

Internship: Cybersecurity - Mock Web Application Testing

Intern: Umar Farooq

Project Title: Vulnerable Web App - XSS , SQL & NoSQL Injection

Contents

Week 1 Security Assessment Report	4
1. Application Setup and Exploration.....	4
Setup Process	4
Pages Explored	4
2. Basic Vulnerability Assessment.....	5
Vulnerabilities Discovered.....	6
Cross-Site Scripting (XSS)	6
SQL Injection (Authentication Bypass).....	6
Example Vulnerable Code (Node.js with MySQL)	6
Why It's Vulnerable:	7
NoSQL Injection (Authentication Bypass)	7
3. Weak Password Storage	7
4. Security Misconfigurations	8
Areas of Improvement	8
5. Summary	8
6. Attachments & Evidence	8
List of Alerts & Their Meanings:.....	12
Absence of Anti-CSRF Tokens (2).....	12
CSP: Failure to Define Directive with No Fallback (2)	12
Content Security Policy (CSP) Header Not Set (3).....	12
Missing Anti-Clickjacking Header (3).....	12
X-Content-Type-Options Header Missing (3)	13
Why Are These Alerts Important?	13
Week 2: Implementing Security Measures in Node.js Express Application	14
Overview	14

1. Input Validation and Sanitization	14
Problem	14
Solution	14
Implementation	14
2. Password Hashing	15
Problem	15
Solution	15
Implementation	15
3. Token-Based Authentication	15
Problem	15
Solution	15
Implementation	16
4. Cross-Site Scripting (XSS) Prevention	16
Problem	16
Solution	16
Basic Fix	16
5. Secure HTTP Headers	17
Problem	17
Solution	17
Implementation	17
6. NoSQL Injection Protection	17
Problem	17
Solution	17
Fix	17
7. Conclusion	18
Week 3: Advanced Security and Final Reporting	19
1. Basic Penetration Testing	19
Tool Used: Nmap	19
Command Executed:	19

Scan Result:	19
Interpretation:.....	19
2. Set Up Basic Logging	19
Why Logging is Important	20
Implementation Snippet:	20
Where Logging Was Used:	20
Output File: security.log.....	21
3. Create a Simple Security Checklist.....	21
1. Input Validation	21
Example:.....	21
2. Password Hashing and Salting	21
Example:.....	21
3. Secure Data Transmission (HTTPS).....	21
4. Logging	22
5. Other Security Measures	22

Week 1 Security Assessment Report

1. Application Setup and Exploration

Setup Process

Source: Custom vulnerable app built using Node.js, Express, MongoDB, EJS.

Dependencies Installed:

npm install

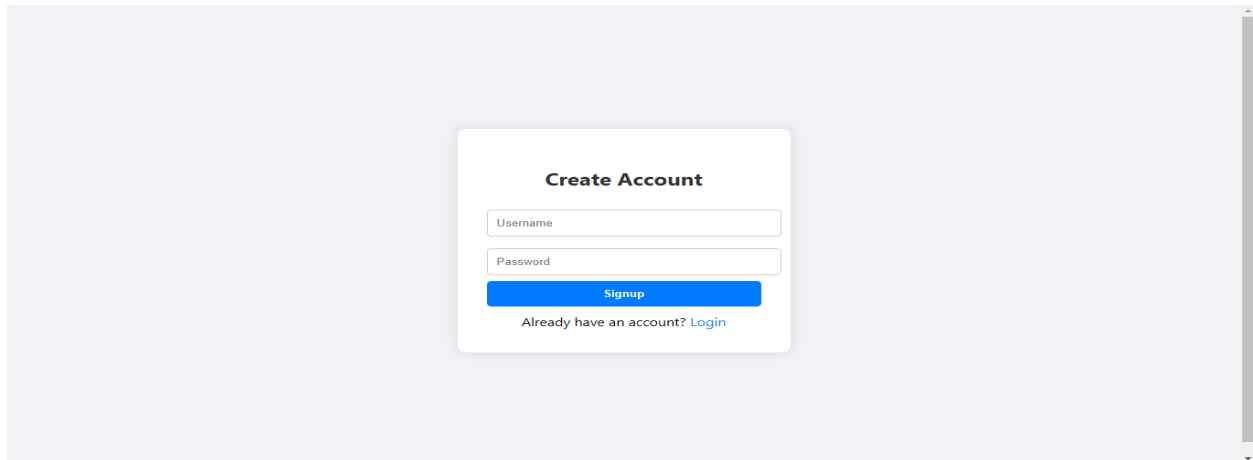
npm start

Application Access:

