Université Euro Méditerranéenne Fès

EuroMed University of Fez

Ecole d'Ingénierie Digitale et d'Intelligence Artificielle
(EIDIA)

End of semestre Project

Major: 2nd Year cyber security engineering

Semester: 3

Course: Sécurity Système Exploitation Smartphones

Theme:

Cybersecurity Analysis Project: Identifying and Mitigating Vulnerabilities in a Node.js Task Management Application

Supervised by :

Pr.TMIMI Mehdi

Prepared by :

CHERKAOUI OMARI Mehdi

2025

Introduction

In today's digital landscape, cybersecurity has become a critical aspect of software development and deployment. With the increasing complexity of web applications and the ever-growing sophistication of cyber threats, identifying and addressing vulnerabilities has become a top priority. This report presents a comprehensive analysis of a Node.js-based task management application, developed as a simulation platform to explore common web application vulnerabilities. The application, intentionally designed with security flaws, serves as a learning tool to understand, identify, and mitigate risks associated with web development practices. By systematically analyzing vulnerabilities, this project contributes to building secure, robust, and resilient web applications.

Objective

The primary objective of this project is to perform a thorough cybersecurity analysis of the provided task management application. This involves:

1. Identifying vulnerabilities such as Cross-Site Scripting (XSS), server misconfigurations, insecure data handling, poor error handling, and weak authentication mechanisms.
2. Proposing and implementing practical solutions to mitigate these security risks.
3. Enhancing the application’s reliability and security through secure coding practices, proper configuration, and robust validation techniques.
4. Documenting the entire process to serve as a guide for understanding common security issues and their remediation in web applications.

Methodology

The methodology of this project involves systematically analyzing the task management application for vulnerabilities and implementing remediation strategies. The process is divided into the following steps:

Application Setup and Exploration

- Deploy the Node.js task management application in a controlled environment.
- Explore the application’s features and architecture to understand its structure and functionality.

Threat Model and Vulnerability Identification

- Perform a threat model analysis to identify areas where security breaches are likely to occur.
- Use manual inspection to detect vulnerabilities in the application.

Categorization of Vulnerabilities

- Classify vulnerabilities into categories, including Cross-Site Scripting (XSS), server misconfigurations, insecure data handling, poor error handling, and weak authentication mechanisms.

Impact Assessment

- Evaluate the potential risks and impact of each vulnerability on the confidentiality, integrity, and availability of the application.

Implementation of Security Fixes

- Apply secure coding practices to address the identified vulnerabilities.
- Examples include input validation, secure configuration settings, and proper error handling mechanisms.

Testing and Validation

- Reassess the application to ensure vulnerabilities have been resolved and the fixes are effective.

Documentation and Reporting

- Document the vulnerabilities, mitigation steps, and results achieved.

I. Vulnerability Analysis :

Section 1: Authentication and Authorization Vulnerabilities

1. Weak Password Storage :

Vulnerability:

- Passwords are stored in plaintext in the database.json file without any hashing, making them highly vulnerable if the file is exposed.

Impact:

- If an attacker gains access to the database.json file, they can see all user credentials without needing to crack them.
- Many users reuse passwords across multiple sites. If their password is exposed, attackers can use it to access other accounts.
- Users lose trust in the application if their credentials are leaked.

Solution:

- Use a password hashing algorithm (e.g., bcrypt or Argon2) to securely store passwords.

2. Inadequate Session Security :

Vulnerability:

- The secret used in the express-session middleware is hardcoded (**mysecretkey**), which can be easily guessed.
- The session configuration lacks secure settings like httpOnly, secure, and sameSite flags for cookies.

Impact:

- A predictable secret makes it easier for attackers to craft valid session cookies.
- Lack of secure and httpOnly flags increases the risk of session hijacking or theft via XSS.

Solution:

- Store secrets and sensitive information in environment variables using tools like dotenv.
- Use a strong, randomly generated session secret.

