Stop SQL Injection

Student: Mohammed JBILOU
ID: 7702249
Course: Cybersecurity
Institution: Keimyung University
Github reposiroty:
https://github.com/Molaryy/KMU_cybersecurity_assignment_2025

Table of Contents

1 - Objective

The purpose of this lab is to demonstrate how SQL injection vulnerabilities arise and how to reliably defend against them.

By the end of the activity, the following skills should be demonstrated:

- Identifying insecure SQL query construction in web applications
- Exploiting a SQL injection vulnerability to bypass authentication
- Implementing secure coding techniques to eliminate injection risks
- Validating the effectiveness of security fixes through structured testing

SQL injection remains one of the most severe and widespread security issues. It appears in the OWASP Top 10 and allows attackers to intercept, manipulate or destroy sensitive data stored in backend databases.

2 - Application Architecture

This lab uses a simple full stack application with a Flask backend and a Next.js frontend. The entire environment is containerized using docker for convenience and isolation.

Backend (Flask)	Frontend (Next.js)	Application Structure

• **Framework**:
  ○ Flask
  ○ Python
• **Database**: SQLite3
• **Port**: 5001
	• **Framework**:
	  ○ Next.js
	  ○ TypeScript
	• **Port**: 3000	

Dockerfile

```
FROM python:3.11-slim

WORKDIR /app

COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

COPY . .

EXPOSE 5001

CMD ["python", "app.py"]
```

snappify.com

requirements.txt

```
Flask==3.0.0
flask-cors==4.0.0
```

snappify.com

Dockerfile

```
FROM node:20-alpine AS builder

WORKDIR /app

COPY package*.json ./
RUN npm ci

COPY . .
RUN npm run build

FROM node:20-alpine AS runner

WORKDIR /app

ENV NODE_ENV=production

COPY --from=builder /app/package*.json ./
COPY --from=builder /app/.next ./.next
COPY --from=builder /app/public ./public
COPY --from=builder /app/node_modules ./node_modules

EXPOSE 3000

CMD ["npm", "start"]
```

snappify.com

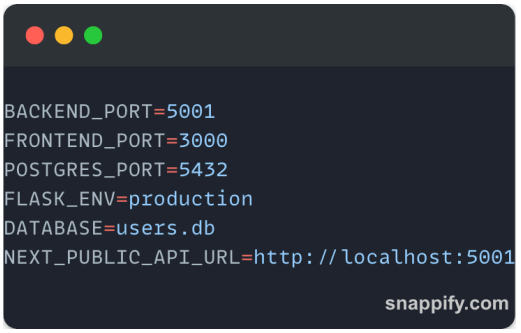
```
.
├── backend
│   ├── app.py
│   ├── Dockerfile
│   └── requirements.txt
├── docker-compose.yml
└── frontend
    ├── app
    │   ├── favicon.ico
    │   ├── globals.css
    │   ├── layout.tsx
    │   └── page.tsx
    ├── Dockerfile
    ├── eslint.config.mjs
    ├── next-env.d.ts
    ├── next.config.ts
    ├── package-lock.json
    ├── package.json
    ├── postcss.config.mjs
    ├── public
    │   ├── file.svg
    │   ├── globe.svg
    │   ├── next.svg
    │   ├── vercel.svg
    │   └── window.svg
    ├── README.md
    └── tsconfig.json
```

snappify.com

The **docker-compose.yml** file coordinates and runs the entire application stack, along with the .env configuration providing the environment variables that will be used thanks to the **env_file** field.

docker-compose.yml

.env



Backend

app.py contains the entire backend application. Its main responsibilities are:

- Initializing the Flask application
- Setting up the SQLite3 database connection
- Providing the /api/login endpoint
- Executing the SQL query that checks email and password
- Returning user data or an authentication failure response

In the vulnerable version, the login route builds the SQL query using string concatenation. In the fixed version, it uses a parameterized query to prevent SQL injection.

app.py

```python
from flask import Flask, request, jsonify
from flask_cors import CORS
import sqlite3
import os

app = Flask(__name__)
CORS(app, resources={
    r"/api/*": {
        "origins": "*",
        "methods": ["GET", "POST", "OPTIONS"],
        "allow_headers": ["Content-Type"]
    }
})

DATABASE = 'users.db'

def get_db_connection():
    conn = sqlite3.connect(DATABASE)
    conn.row_factory = sqlite3.Row
    return conn

def init_db():
# Init database with users when the application starts

@app.route('/api/login', methods=['POST'])
def login():
# Login logic

if __name__ == '__main__':
    if not os.path.exists(DATABASE):
        print("Initializing database...")
        init_db()
        print("Database initialized with sample data!")

    app.run(host='0.0.0.0', port=5001, debug=True)

```

This is the **init_db** function that will create users for the testing purpose:

app.py

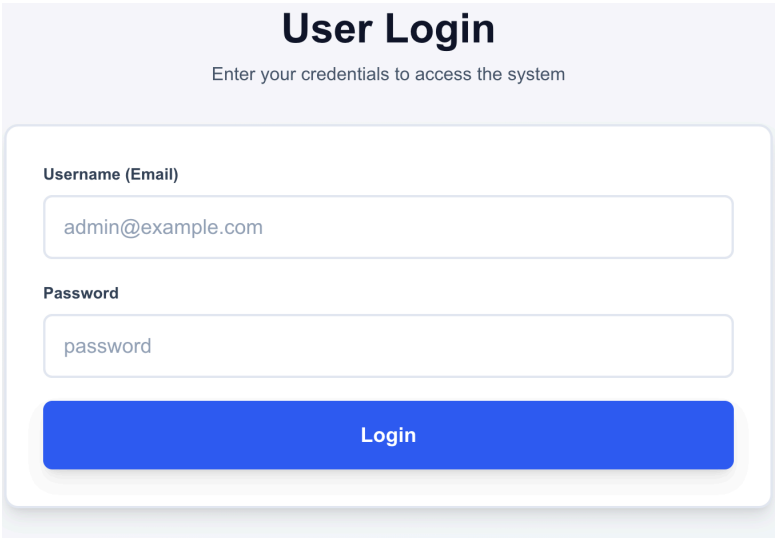
```python
def init_db():
    conn = get_db_connection()
    cursor = conn.cursor()

    cursor.execute('''
        CREATE TABLE IF NOT EXISTS users (
            id INTEGER PRIMARY KEY AUTOINCREMENT,
            name TEXT NOT NULL,
            email TEXT NOT NULL UNIQUE,
            password TEXT NOT NULL
        )
    ''')

    sample_users = [
        ('Alice Johnson', 'alice@example.com', 'password123'),
        ('Bob Smith', 'bob@example.com', 'securepass456'),
        ('Charlie Brown', 'charlie@example.com', 'mypassword789'),
        ('Admin User', 'admin@example.com', 'admin_secret_password'),
        ('David Wilson', 'david@example.com', 'david2024'),
    ]

    try:
        cursor.executemany(
            'INSERT INTO users (name, email, password) VALUES (?, ?, ?)',
            sample_users
        )
        conn.commit()
    except sqlite3.IntegrityError:
        pass

    conn.close()
```