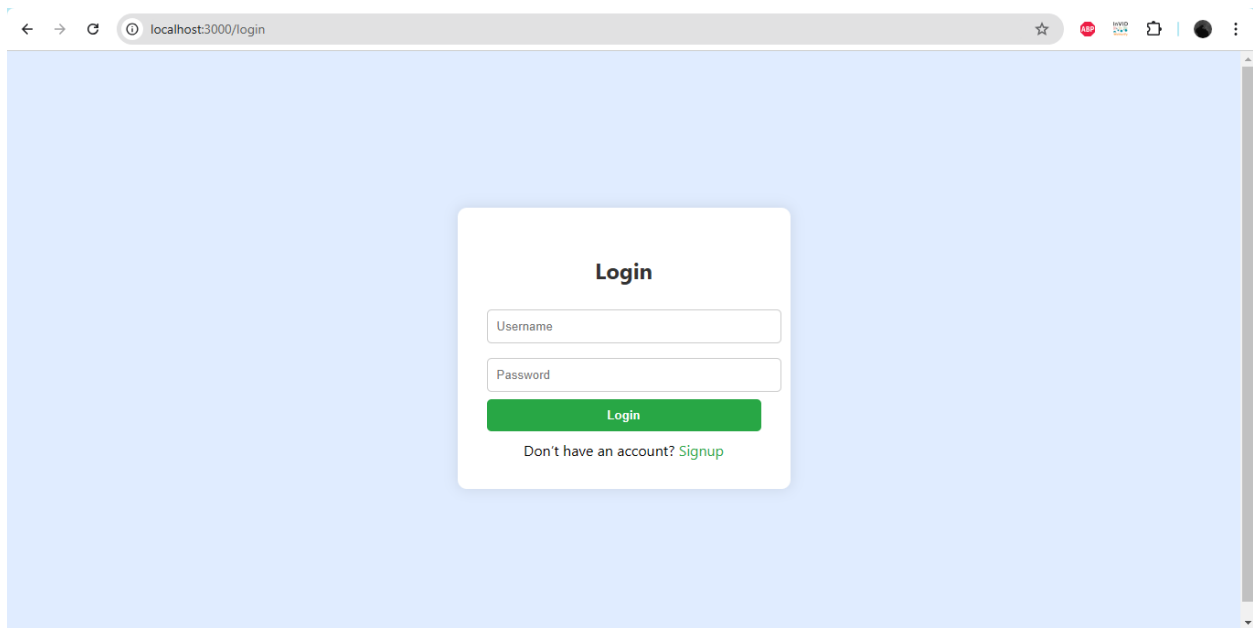
URL: <http://localhost:3000>

Pages Explored

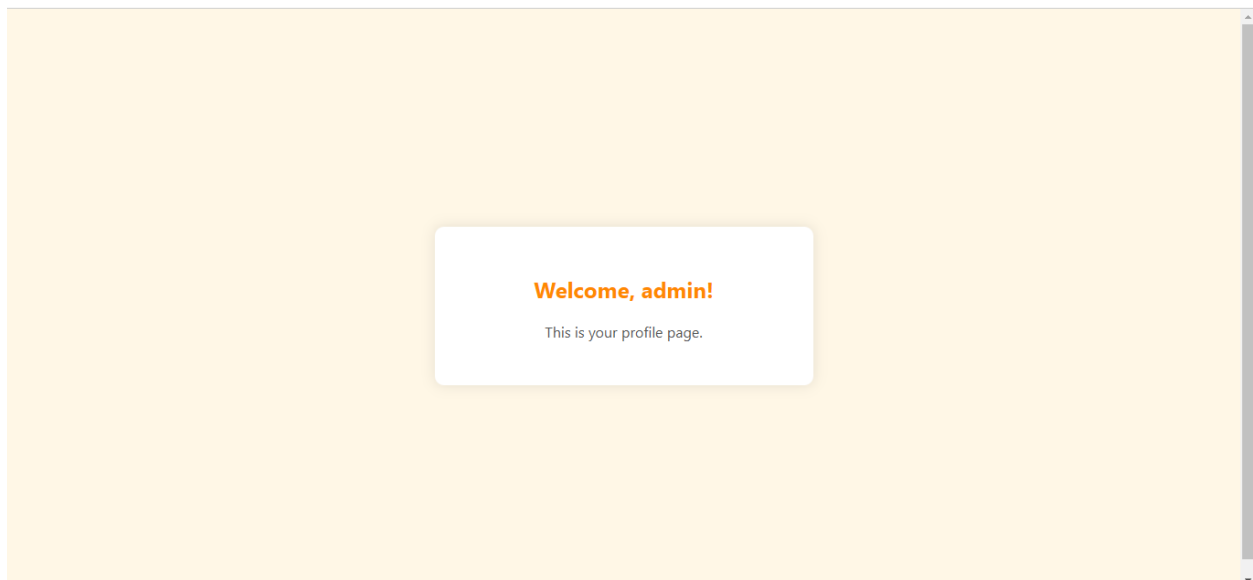
/signup — New user registration

A screenshot of a web application's 'Create Account' page. The page has a light blue background. In the center, there is a white rounded rectangle containing the form. The form has a title 'Create Account' in bold. Below the title are two input fields: 'Username' and 'Password'. Below these fields is a blue button with the text 'Signup'. At the bottom of the form, there is a link that says 'Already have an account? Login'.

/login — User authentication



/profile — Displays user's name post-login



2. Basic Vulnerability Assessment

Tools Used

- **OWASP ZAP** – Automated vulnerability scanning
- **Postman** – For manual injection testing
- **Browser Developer Tools** – Manual XSS simulation
- **MongoDB Compass** – Viewing and verifying database changes

Vulnerabilities Discovered

Cross-Site Scripting (XSS)

Type Reflected XSS

Location /profile?user=

Payload <script>alert('XSS')</script>

Impact Malicious JavaScript executes in user's browser.

Cause User input rendered in HTML without sanitization.

SQL Injection (Authentication Bypass)

Type: SQL Injection

Location: POST /login

Payload:

```
username=' OR '1'='1' --  
password=anything
```

Impact: Bypasses login without valid credentials.

Cause: Unsanitized input directly used in an SQL query.