3. Lack of Account Lockout Mechanism:

Vulnerability:

- The "/login" endpoint does not have any protection against brute force or credential stuffing attacks.

Impact:

- Attackers can try multiple username-password combinations to compromise user accounts.

Solution:

- Implement an account lockout mechanism to block login attempts after a certain number of failed attempts.
- We can use rate-limiting middleware (e.g., express-rate-limit) to limit login attempts.

4. Session Fixation :

Vulnerability:

- The same session ID is reused after login, making the application vulnerable to session fixation attacks.

Impact:

- An attacker could set a session ID and trick a user into logging in, allowing the attacker to access the victim's session.

Solution:

- Regenerate the session ID after successful login.

5. Insufficient Authorization Checks :

Vulnerability:

- Session checks are present but inconsistent in certain routes. For example:
- /add and /tasks only check for session presence but do not verify if the logged-in user is authorized to access or modify specific tasks.

Impact:

- A malicious user could manipulate requests (e.g., using Postman) to access or modify other users' data.

Solution:

- Enforce authorization checks for each action by validating that the logged-in user has permission for the requested resource.

II. Section 2: Data Handling Vulnerabilities

1. Lack of Input Validation and Sanitization:

Vulnerability:

- The application does not validate or sanitize user input for routes like /tasks, /add, and /delete. This opens the door to injection attacks, such as SQL Injection, NoSQL Injection, or malicious payloads in task content.

Impact:

- Attackers can inject malicious code into the application, such as:
  - Adding scripts that could lead to **Cross-Site Scripting (XSS)**.
  - Manipulating queries to access or delete unintended data.

Solution:

- Validate input types, lengths, and formats before processing.
- Sanitize inputs to remove potentially malicious characters.

2. Insecure JSON File Handling :

Vulnerability:

- The database.json file is directly read and written without proper locking or error handling.
- There is no restriction on overwriting or corrupting the file.

Impact:

- Concurrent read/write operations can cause data corruption.
- Malicious actors with access to the file system could inject malicious data into the file.

Solution:

- Use atomic file writes to prevent corruption during simultaneous read/write operations.
- Validate the data before writing to ensure its integrity.

3. Exposure of Sensitive Data :

Vulnerability:

- User credentials (even if hashed in future) and task details are stored in a single file, which lacks encryption.

Impact:

- If database.json is accessed, attackers gain access to sensitive user and application data.

Solution:

- Encrypt sensitive data before storing it.
- We can use AES encryption.

III. Section 3: Cross-Site Scripting (XSS) Vulnerabilities

1. Lack of Output Encoding in Templates:

Vulnerability:

- The application directly renders user-supplied data (e.g., task names and descriptions) into HTML without encoding or escaping. This can lead to **Stored XSS** or **Reflected XSS** attacks.
- Example :
  If a malicious user submits a task with a name like **<script> alert('XSS') </script>,** the script will execute whenever this task is rendered.

Impact:

- An attacker can inject malicious JavaScript, which could:
  - Steal cookies or session tokens.
  - Redirect users to phishing sites.
  - Perform actions on behalf of logged-in users (e.g., CSRF attacks).

Solution:

- Use an HTML escaping library to ensure all user-supplied content is safely encoded before rendering.

2. Vulnerable Input Fields :

Vulnerability:

- Input fields like task creation forms (/add) do not sanitize or validate user input before rendering.
- Example: A user submits **<script> alert('XSS') </script>** as a task name, which is then stored in database.json and rendered in subsequent views.

Impact:

- **Stored XSS:** The malicious script persists in the database and executes whenever another user views the affected task.
- **Reflected XSS:** Malicious input is reflected immediately in responses.

Solution:

- Sanitize and validate input on both the client and server side.

Error Handling Vulnerabilities in the Project

Proper error handling is crucial for security. If error messages expose sensitive details about the system, they can help attackers craft targeted attacks.

