Lab 5- SQL Injection Prevention & Data Leak Security

1. Prerequisites (local machine)

1. Install Node.js (LTS), npm/yarn, Git.
2. Install Heroku CLI: https://devcenter.heroku.com/articles/heroku-cli
3. Install PostgreSQL locally for development.
4. Install SQLMap on your machine (pip install sqlmap or package for your OS) — only for testing your own app.
5. Recommended dev tools: VS Code, Postman / curl.

2. Create project & initialize Git + Heroku app

Run these in your project root or terminal. Replace placeholders (e.g., <app-name>) with your values.

```
mkdir heroku-sql-demo
cd heroku-sql-demo
git init
npm init -y
npm i express pg bcryptjs dotenv helmet express-rate-limit express-validator
# optional for dev:
npm i -D nodemon
```

Login and create Heroku app + add Postgres:

```
heroku login
heroku create my-sql-injection-demo-<your-unique-suffix>
heroku addons:create heroku-postgresql:hobby-dev
# (Heroku will create DATABASE_URL in config vars automatically)
```

Check config:

```
heroku config --app my-sql-injection-demo-<suffix>
# should show DATABASE_URL
```

3. Structure of the Project:

```
heroku-sql-demo/
├── server.js
├── package.json
├── public/
│   ├── login-vulnerable.html
│   ├── login-secure.html
│   └── style.css
```

i) server.js

```
const express = require('express');
const { Pool } = require('pg');
const path = require('path');

const app = express();
app.use(express.json());
app.use(express.urlencoded({ extended: true })); // to parse form data

// Serve static files from /public
app.use(express.static(path.join(__dirname, 'public')));

// Connect to Postgres
const pool = new Pool({
  connectionString: process.env.DATABASE_URL,
  ssl: { rejectUnauthorized: false },
});
// --- Home Page ---
app.get('/', (req, res) => {
  res.send(`
    <h2>SQL Injection Demo</h2>
    <p>⚠ Try <a href="/login-vulnerable.html">/login-vulnerable</a> to see SQL
Injection in action (unsafe)</p>
    <p>✅ Try <a href="/login-secure.html">/login-secure</a> to see the safe
version (parameterized)</p>
  `);
});
// --- Vulnerable handler (uses string concatenation ❌) ---
app.post('/login-vulnerable', async (req, res) => {
  const { username, password } = req.body;
  try {
    const query = `SELECT * FROM users WHERE username='${username}' AND
password='${password}';
    console.log("Executing query:", query); // for debugging

    const result = await pool.query(query);

    if (result.rows.length > 0) {
      res.send('⚠ Login successful (but vulnerable to SQL Injection)!');
    } else {
      res.status(401).send('❌ Unauthorized');
    }
  } catch (err) {
    console.error(err);
    res.status(500).send('⚠ Server error');
  }
});
// --- Secure handler (parameterized ✅) ---
app.post('/login-secure', async (req, res) => {
  const { username, password } = req.body;
  try {
    const result = await pool.query(
      'SELECT * FROM users WHERE username=$1 AND password=$2',
      [username, password]
    );
    if (result.rows.length > 0) {
      res.send('✅ Login successful (safe from SQL Injection)!');
    } else {
      res.status(401).send('❌ Unauthorized');
    }
  } catch (err) {
    console.error(err);
    res.status(500).send('⚠ Server error');
  }
});
app.listen(process.env.PORT || 5000, () => console.log('🚀 Running'));
```

ii) login-vulnerable.html

```
<!DOCTYPE html>
<html>
<head>
  <title>Vulnerable Login</title>
</head>
<body>
  <h2>⚠ Vulnerable Login (Unsafe Example)</h2>
  <form action="/login-vulnerable" method="POST">
    <input type="text" name="username" placeholder="Username" required /><br/><br/>
    <input type="password" name="password" placeholder="Password" required
/><br/><br/>
    <button type="submit">Login</button>
  </form>

  <p style="color:red;">
    This form is vulnerable to SQL Injection.<br>
    Try: <b>'' OR '1'='1'</b> in the username or password field.
  </p>

  <p><a href="/">⬅ Back to Home</a></p>
</body>
</html>
```