Example Vulnerable Code (Node.js with MySQL)

```
app.post('/login', (req, res) => {  
  const { username, password } = req.body;  
  const query = `SELECT * FROM users WHERE username = '${username}' AND password = '${password}'`;
```

```
db.query(query, (err, results) => {  
  if (results.length > 0) {  
    res.send('Login successful');  
  } else {  
    res.send('Invalid credentials');  
  }  
});  
});
```

Why It's Vulnerable:

If a user submits:

username = ' OR '1'='1' -- password = anything

The resulting SQL becomes:

```
SELECT * FROM users WHERE username = '' OR '1'='1' -- ' AND password = 'anything'
```

This always returns true due to '1'='1', effectively bypassing authentication.

NoSQL Injection (Authentication Bypass)

Type NoSQL Injection

Location POST /login

Payload username[\$ne]=null and password[\$ne]=null

Impact Bypasses login without valid credentials.

Cause Direct use of req.body in MongoDB findOne() query.

3. Weak Password Storage

Type Insecure Storage

Observation Passwords stored as plain text in MongoDB

Impact If database is leaked, all user credentials are exposed.

Cause No hashing or encryption of passwords.

4. Security Misconfigurations

Observation No HTTP headers for protection (e.g., no CSP, no X-Content-Type)

Tool OWASP ZAP, Browser Dev Tools

Impact Leaves app more exposed to clickjacking, XSS, and MIME attacks.

Areas of Improvement

Area Recommendation

XSS Protection Use escape-html, sanitize input, implement CSP

NoSQL Injection Prevention Validate and sanitize req.body, avoid direct object queries

Password Security Hash passwords using bcrypt before storing

Security Headers Use helmet middleware to set headers like X-XSS-Protection

Input Validation Use input schemas (e.g., with Joi or express-validator)

Logging & Alerts Implement login attempt logging and alerting for abnormal activity

5. Summary

This mock application successfully demonstrated how:

Improper input handling results in XSS and NoSQL injection.

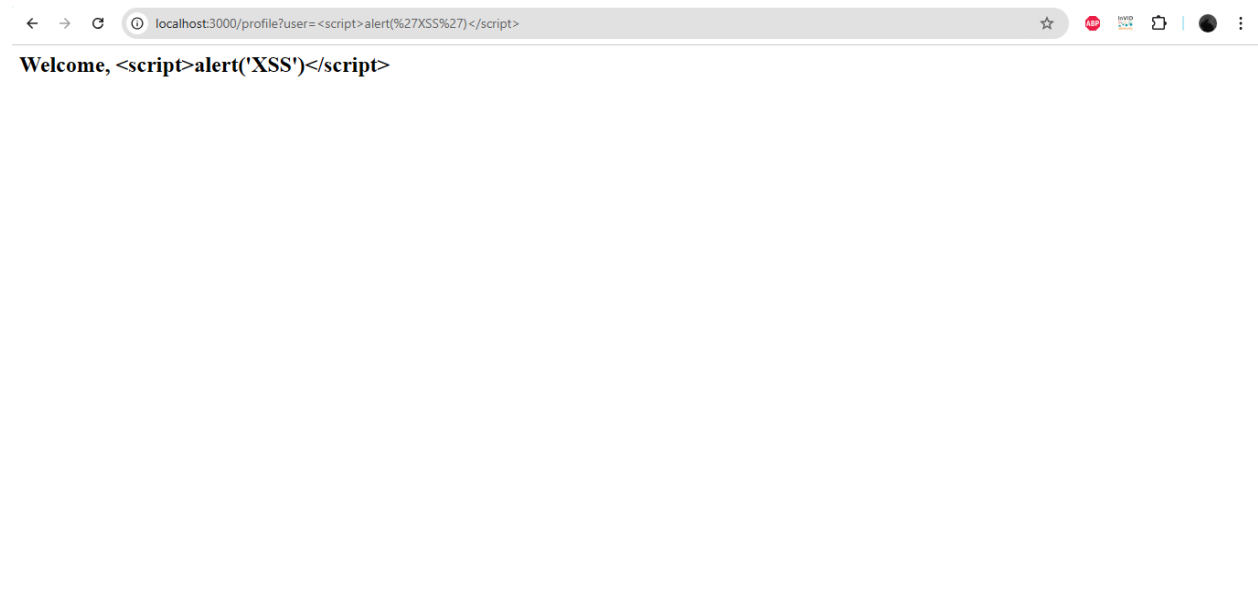
Weak security practices like plaintext password storage and no input validation can be exploited.

Security tools like OWASP ZAP and Postman aid in early detection.

