Tampereen yliopisto

Aleksi Kirjavainen 153125243

SECURE PROGRAMMING EXERCISE WORK

Informaatioteknologian ja viestinnän tiedekunta
Toukokuu 2025

1. Introduction 3
1.1 General description 3
1.2 User interface description 3
2. Structure of the program 4
3. Secure Programming Solutions 5
3.1 User Authentication 5
3.2 Protected Routes 5
3.3 Secure File Handling 5
3.4 Broken Access Control (OWASP A01) 5
3.5 Input Validation and Sanitization (OWASP A03) 6
3.6 Security Misconfiguration (OWASP A05) 6
3.7 Dependency Vulnerability Scanning (OWASP A06) 6
3.7.1 Vulnerabilities found by Dependency-Check 6
3.8 Static Code Analysis 6
3.8.1 Issued found by SonarCloud 6
4. Observations and Suggestions for Improvement 7
4.1 Known Security Issues 7
4.2 Suggestions for Future Improvement 7

1. Introduction

This project is a Secure File Sharing Web Application designed for uploading, managing, and downloading personal files securely over the web. It focuses on implementing secure programming practices, following guidelines such as the OWASP Top 10.

The program demonstrates secure authentication, file handling, and vulnerability mitigation in a full-stack web application environment using React on the frontend and Node.js with Express on the backend. The purpose of the project is to showcase secure development practices rather than to serve as a production-grade storage system.

1.1 General description

The application allows users to register with a valid email address and a secure password. After registering, users can log in, upload, download, and delete files or view their previously uploaded files.

All operations are protected by authentication mechanisms. Each user can only see, download, or delete their own uploaded files. The session management uses secure HttpOnly cookies to ensure safe and controlled access.

The main goal of the application is to illustrate secure coding principles such as authentication handling, input validation, proper error handling, and security against common web vulnerabilities.

1.2 User interface description

The web interface is designed to be minimalistic and easy to use.

Authentication Pages: Users can register a new account or log in with an existing one. The registration form checks that passwords match and meet minimum length requirements. After a successful login, users are redirected to their personal dashboard.

Dashboard: After logging in, users are welcomed to their dashboard, which displays their email address. The dashboard offers a button to navigate to the file management section and a logout button that safely ends the session.

Files Section: The files section is divided into two parts. Users can either upload a new file or browse their uploaded files. Uploading is done through a simple drag-and-drop area or by selecting a file manually.

My Files Page: Uploaded files are listed in a table showing the filename, file size, and last modified date. Users can download files by clicking on the download link or delete files through a delete button which prompts for confirmation before deletion.

Navigation: Each subpage contains a back button to easily return to the main files page. Protected routes ensure that users must be authenticated to access any content beyond the login and registration forms. Unauthenticated users are automatically redirected back to the login page.

2. Structure of the program

The project is organized into a typical full-stack web application structure, separating the backend server code and the frontend user interface into different folders.

The root folder contains the shared project settings, such as the GitHub Actions workflows for continuous integration. Two main subdirectories are used: server/ for backend logic and ui/ for the frontend client application.

The server/ directory contains all the code required for handling authentication, file uploads, file downloads, and user-specific file management. It also contains middleware functions for security measures such as authentication verification and filename sanitization. The backend is built with Node.js, Express, and TypeScript. Environment variables used by the server, such as the JWT secret, are defined in a .env file placed inside the server/ directory.

The server/src folder includes:

- index.ts which sets up the Express server, connects middleware, and defines the base API routes.
- users.ts which implements registration, login, authentication, and session management functionality.
- upload.ts which handles file uploads, size and type validation, and secure filename storage.
- files.ts which provides secure endpoints for listing, downloading, and deleting user files.
- functions.ts which contains helper functions for authenticating requests and sanitizing filenames.

The ui/ directory contains the React-based frontend. It allows users to interact with the server through a clean and simple interface. It handles the registration and login forms, the dashboard, file upload features, file listing, and the secure access control to private routes. The frontend is written using React with TypeScript.

Protected areas of the application are guarded by the PrivateRoute component, which ensures that only authenticated users can access sensitive views such as the dashboard or file management pages.

Continuous Integration and Continuous Deployment (CI/CD) pipelines are configured using GitHub Actions. The pipelines run static code analysis through SonarCloud and dependency vulnerability checks with OWASP Dependency-Check. Reports are generated and uploaded as artifacts on every push and pull request.

3. Secure Programming Solutions

The project was developed with a strong focus on secure programming practices, following the OWASP Top 10 guidelines. Several protection mechanisms were implemented both on the frontend and backend to minimize common vulnerabilities.

3.1 User Authentication

User authentication is based on email and password credentials. During registration, inputs are validated server-side to ensure proper formatting and strength. Passwords are securely hashed using the bcrypt algorithm before storage, making it infeasible to retrieve original passwords even if the server is compromised.

