **Signature-Based Tests**:
   - Injecting known SQL injection patterns
   - Testing common XSS payloads
   - Attempting path traversal sequences

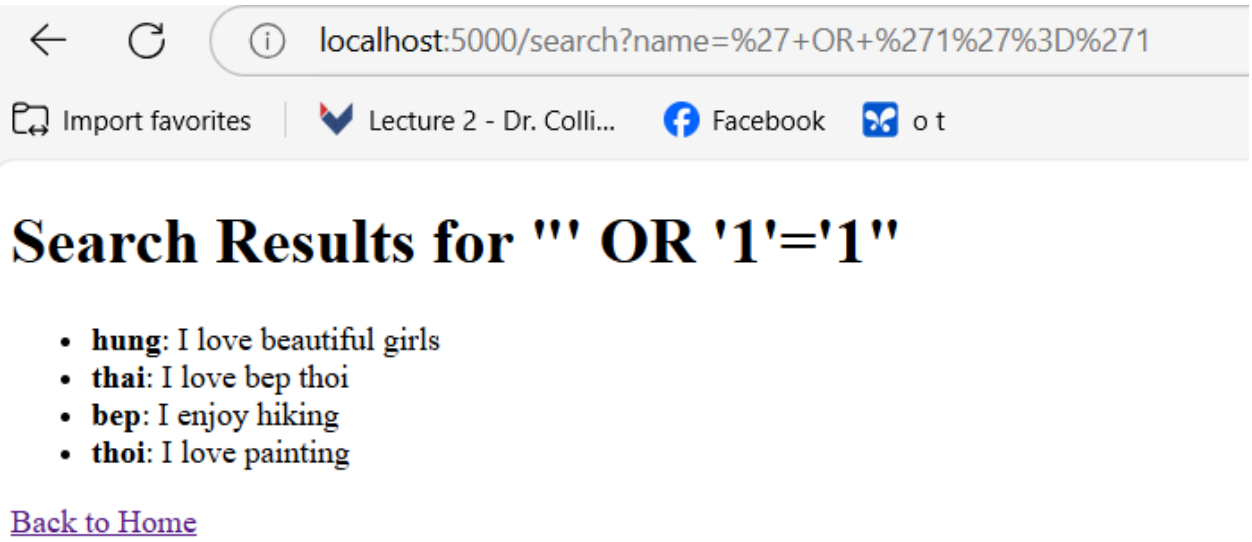
Identifying Signals

- Database error messages in responses
- Unexpected application behavior
- Successful JavaScript execution
- Access to unauthorized files

IV. Exploitation Steps

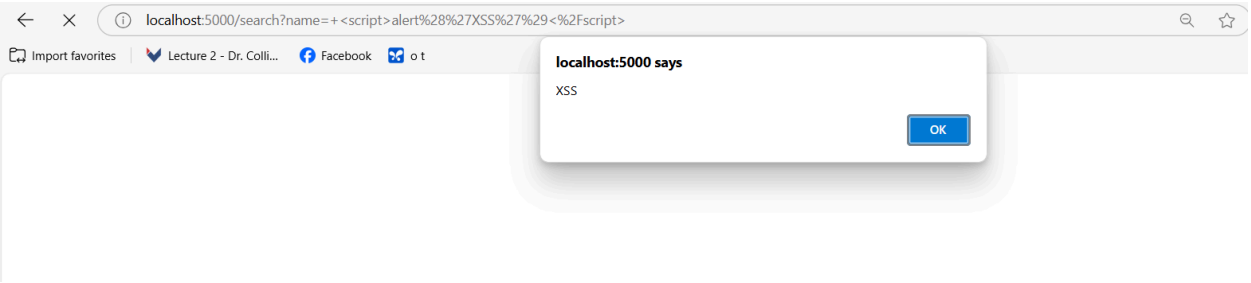
1. SQL Injection

1. Navigate to search page
2. Enter payload: `' OR '1'='1`
3. Observe all users being returned regardless of search term



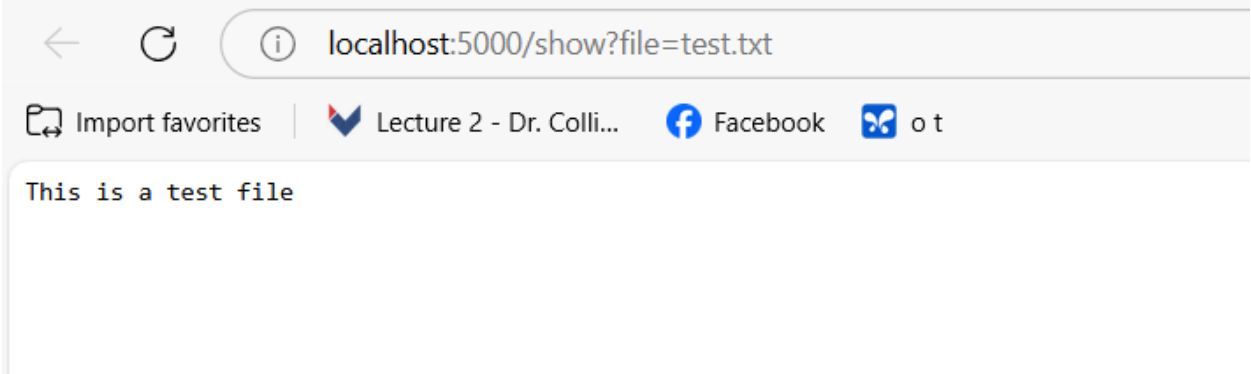
2. XSS

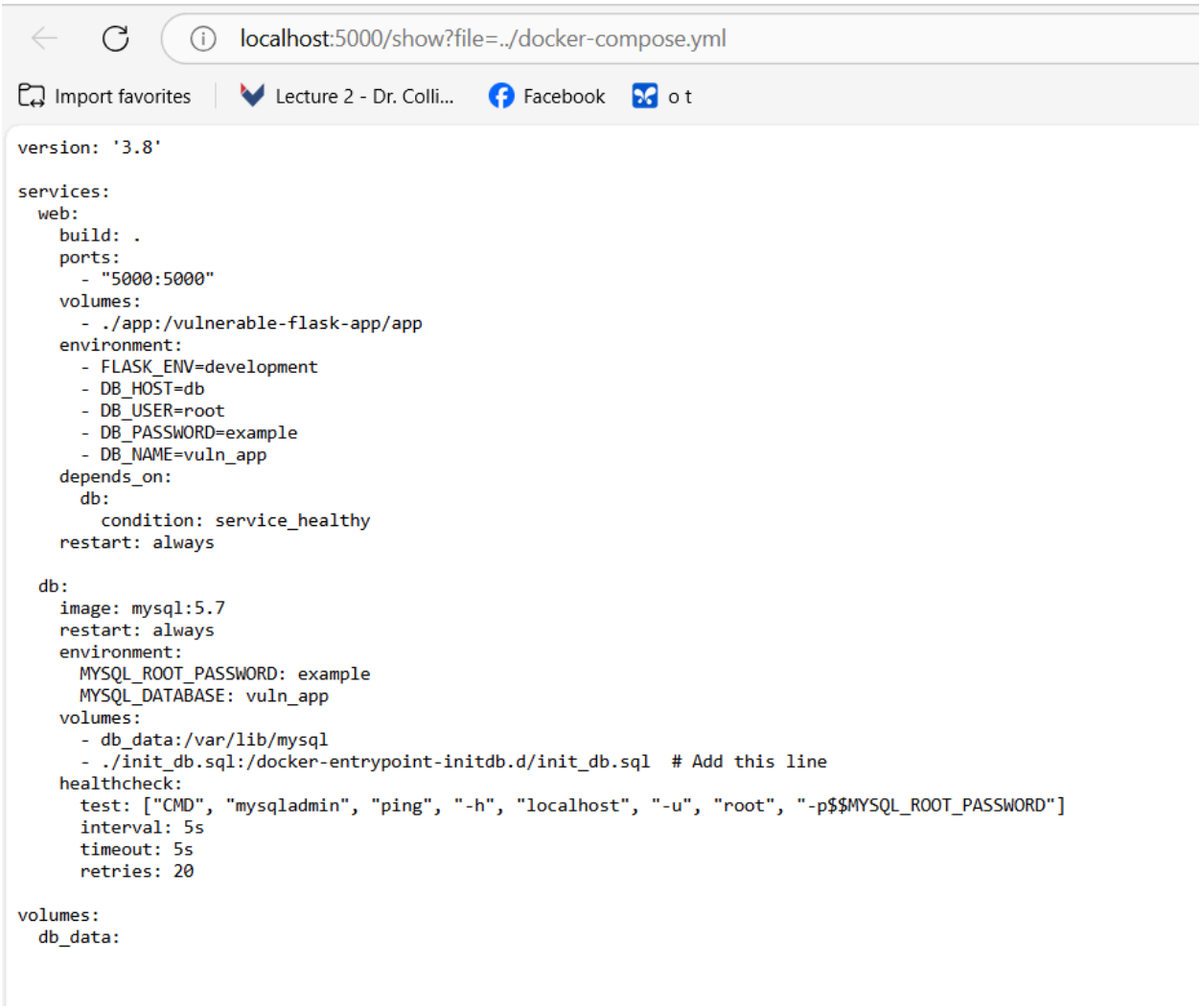
1. Navigate to search page
2. Enter payload: `<script>alert('XSS')</script>`
3. Submit the form



3. Path Traversal

1. Navigate to file viewing functionality
2. Enter payload: `../docker-compose.yml`
3. Submit the request






```
version: '3.8'

services:
  web:
    build: .
    ports:
      - "5000:5000"
    volumes:
      - ./app:/vulnerable-flask-app/app
    environment:
      - FLASK_ENV=development
      - DB_HOST=db
      - DB_USER=root
      - DB_PASSWORD=example
      - DB_NAME=vuln_app
    depends_on:
      db:
        condition: service_healthy
    restart: always

  db:
    image: mysql:5.7
    restart: always
    environment:
      MYSQL_ROOT_PASSWORD: example
      MYSQL_DATABASE: vuln_app
    volumes:
      - db_data:/var/lib/mysql
      - ./init_db.sql:/docker-entrypoint-initdb.d/init_db.sql  # Add this line
    healthcheck:
      test: ["CMD", "mysqladmin", "ping", "-h", "localhost", "-u", "root", "-p$$MYSQL_ROOT_PASSWORD"]
      interval: 5s
      timeout: 5s
      retries: 20

volumes:
  db_data:
```

V. Root Cause Analysis

SQL Injection

The vulnerability exists because user input is directly concatenated into the SQL query without parameterization. This allows attackers to modify the intended SQL query structure.

XSS

The vulnerability exists because user-supplied input is rendered directly in the HTML response without sanitization, allowing injection of arbitrary JavaScript code.

Path Traversal

The vulnerability exists because the application does not properly validate file paths, allowing navigation outside the intended directory using `../` sequences.

VI. Proposed Mitigations

SQL Injection

```
# SECURE VERSION
@main.route('/search')
def search():
    name = request.args.get('name', '')
    sql = "SELECT id, name, bio FROM users WHERE name LIKE %s"
    with conn.cursor(pymysql.cursors.DictCursor) as cursor:
        cursor.execute(sql, ('%' + name + '%',))
```

XSS

```
<!-- SECURE VERSION -->
<h1>Search Results for "{{ name|escape }}"</h1>
```

Path Traversal

```
# SECURE VERSION
@main.route('/show')
```

```
def show_file():
    filename = request.args.get('file', '')
    # Prevent directory traversal
    if '..' in filename or filename.startswith('/'):
        abort(403)

    safe_path = os.path.join(os.path.abspath('files'), filename)

    # Verify the resolved path is within allowed directory
    if not os.path.abspath(safe_path).startswith(os.path.abspath('files')):
        abort(403)

    if not os.path.exists(safe_path):
        abort(404)
    return "<pre>" + open(safe_path, 'r', errors='ignore').read() + "</pre>"
```