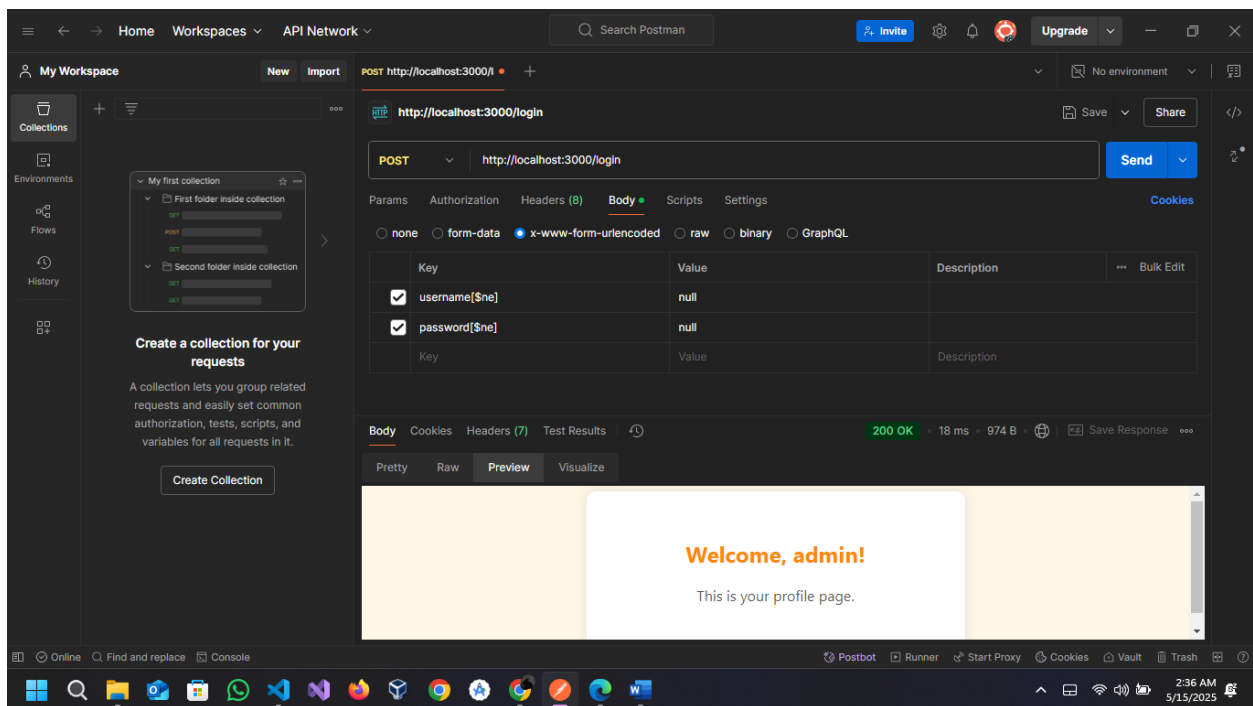
6. Attachments & Evidence

Screenshots:

XSS alert in browser



NoSQL injection login via Postman



Code snippet highlights showing vulnerable logic:

```
const express = require("express");
```

```

const bodyParser = require("body-parser");
const mongoose = require("mongoose");
const app = express();

// View engine
app.set("view engine", "ejs");
app.use(bodyParser.urlencoded({ extended: true }));
app.use(bodyParser.json());

// Connect to MongoDB
mongoose.connect("mongodb://localhost/vulnerableDB", {
  useNewUrlParser: true,
  useUnifiedTopology: true
}).then(() => {
  console.log("MongoDB connected!");
}).catch(err => {
  console.log("MongoDB connection error:", err);
});

// User schema
const userSchema = new mongoose.Schema({
  username: String,
  password: String
});
const User = mongoose.model("User", userSchema);

// Home redirect
app.get("/", (req, res) => res.redirect("/login"));

// Signup route (vulnerable)
app.get("/signup", (req, res) => res.render("signup"));
app.post("/signup", (req, res) => {
  const { username, password } = req.body;
  const newUser = new User({ username, password });
  newUser.save()
    .then(() => res.redirect("/login"))
    .catch(err => {
      console.log(err);
      res.send("Error during signup");
    });
});

// Login route (vulnerable to NoSQL injection)
app.get("/login", (req, res) => res.render("login"));
app.post("/login", async (req, res) => {

```

```

try {
  console.log("Received login attempt:", req.body);

  // Allow full injection (e.g., username[$ne]=null)
  const user = await User.findOne(req.body);

  if (user) {
    // Redirect with raw username (no escaping - vulnerable to XSS)
    res.redirect(`/profile?user=${user.username}`);
  } else {
    res.send("Login failed");
  }
} catch (err) {
  console.log(err);
  res.send("Error during login");
}
});

// Profile route (XSS vulnerability)
app.get("/profile", (req, res) => {
  // NO escaping - vulnerable to XSS
  const username = req.query.user;
  res.render("profile", { username });
});

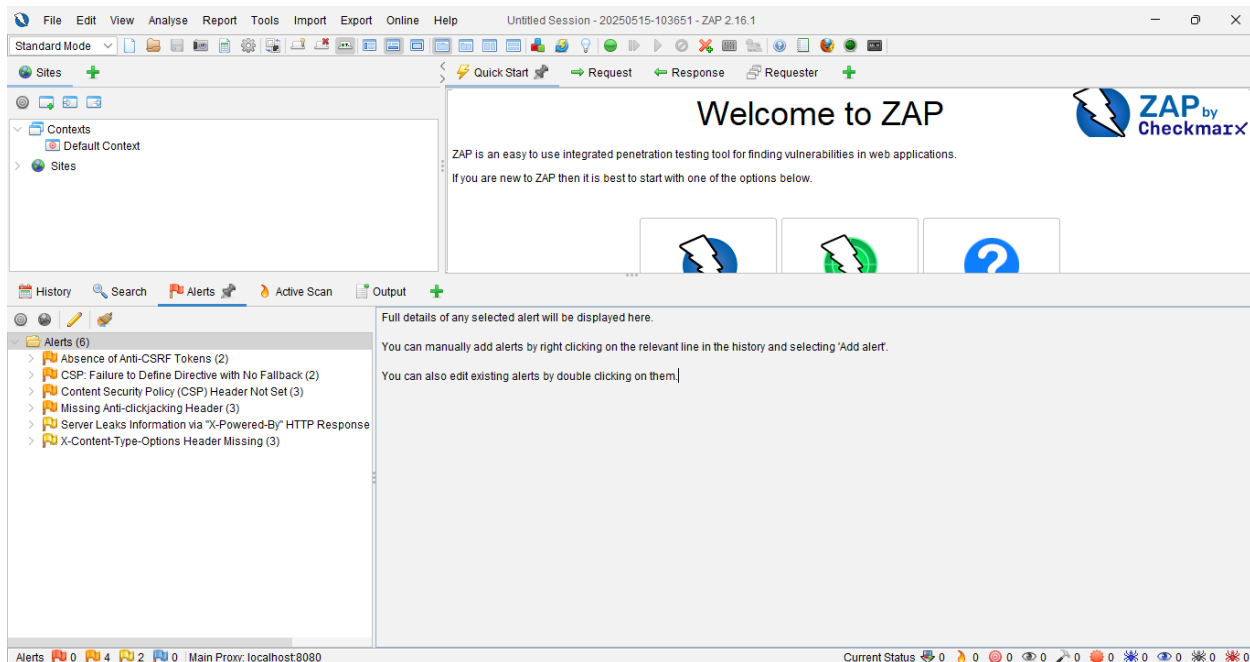
// Start server
app.listen(3000, () => console.log("App running at http://localhost:3000"));

```

```

PS F:\Developer Hub Internship\Week 1\vulnerableweb> node app.js
(node:19952) [MONGODB DRIVER] Warning: useUrlParser is a deprecated option: useUrlParser has no effect since Node.js Driver version 4.0.0 and will be removed in the next major version
(Use `node --trace-warnings ...` to show where the warning was created)
(node:19952) [MONGODB DRIVER] Warning: useUnifiedTopology is a deprecated option: useUnifiedTopology has no effect since Node.js Driver version 4.0.0 and will be removed in the next major version
App running at http://localhost:3000
MongoDB connected!
Received login attempt: { username: 'ZAP', password: 'ZAP' }
Received login attempt: {
  "<?php exec('cmd.exe /C echo jtfy24k5cyu4b945a14h',$colm);echo join("\n",$colm);die();?>": ''
}
Received login attempt: {
  "<?php exec('echo jtfy24k5cyu4b945a14h',$colm);echo join("\n",$colm);die();?>": ''
}
Received login attempt: { 'class.module.classLoader.DefaultAssertionStatus': 'nonsense' }
Received login attempt: { username: "admin' or 1=1 -- -", password: '12345' }

```



These alerts are security warnings generated by a cybersecurity tool (like OWASP ZAP or Burp Suite) during a scan of a web application. They highlight potential vulnerabilities that could be exploited by attackers. Here's what each alert means:

List of Alerts & Their Meanings:

Absence of Anti-CSRF Tokens (2)

The application is missing CSRF (Cross-Site Request Forgery) tokens in 2 requests/pages, making it vulnerable to CSRF attacks.

