TASK 3: Secure Coding Review

Project: Simple Node.js Login System

Author: Islam Gadirli

1. Introduction

This report presents a secure coding review of a small Node.js login system. The goal is to identify potential security vulnerabilities, assess their severity, and provide recommendations for secure coding practices. The reviewed application includes basic user registration and login functionality using Express.js.

```
const express = require('express');
const bodyParser = require('body-parser');

const app = express();
app.use(bodyParser.json());

// Simple in-memory “database”
const users = [];

// Register endpoint
app.post('/register', (req, res) => {
    const { username, password } = req.body;

    if (!username || !password) {
        return res.status(400).send('Missing username or password');
    }

    // Check if user exists
    const exists = users.find(u => u.username === username);
    if (exists) return res.status(400).send('User already exists');

    // Store password as plain text
    users.push({ username, password });
    res.send('User registered');
});

// Login endpoint
app.post('/login', (req, res) => {
    const { username, password } = req.body;

    const user = users.find(u => u.username === username && u.password === password);
    if (!user) return res.status(400).send('Invalid username or password');

    res.send('Login successful');
});

// Start server
app.listen(3000, () => console.log('Server running on http://localhost:3000'));
```

2. Application Overview

Programming Language: JavaScript (Node.js)

Framework: Express.js

Parser: body-parser middleware

Data Storage: In-memory JavaScript array

Functionality:

1. Register new users (username and password)
2. Authenticate users via login
3. Store user information in-memory

Design Notes:

- Single-file implementation (app.js)
- Passwords stored in plain text for demonstration purposes
- No authentication tokens, session management, or rate limiting

```
1  const express = require('express');
2  const bodyParser = require('body-parser');
3
4  const app = express();
5  app.use(bodyParser.json());
```

```
7  // Simple in-memory "database"
8  const users = [];
```

3. Vulnerability Assessment

Vulnerability	Severity	Location / File	Description	Recommendation / Fix
Plain Text Passwords	High	app.js – registration	Passwords are stored in plain text. Exposes user credentials.	Hash passwords using bcrypt or similar before storage.
No Input Validation	Medium-High	app.js – all endpoints	User inputs are not validated, risking injection attacks.	Validate and sanitise inputs (e.g., regex, express-validator).
Missing Authentication Tokens	Medium	app.js – login	Login only returns a success message; no JWT or session is generated.	Implement JWT or session management to secure routes.
Detailed Error Messages	Medium	app.js – login	Reveals whether the username exists; aids attackers.	Use generic error messages: "Invalid username or password".
No Rate Limiting / Brute Force Protection	High	app.js – login	Unlimited login attempts are possible.	Use express-rate-limit and implement account lockouts.
In-Memory Storage	Low-Medium	app.js – users' array	Users reset on server restart; not secure for production.	Use a secure database (MongoDB, PostgreSQL) with access control.

Plain Text Password Storage – HIGH

Passwords are stored without hashing. This exposes users to credential theft.

Fix Recommendation:

users.push({ username, password }); ➔ const hashed = await bcrypt.hash(password, 10);

Missing Authentication Tokens – MEDIUM

No session, no JWT means it cannot secure further routes.

Fix Recommendation:

res.send('Login successful'); ➔ const token = jwt.sign({ username }, SECRET_KEY);

Detailed Error Messages – MEDIUM

Although this error is now generic, the register endpoint leaks information. This reveals whether a username is registered, helping attackers enumerate accounts.

Fix Recommendation:

if (!user) return res.status(400).send('Invalid username or password'); ➔ if (exists) return res.status(400).send('User already exists');

4. Recommendations & Best Practices

- ✓ Password Security: Always hash passwords with a strong algorithm (bcrypt, argon2).
- ✓ Input Validation: Sanitise and validate all inputs to prevent injections.
- ✓ Authentication: Use JWT or sessions for login and protected routes.
- ✓ Error Handling: Avoid revealing sensitive information in errors.
- ✓ Rate Limiting: Limit login attempts to prevent brute-force attacks.
- ✓ Secure Storage: Move from in-memory arrays to a secure database.
- ✓ Transport Security: Use HTTPS/TLS for all communication.
- ✓ Environment Variables: Store secrets in .env files, not in code.

5. Conclusion

The reviewed Node.js login system contains several high-severity vulnerabilities, primarily involving password storage, missing authentication mechanisms, and weak input validation.

By implementing the recommended security controls, the application can be significantly strengthened and made suitable for real-world scenarios.