1. Unhandled Promise Rejections :

Vulnerability:

- If **readData** or **writeData** fails (e.g., due to a corrupted database.json file or permission issues), the error propagates and causes the server to crash or expose internal details.

Impact:

- Revealing file paths or stack traces to attackers.
- Denial of Service (DoS) due to unhandled errors.

Solution:

- Wrap all async calls in a "**try...catch**" block to handle potential rejections

2. **Session Management Errors:**
   Vulnerability:

   - No error handling for “session.destroy()” in the logout route.

   Impact:

   - Incomplete user logouts.

   Solution:

   - Add error handling for “session.destroy()”.

Other vulnerabilities:

- **No CSRF Protection:**
  Vulnerability:
  - The application does not protect against Cross-Site Request Forgery (CSRF) attacks, where an attacker tricks a user into performing unintended actions.

  Impact :

  - Unauthorized Actions: Attackers can perform actions on behalf of the user without their consent (e.g., changing passwords, deleting tasks).

  Solution:

  - Use a CSRF token middleware like **csurf (**to our form**)**.

- **No HTTPS Enforcement:**
  Vulnerability:
  - The application does not enforce HTTPS, which means sensitive data (e.g., passwords, session cookies) is transmitted over unencrypted connections.

  Impact:

  - Man-in-the-middle (MITM) attacks, eavesdropping, and data interception.

  Solution:

  - Use HTTPS in production and enforce it using middleware like **helmet**.

  Helmet adds the following protections:

  - **Prevents X-Powered-By Header Exposure:** Removes the **X-Powered-By** header, which reveals the use of Express.
  - **Enforces Content Security Policy (CSP):** Restricts resources loaded by the app.
  - **Sets HSTS Headers:** Enforces HTTPS connections.

Project structure:

/public

├── login.html

├── register.html

/private/protected

├── index.html

├── dashboard.html

Key achievements of this project include:

1. **Security Enhancements**: Implementation of secure authentication, password hashing, and input validation to mitigate risks and protect user data.
2. **User Interface Development**: Creation of a visually appealing and responsive interface with intuitive functionality for task management.
3. **Task Management System**: Development of a robust system for users to manage tasks effectively, ensuring a seamless and secure experience.
4. **Comprehensive Documentation**: Detailed documentation of identified vulnerabilities, their potential impact, and the corrective measures implemented to address them.

Results achieved:

All vulnerablities was handled, you can find below my Github Repository where you'll find all the codes including the static files and a functional dashboard with task addition and deletion .

Conclusion

This project has been a comprehensive effort to analyze, secure, and enhance a task management application built with Node.js. By conducting a thorough source code analysis, we identified critical vulnerabilities such as insecure authentication mechanisms, risks of SQL/XSS injection, improper session handling, and inadequate input validation. These weaknesses were addressed through the implementation of robust security measures, including strong authentication protocols, secure password storage using **bcrypt**, protection against CSRF and XSS attacks, and improved error handling and logging.

In parallel, we developed a modern and user-friendly interface using HTML, CSS, and Vanilla JavaScript. The static pages **login.html**, **register.html**, **dashboard.html**, and **index.html** were designed to provide a seamless and interactive user experience. The dashboard, in particular, allows authenticated users to add, view, and delete tasks efficiently, leveraging the Fetch API for smooth backend communication.

This project underscored the importance of proactive security practices in web development, emphasizing the need for secure coding, proper validation, and protection against common attack vectors. The lessons learned and the best practices adopted during this project will serve as a foundation for future development efforts, ensuring the creation of secure, reliable, and user-friendly applications.

In conclusion, this project successfully delivered a fully functional, secure, and user-friendly task management application. It not only addressed existing vulnerabilities but also provided a robust framework for future enhancements. By adhering to cybersecurity standards and prioritizing user experience, we have created a solution that balances functionality, security, and usability, setting a strong precedent for future projects in web development and cybersecurity.