iii) login-secure.html

```
<!DOCTYPE html>
<html>
<head>
  <title>Secure Login</title>
</head>
<body>
  <h2>✅ Secure Login (Parameterized Queries)</h2>
  <form action="/login-secure" method="POST">
    <input type="text" name="username" placeholder="Username" required /><br/><br/>
    <input type="password" name="password" placeholder="Password" required
/><br/><br/>
    <button type="submit">Login</button>
  </form>

  <p style="color:green;">
    This form is safe because it uses parameterized queries.<br>
    SQL Injection attempts will fail.
  </p>

  <p><a href="/">⬅ Back to Home</a></p>
</body>
</html>
```

4. Initialising PostgreSQL in Heroku:

```
heroku pg:psql --app <my-sql-injection-demo-name>


Inside the psql prompt, create your demo table (example users):
CREATE TABLE users (
    id SERIAL PRIMARY KEY,
    username VARCHAR(100),
    password VARCHAR(100)
);


INSERT INTO users (username, password) VALUES
('admin', 'admin123'),
('test', 'test123');


Exit psql:
\q
```

5. Deploy your app code

If your code is already in Git:

```
git add .
git commit -m "Deploy SQL injection demo"
git push heroku main
(If your branch is master, replace main with master.)
```

Scale dynos (make sure the web process runs)

```
heroku ps:scale web=1 --app my-sql-injection-demo-name
```

Open your app

```
heroku open --app my-sql-injection-demo-name
```

NOTE: **If Git identity not configured**

Git doesn't know who you are. Set your name/email first:

```
git config --global user.name "XYZ"
git config --global user.email "xyz.cs23@rvce.edu.in"
```

6. Test SQL Injection

Install **sqlmap** (on your local machine):

```
pip install sqlmap
```

Run a test (replace <appname> with your Heroku app):

```
sqlmap -u "https://my-sql-injection-demo-name.herokuapp.com/login-secure" \
--data="username=admin&password=wrong" --risk=3 --level=5
```

- If your query is secure, sqlmap will report: **"parameter not injectable"**.
- Try a manual test in the browser:
  Username: ' OR '1'='1
  Password: anything
  With parameterized queries, this should **fail** (unauthorized).

7. Monitor Logs

While testing, keep logs open:

```
heroku logs --tail --app my-sql-injection-demo-name
```

You'll see SQL errors (if any) or blocked attempts.

Alternative Method for Testing:

We can also test the SQL injection directly through login screen:

Open the app

```
heroku open
```

Then test:

- [your-app]/login-vulnerable.html
- [your-app]/login-secure.html

Test Case 1 — Normal Login (Valid User)

1. Go to

```
https://your-app.herokuapp.com/login-vulnerable.html
```

2. Enter a **real username and password** (must exist in your users table).

Expected: You see

⚠ Login successful (but vulnerable to SQL Injection)!

Test Case 2 — Wrong Password (Normal Behavior)

1. Enter a real username but wrong password.
   Expected:

   Unauthorized

Test Case 3 — SQL Injection Attack

Now let's bypass authentication using SQL Injection.

1. Go to the vulnerable page again:

   https://your-app.herokuapp.com/login-vulnerable.html

2. Enter:
   - **Username**: ' OR '1'='1
   - **Password**: anything (e.g., abc)
3. What happens?
   - The backend query becomes:
   - SELECT * FROM users WHERE username='' OR '1'='1' AND password='abc';
   - Because '1'='1' is **always true**, Postgres returns all rows.
     ✅ Expected:

⚠ Login successful (but vulnerable to SQL Injection)!

Test Case 4 — Secure Page Protection

1. Go to:

   https://your-app.herokuapp.com/login-secure.html

2. Try the same injection:
   - **Username**: ' OR '1'='1
   - **Password**: abc
3. Since parameterized queries are used:
4. SELECT * FROM users WHERE username=$1 AND password=$2;

This looks for a literal username of ' OR '1'='1 (not evaluated as SQL).

Expected:

Unauthorized

Sample Output/Images: