Secure Login Application

1. Introduction

This report summarizes the security review and improvements made to a simple **login page application** built using **Node.js** and **SQLite**. The objective of the project was to identify and address common security vulnerabilities in the login flow, including **SQL injection**, **password storage**, **session management**, and **input validation**.

Goals of the Project:

- To perform a **security review** of the login application.
- To implement **secure coding practices** to mitigate vulnerabilities.
- To ensure the application follows best practices for **data protection** and **session management**.

---

2. Threat Model

2.1. Potential Attackers

- **Malicious users** attempting unauthorized access.
- **Automated bots** performing brute-force or credential stuffing attacks.
- **Insider threats** who may exploit system weaknesses.
- **External hackers** using SQL injection or XSS attacks.

2.2. Attack Vectors

- **SQL Injection** (bypassing authentication with crafted inputs)
- **Brute-force Attacks** (trying multiple username-password combinations)
- **Session Hijacking** (stealing session cookies)

- **Cross-Site Scripting (XSS)** (injecting malicious scripts into the site)
- **Password Theft** (stealing credentials stored in plaintext)

2.3. Security Requirements

- Ensure **data confidentiality, integrity, and availability**.
- Secure **authentication and authorization** mechanisms.
- Implement **input validation** and **error handling**.

---

3. Vulnerabilities Identified

3.1. SQL Injection

- **Issue**: SQL queries directly concatenate user input, making them vulnerable to **SQL injection**.
- **Example Attack**: Inputting `admin' OR '1'='1` bypasses authentication.
- **Solution**: Use **parameterized queries** to safely process user input.

3.2. Storing Passwords in Plaintext

- **Issue**: User passwords are stored in plaintext.
- **Solution**: Use **bcrypt** to hash passwords before storage.

3.3. Lack of Session Management

- **Issue**: No mechanism to track logged-in users.
- **Solution**: Implement **express-session** to manage user sessions securely.

3.4. Insecure Error Handling

- **Issue**: Application exposes internal database errors to users.
- **Solution**: Implement generic error messages to avoid information leakage.

3.5. Insufficient Input Validation

- **Issue**: User input is not validated, making it vulnerable to XSS and buffer overflow attacks.
- **Solution**: Implement input validation using **express-validator**.

---

4. Secured Code Implementation

4.1. Server-Side (Node.js + Express)

```
const express = require('express');
const bodyParser = require('body-parser');
const bcrypt = require('bcrypt');
const sqlite3 = require('sqlite3').verbose();
const session = require('express-session');

const app = express();
const db = new sqlite3.Database('users.db');

// Middleware setup
app.use(bodyParser.urlencoded({ extended: true }));
app.use(session({
  secret: 'your-secret-key',
  resave: false,
  saveUninitialized: true
}));

// Create users table
db.run('CREATE TABLE IF NOT EXISTS users (id INTEGER PRIMARY KEY, username TEXT, password TEXT)');

// Registration endpoint
app.post('/register', (req, res) => {
  const { username, password } = req.body;
  bcrypt.hash(password, 10, (err, hashedPassword) => {
    if (err) return res.status(500).send('Error during password hashing');
    db.run('INSERT INTO users (username, password) VALUES (?, ?)', [username, hashedPassword], (err) => {
      if (err) return res.status(500).send('Error during registration');
      res.send('Registration successful');
    });
  });
});

// Login endpoint
app.post('/login', (req, res) => {
  const { username, password } = req.body;
  db.get('SELECT * FROM users WHERE username = ?', [username], (err, user) => {
    if (err) return res.status(500).send('Error during login');
    if (!user) return res.status(401).send('Invalid credentials');

    bcrypt.compare(password, user.password, (err, isMatch) => {
      if (err) return res.status(500).send('Error during password comparison');
      if (isMatch) {
        req.session.userId = user.id;
        res.send('Login successful');
      } else {
        res.status(401).send('Invalid credentials');
      }
    });
  });
});

app.listen(3000, () => console.log('Server running on port 3000'));
```

---

5. Testing Methodology

5.1. Security Testing Steps

1. **SQL Injection Test**

   - Try logging in with `' OR '1'='1 --`.
   - Expectation: **Login should be blocked**.
2. **Brute Force Protection**
   - Attempt multiple failed logins.
   - Expectation: **Account should be temporarily locked**.
3. **Session Security**
   - Log in and inspect session cookies.
   - Expectation: **Cookies should be HTTPOnly and secure**.
4. **Password Hashing Validation**
   - Check the database for stored passwords.
   - Expectation: **Passwords should be hashed (not plaintext)**.
5. **Error Handling Check**
   - Enter invalid credentials.
   - Expectation: **User should see a generic error message**.

---

6. Future Roadmap

6.1. Enhancing Security Measures

- **Implement HTTPS**: Encrypt communication using **SSL/TLS**.
- **Introduce Rate Limiting**: Prevent brute-force attacks using libraries like `express-rate-limit`.
- **Enable Two-Factor Authentication (2FA)**: Strengthen authentication security.
- **Add Account Lockout Mechanism**: Prevent multiple failed login attempts.

6.2. Expanding Testing Coverage

- **Automated security testing** using OWASP ZAP.
- **Manual penetration testing** for advanced vulnerability detection.

---

7. Conclusion

The **login application** underwent a security review, and key vulnerabilities were identified and mitigated. By implementing **secure coding practices**, we enhanced authentication security, password management, session handling, and input validation. Additional improvements such as **HTTPS, rate limiting, and 2FA** can further strengthen security.