CSP: Failure to Define Directive with No Fallback (2)

The Content Security Policy (CSP) is improperly configured (missing directives), reducing protection against attacks like XSS.

Content Security Policy (CSP) Header Not Set (3)

The CSP header is missing in 3 instances, leaving the app open to Cross-Site Scripting (XSS) and data injection attacks.

Missing Anti-Clickjacking Header (3)

No X-Frame-Options or similar headers are set, making the app vulnerable to clickjacking attacks.

Server Leaks Information via "X-Powered-By" HTTP Response

The server exposes sensitive details (like backend technology/version) in the X-Powered-By header, aiding attackers in targeted exploits.

X-Content-Type-Options Header Missing (3)

The X-Content-Type-Options: nosniff header is missing in 3 cases, allowing MIME-sniffing attacks (where browsers may misinterpret file types).

Why Are These Alerts Important?

These vulnerabilities can be exploited to steal data, hijack sessions, or take control of user accounts.

Developers should fix them by:

Adding CSRF tokens to forms.

Setting proper security headers (CSP, X-Frame-Options, etc.).

Removing sensitive server information.

Week 2: Implementing Security Measures in Node.js Express Application

Overview

This document outlines the security enhancements made to a previously vulnerable Node.js Express web application that interacts with MongoDB. The focus is on addressing key security vulnerabilities such as NoSQL Injection, Cross-Site Scripting (XSS), password storage vulnerabilities, and insecure HTTP headers.

1. Input Validation and Sanitization

Problem

Previously, the application did not validate or sanitize any inputs. This left it vulnerable to:

- NoSQL Injection via the login form.
- Invalid or malicious input being stored in the database.

Solution

We installed the **validator** library to enforce strict validation of user inputs such as email addresses and usernames.

Implementation

```
npm install validator
```

In the signup and login routes, inputs were validated and sanitized as follows:

```
const validator = require('validator');

if (!validator.isAlphanumeric(username)) {
  return res.status(400).send('Invalid username');
}

if (!validator.isStrongPassword(password)) {
  return res.status(400).send('Weak password');
}
```

2. Password Hashing

Problem

Passwords were previously stored in plain text in the database, making the system highly insecure in case of data leaks.

Solution

We installed and integrated bcrypt to hash passwords before storing them.

Implementation

```
npm install bcrypt
```

```
const bcrypt = require('bcrypt');  
const hashedPassword = await bcrypt.hash(password, 10);  
const newUser = new User({ username, password: hashedPassword });
```

During login, the password is compared securely:

```
const user = await User.findOne({ username });  
if (user && await bcrypt.compare(password, user.password)) {  
  // Successful login  
}
```

3. Token-Based Authentication

Problem

The application used sessions insecurely and had no reliable way to authenticate API users.

Solution

We implemented token-based authentication using **jsonwebtoken**.

Implementation

```
npm install jsonwebtoken
```

```
const jwt = require('jsonwebtoken');
const token = jwt.sign({ id: user._id }, 'your-secret-key', { expiresIn: '1h' });
res.send({ token });
```

The token can then be used for protected routes using middleware:

```
const authMiddleware = (req, res, next) => {
  const token = req.headers.authorization?.split(' ')[1];
  try {
    const decoded = jwt.verify(token, 'your-secret-key');
    req.user = decoded;
    next();
  } catch (err) {
    res.status(401).send("Unauthorized");
  }
};
```

4. Cross-Site Scripting (XSS) Prevention

Problem

Username were rendered on the profile page without escaping, making the app vulnerable to reflected XSS attacks.

Solution

Escape user input before rendering, or sanitize it using a library like **xss**.

Basic Fix

```
<%= username %> <!-- Changes to: -->
<%- username.replace(/</g, "&lt;").replace(/>/g, "&gt;") %>
```


5. Secure HTTP Headers

Problem

HTTP headers were not configured to prevent clickjacking, MIME-sniffing, or enforce secure content policies.

Solution

We installed and configured helmet to secure HTTP headers.

Implementation

```
npm install helmet
```

```
const helmet = require('helmet');  
app.use(helmet());
```

6. NoSQL Injection Protection

Problem

User input was directly passed to `User.findOne(req.body)`, allowing payloads like `{ "username": { "$ne": null } }`.

Solution

Ensure queries are constructed safely, and input is validated strictly.

Fix

```
const { username, password } = req.body;  
const user = await User.findOne({ username });  
if (user && await bcrypt.compare(password, user.password)) {  
  // Authenticated  
}
```

7. Conclusion

By applying input validation, password hashing, token-based authentication, secure headers, and proper rendering, the application is now significantly more secure against common web threats. Further improvements can include:

- Rate limiting
- Logging and monitoring
- HTTPS enforcement

Week 3: Advanced Security and Final Reporting

1. Basic Penetration Testing

Tool Used: Nmap

Command Executed:

```
nmap -sV -p 3000 localhost
```

Scan Result:

```
Starting Nmap 7.95(https://nmap.org ) at 2025-05-16 23:31 Pakistan Standard Time
Nmap scan report for localhost(127.0.0.1)
Host is up(0.0020s latency).
Other addresses for localhost(not scanned): :: 1

PORT      STATE SERVICE VERSION
3000/tcp  open  http    Node.js Express framework

Service detection performed.Please report any incorrect results at https://nmap.org/submit/ .
Nmap done: 1 IP address(1 host up) scanned in 13.18 seconds
```

Interpretation:

- Port 3000 is open and running a Node.js Express application.
- The server is up and responding with low latency.
- No other suspicious services were found running.
- Confirms that only the intended service is exposed — no extra ports or vulnerabilities were revealed during the scan.

2. Set Up Basic Logging

The **winston** library was integrated to capture and store security-related events, such as:

- User signups
- Login attempts
- Error during authentication or server activity

Why Logging is Important

Logging helps in:

- Detecting suspicious activities (e.g. repeated failed logins)
- Debugging and monitoring issues in production
- Auditing user behavior and server events

Implementation Snippet:

```
const winston = require('winston');

const logger = winston.createLogger({
  transports: [
    new winston.transports.Console(),
    new winston.transports.File({ filename: 'security.log' })
  ]
});

logger.info('Application started');
```

Where Logging Was Used:

App startup:

```
logger.info("Application started");
```

Signup route:

```
logger.info(`New user signup attempt: ${username}`);
```

Login route:

```
logger.info(`Login attempt by user: ${username}`);
```

Error handling:

```
logger.error(`Error during login: ${err}`);
```

Output File: security.log

- All logs are written to security.log in the root directory.
- Console output is also visible for real-time monitoring during development.

3. Create a Simple Security Checklist

1. Input Validation

- Basic validation checks are performed on the server side to ensure no empty fields are submitted for username and password.
- NoSQL injection prevention is implemented by ensuring that database queries are built using clean user inputs.
- Future improvement: Use express-validator middleware for more robust and structured validation.

Example:

```
if (!username || !password) {  
  return res.render("signup", { errorMessage: "All fields are required" });  
}
```

2. Password Hashing and Salting

- Passwords are hashed using the **bcrypt** library before being stored in the MongoDB database.
- A salt round of 10 is applied to each password for enhanced security.

Example:

```
const hashedPassword = await bcrypt.hash(password, 10);
```

3. Secure Data Transmission (HTTPS)

- HTTPS is not currently enabled in the local development environment.
- The app is designed to be deployed with HTTPS enabled via Nginx or a cloud hosting service like Heroku, Render, or Vercel.
- HTTPS ensures data like login credentials are encrypted during transmission.

4. Logging

- All login attempts, signups, and application start events are logged using the **winston** library.
- Logs are stored in security.log and are useful for auditing and debugging.

5. Other Security Measures

- Cross-Site Scripting (XSS) is mitigated using EJS's auto-escaping syntax (`<%= username %>`).
- Manual testing confirmed that inputs like `<script>alert(1)</script>` are rendered as plain text and not executed.
- NoSQL injection attacks were tested using payloads such as `{"username": {"$ne": null}}`, and proper sanitization was confirmed.