Sessions are managed through signed JSON Web Tokens (JWTs), which are issued on successful login and stored in HttpOnly cookies. This protects the token from being accessed via JavaScript and mitigates Cross-Site Scripting (XSS) risks. Tokens have a limited validity period of one day.

3.2 Protected Routes

Access to protected areas of the frontend, such as the dashboard and file management pages, is restricted using a PrivateRoute component. The PrivateRoute component checks the user's session state before rendering protected pages.

On the backend, every sensitive API endpoint validates the user's authentication token before processing any requests. If a token is missing or invalid, the server immediately rejects the request. Logging out clears the HttpOnly cookie.

3.3 Secure File Handling

Filenames are sanitized to remove unsafe characters and prevent directory traversal attacks. Only specific file types are allowed for upload, including .png, .jpg, .jpeg, .pdf, and .txt, and each uploaded file is limited to a maximum size of 10 megabytes.

During file download and deletion, filenames are validated to ensure that users can only interact with files they have personally uploaded. This prevents unauthorized file access or modification across users.

Path traversal is explicitly prevented by validating that any file accessed remains strictly within the designated uploads directory.

3.4 Broken Access Control (OWASP A01)

Every action involving user-specific resources, such as listing, uploading, downloading, or deleting files, is protected. Users can only access files associated with their own accounts.

File names are prefixed with the sanitized version of the user's email address, isolating user data and preventing unauthorized file access. Server-side checks are enforced to validate the user's identity based on the token.

3.5 Input Validation and Sanitization (OWASP A03)

User input is validated both on the client and server sides. Server-side validation is performed using the express-validator library to enforce format and strength requirements on registration and login forms.

Uploaded filenames are sanitized to remove any potentially dangerous characters, and only files with known and safe extensions are accepted.

3.6 Security Misconfiguration (OWASP A05)

The application minimizes the risk of security misconfigurations by setting up proper CORS policies. Only requests originating from http://localhost:3000 are accepted during development, preventing unauthorized domains from interacting with the backend.

Additionally, the Express server has been configured to disable the "X-Powered-By" header. This prevents disclosing underlying technologies to attackers.

3.7 Dependency Vulnerability Scanning (OWASP A06)

The project integrates OWASP Dependency-Check into the CI/CD pipeline using GitHub Actions. On every push and pull request, the dependency scanner automatically analyzes both the backend and frontend projects for known vulnerabilities in third-party packages.

The scanner uses an NVD API key to improve vulnerability database access speed and generates an HTML report, which is attached as an artifact in GitHub Actions.

3.7.1 Vulnerabilities found by Dependency-Check

In the server code, Dependency-Check reported that the Express framework has a vulnerability in its response.lists method, but it wasn't used in the project. In the UI code, it found 7 vulnerable packages, that were all dependencies of react-scripts. Since react-scripts was already using the latest version, the only way to fix the vulnerabilities would have been to change to another framework, which would not have been worth the hassle.

3.8 Static Code Analysis

Static code analysis is automated using SonarCloud, integrated into GitHub Actions. The analysis is triggered on every push and pull request and checks the project for bugs, code smells, security vulnerabilities, and maintainability issues. The tool also monitors test coverage.

3.8.1 Issued found by SonarCloud

During evelopment, SonarCloud reported two problems that were fixed.

SonarCloud reported that users could possibly access files outside of the uploads folder by giving a special filename, like ../../etc/passwd. To fix this, a sanitizeFilename function was added to clean up the filenames, and an extra check was added to make sure that the requested file always stays inside the uploads folder. If the path is invalid, the server now blocks the request.

SonarCloud also warned that the Express server was leaking the "X-Powered-By" header, which shows that the server is using Express. This was fixed by calling app.disable("x-powered-by") when setting up the server. Now the server does not reveal its technology in the headers.

4. Observations and Suggestions for Improvement

4.1 Known Security Issues

The major functional parts of the application have been designed to be secure against common vulnerabilities, including injection attacks, broken access control, and security misconfigurations. However, there are some areas where risks remain:

- Uploaded files are saved in plain format without additional encryption.
- In-memory user storage is used instead of a proper database.
- CORS is restricted for development purposes, but for production deployment, further tightening and HTTPS-only enforcement would be necessary.
- Full unit test coverage was not achieved. SonarCloud suggests a minimum of 80% code coverage, but only limited tests were written as examples.

4.2 Suggestions for Future Improvement

If this project were to be expanded further, the following improvements could be made:

- Update unit tests to meet 80% code coverage.
- Integrate antivirus scanner to automatically scan uploaded files for malware.
- Move user management to a real database
- Implement file encryption at the filesystem level,
- Perform a manual security audit and a dynamic penetration test on the deployed application.