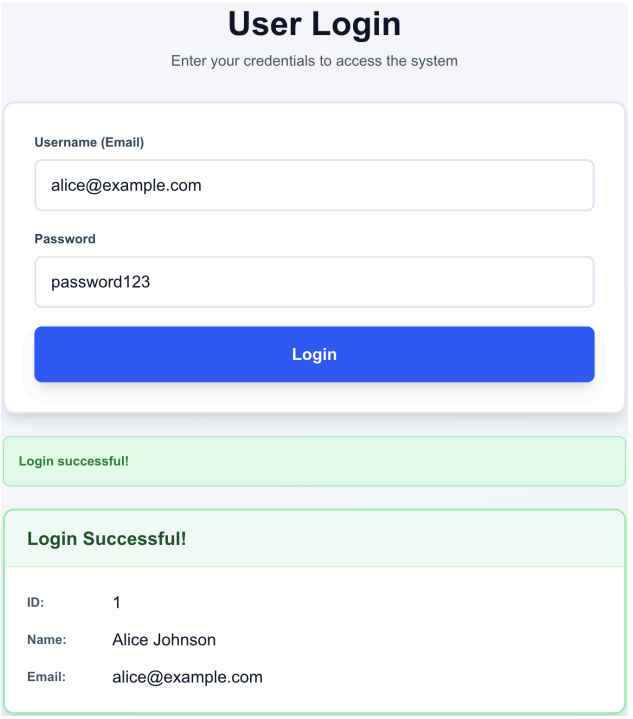
Frontend

page.tsx is the main page of the Next.js application. It does the following:

- Renders the login interface (email and password form)
- Captures user input
- Sends a POST request to the backend /api/login endpoint
- Displays either a success message or an error message based on the response

It serves as the user facing part of the application and acts as the bridge between the user and the backend authentication logic.





Here is the code for the main page, I removed most of the code so the reader can easily understand thanks to my comments, if not, please refer directly to the to file with the provided link of the github repository at the beginning.

app.tsx

```tsx
'use client';

import { useState } from 'react';

interface User {
  id: number;
  name: string;
  email: string;
}

export default function Home() {
  const [username, setUsername] = useState('');
  const [password, setPassword] = useState('');
  const [user, setUser] = useState<User | null>(null);
  const [error, setError] = useState('');
  const [message, setMessage] = useState('');
  const [loading, setLoading] = useState(false);

  const handleLogin = async (e: React.FormEvent) => {
    // Login logic with the API
  };

  return (
    <div className="flex min-h-screen ...">
      <div className="w-full max-w-2xl">
        <div className="text-center mb-8">
          <h1 className="text-4xl ...">
            User Login
          </h1>
          <p className="text-slate-600 dark:text-slate-400">
            Enter your credentials to access the system
          </p>
        </div>

        <div className="bg-white ...">
          <form onSubmit={handleLogin} className="space-y-5">
            {/* Inputs + Labels */}
            <button
              type="submit"
              disabled={loading}
              className="w-full px-8 py-4 text-lg ..."
            >
              {loading ? 'Logging in...' : 'Login'}
            </button>
          </form>
        </div>

        {/* Display message received from API */}

        {/* Error message if invalid email or password */}

        {/* Displaying user information if connected */)}
      </div>
    </div>
  );
}
```

3 - Setup Instructions

Prerequisites

- Docker
- Docker Compose

Running the Vulnerable Version:

```
cd exploit && docker compose up -d --build
```

Running the Patched Version:

```
cd fixed && docker compose up -d --build
```

```
docker ps

CONTAINER ID   IMAGE              COMMAND                  CREATED              STATUS              PORTS                                                 NAMES
ffee9aa84711   exploit-frontend   "docker-entrypoint.s…"   About a minute ago   Up About a minute   0.0.0.0:3000→3000/tcp, :::3000→3000/tcp               exploit-frontend
eeddccdf370d   exploit-backend    "python app.py"          About a minute ago   Up About a minute   0.0.0.0:5001→5001/tcp, :::5001→5001/tcp               exploit-backend
```


Then you'll be able to visit: http://localhost:3000 to try the SQL injection, you can first try connecting as:

email: alice@example.com
password: password123

4 - Phase 1: Exploit

Identifying the Vulnerability

The login form takes an email and password and sends them to the backend API. Inside the vulnerable version of the backend, the SQL query is constructed using direct string interpolation:

app.py

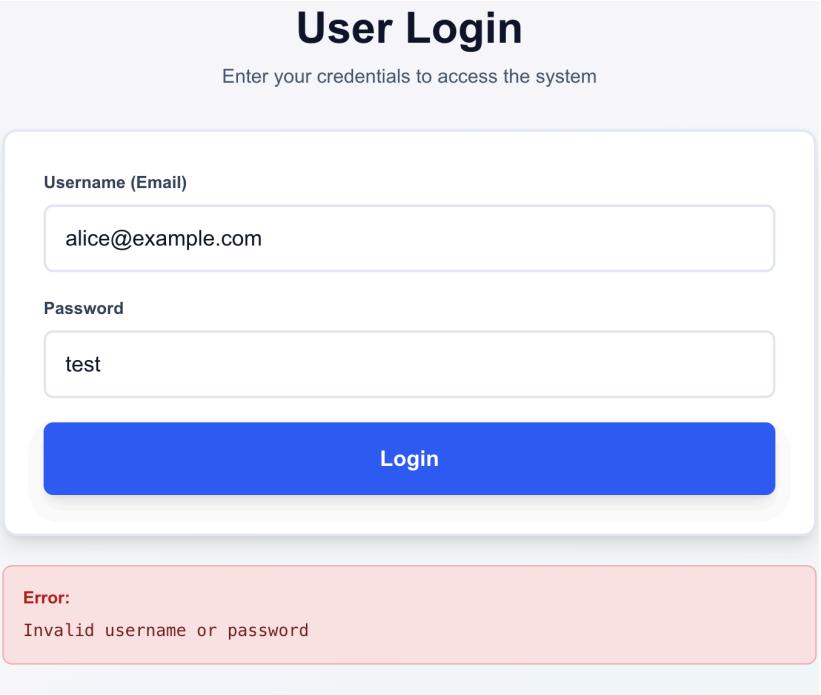
```
query = f"SELECT id, name, email FROM users WHERE email = '{username}' AND password = '{password}'"
```

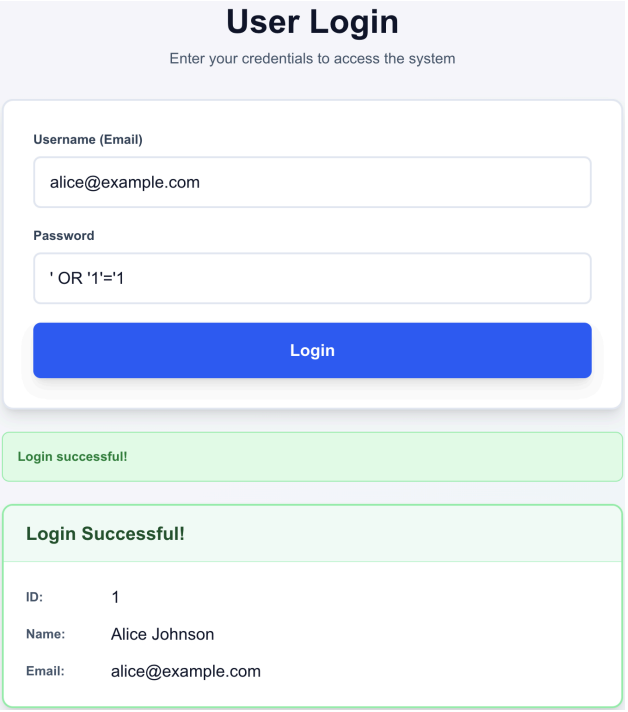

This design is unsafe because user input is inserted directly into the SQL query. No parameterization, escaping or sanitization is applied.

Crafting the SQL Injection Payload

Goal: Bypass the authentication process without knowing valid credentials.

Payload:





Since the expression '1'='1' always returns true, the entire WHERE clause becomes true, allowing the attacker to gain access as the first matching user.

Observed Behavior

After the injection payload is submitted:

- Authentication is bypassed
- User data from the database is returned
- The query used by the backend is visible
- The attacker is logged in as the first database user

This vulnerability compromises confidentiality, authentication integrity and overall system trust.

5 - Phase 2: Patch

Locating the Vulnerable Code

The insecure logic is found in /api/login inside backend/app.py. The vulnerable query is created by directly inserting user input into an SQL string.

app.py

```python
@app.route('/api/login', methods=['POST'])
def login():
    data = request.get_json()
    username = data.get('username', '')
    password = data.get('password', '')

    query = f"SELECT id, name, email FROM users WHERE email = '{username}' AND password = '{password}'"

    conn = get_db_connection()
    cursor = conn.cursor()

    try:
        cursor.execute(query)
        result = cursor.fetchone()

        if result:
            user = {
                'id': result['id'],
                'name': result['name'],
                'email': result['email']
            }

            conn.close()

            return jsonify({
                'success': True,
                'message': 'Login successful!',
                'user': user,
                'query': query
            })
        else:
            conn.close()

            return jsonify({
                'success': False,
                'message': 'Invalid username or password',
                'query': query
            }), 401

    except Exception as e:
        conn.close()
        return jsonify({
            'success': False,
            'error': str(e),
            'query': query
        }), 500
```

Implementing the fix

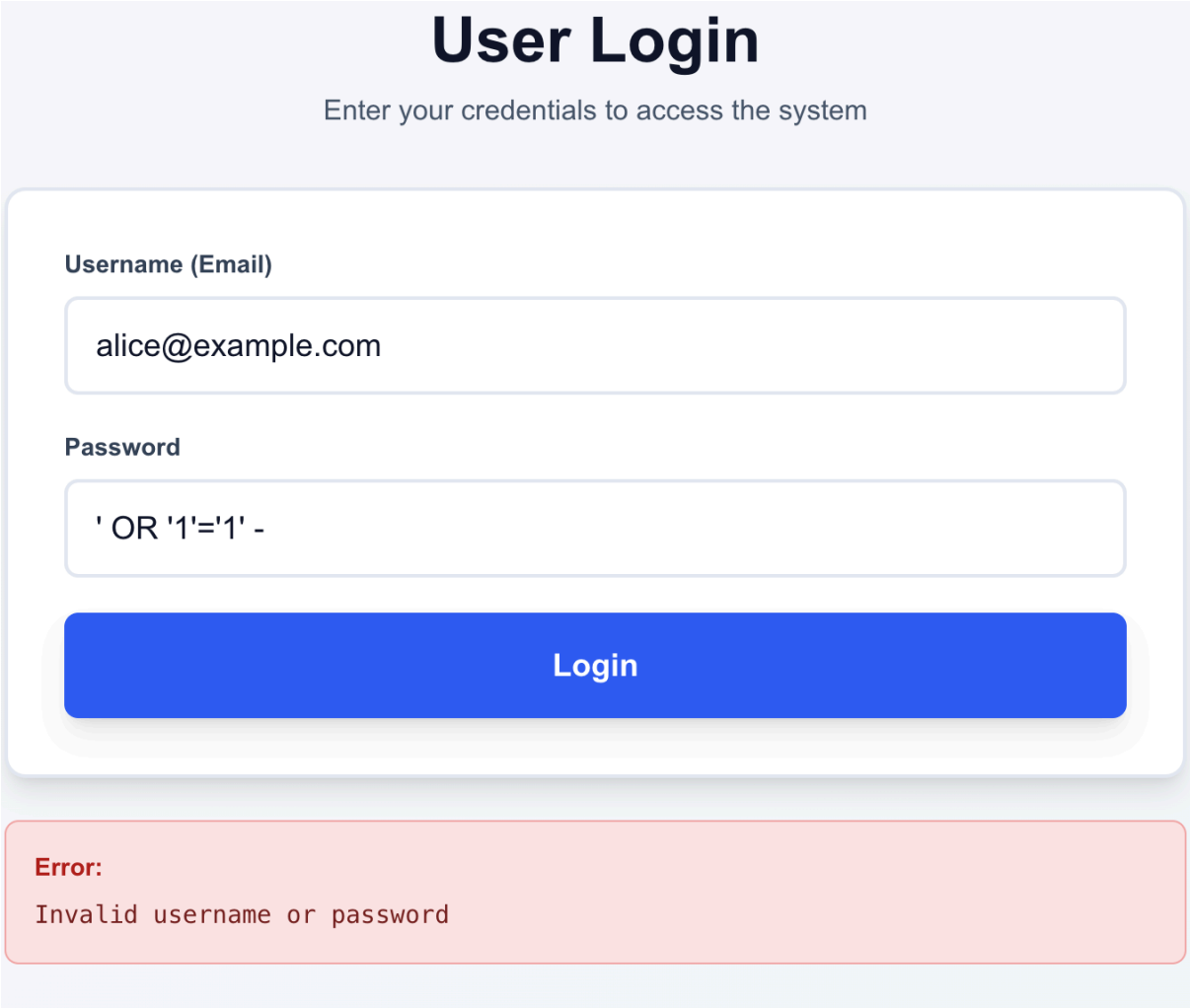
The recommended solution is to use parameterized queries. This prevents user input from being interpreted as SQL code.

Secure Version:

app.py

```python
@app.route('/api/login', methods=['POST'])
def login():
    data = request.get_json()
    username = data.get('username', '')
    password = data.get('password', '')

    query = "SELECT id, name, email FROM users WHERE email = ? AND password = ?"

    conn = get_db_connection()
    cursor = conn.cursor()

    try:
        cursor.execute(query, (username, password))
        result = cursor.fetchone()

        if result:
            user = {
                'id': result['id'],
                'name': result['name'],
                'email': result['email']
            }

            conn.close()

            return jsonify({
                'success': True,
                'message': 'Login successful!',
                'user': user,
                'query': f"{query} [Parameters: email={username}, password=***]"
            })
        else:
            conn.close()

            return jsonify({
                'success': False,
                'message': 'Invalid username or password',
                'query': f"{query} [Parameters: email={username}, password=***]"
            }), 401

    except Exception as e:
        conn.close()
        return jsonify({
            'success': False,
            'error': str(e),
            'query': f"{query} [Parameters: email={username}, password=***]"
        }), 500
```

Now the SQL Injection doesn’t work anymore:



This approach ensures that:

- SQL structure is fixed and not influenced by user input
- User provided values are handled as data only
- Database drivers automatically escape special characters

Aspect	Vulnerable Version	Secure Version
Query format	Uses f-strings	Uses placeholders
Parameter Handling	Embedded directly	Passed separately
Injection Risk	Very high	Eliminated
Special Characters	Treated as SQL code	Escaped automatically

6 - Conclusion

This lab demonstrated the full lifecycle of an SQL injection vulnerability, beginning with the discovery of a critical flaw in the login logic. The backend relied on string concatenation to build SQL queries, which made it possible for user input to alter the query structure. By crafting a simple injection payload, the authentication mechanism was bypassed, showing how easily an attacker can exploit poorly handled input.

After confirming the vulnerability, the backend code was refactored to use parameterized queries. This change ensured that user provided values were treated strictly as data rather than executable SQL, effectively removing the injection vector. The fix was then validated through a series of tests, including malicious payloads, valid credentials and multiple edge cases, all of which confirmed that the patch blocked the attack while preserving expected functionality.

The lab reinforced several important security concepts: SQL injection occurs when user input is mixed directly with SQL commands, when validation is absent and when logic and data are not clearly separated. Preventing these issues requires using prepared statements, validating input, limiting database privileges and adopting safe design patterns. Additional defense in depth measures such as proper error handling, permission restrictions and regular code reviews further reduce risk.

By completing all stages of the exploit, patch and testing phases, the lab provided a practical and comprehensive understanding of SQL injection risks and the secure coding practices required to mitigate them effectively.

7 - References

- OWASP Top 10: https://owasp.org/www-project-top-ten/
- OWASP SQL Injection Prevention Cheat Sheet: https://cheatsheetseries.owasp.org/cheatsheets/SQL_Injection_Prevention_Cheat_Sheet.html
- SQLite Parameterized Queries: https://docs.python.org/3/library/sqlite3.html
- Flask Security Guidelines: https://flask.palletsprojects.com/en/stable/web-security/
- CWE 89: SQL Injection: https://cwe.mitre.org/data/definitions/89.html
- Python DB API Specification: https://peps.python.org/pep-0249/