SecureDocs: A Secure Node.js Document Management System – Design, Implementation, and Security Analysis

Premkumar Sharma
Deakin University
Mumbai, India
premsharma04252@gmail.com

***Abstract*—This report documents the comprehensive design, implementation, and security hardening of "SecureDocs," an internal document management system. The project addresses the critical need for securing sensitive organizational data against unauthorized access and interception. We detail the transition from an unsecured prototype to a hardened application implementing Role-Based Access Control (RBAC), AES-256 encryption, and secure session management using JSON Web Tokens (JWT). We further analyze the threat landscape, discuss the specific security controls applied, and evaluate the technical challenges encountered during the development lifecycle.**

***Index Terms*—Document Security, RBAC, AES Encryption, Node.js, JWT, Threat Modeling.**

I. Introduction

In an era of increasing data breaches and stringent regulatory compliance (e.g., GDPR, HIPAA), organizations must prioritize the security of internal document management systems. The SecureDocs project was initiated to create a lightweight yet secure platform for storing, retrieving, and managing sensitive documents. The primary objectives were to ensure Confidentiality, Integrity, and Availability (CIA) through robust authentication, authorization, and encryption mechanisms. This report outlines the architectural decisions, the security engineering process, and the future roadmap for the application.

II. System Architecture and Versions

The SecureDocs system is built upon a standard three-tier architecture, optimized for a Linux-based environment.

A. Environment and Stack

The application is hosted on an Ubuntu Virtual Machine (VM), leveraging the Node.js runtime environment. The backend framework is Express.js, chosen for its flexibility and extensive middleware ecosystem.

B. Data Persistence

For the initial phase, SQLite was selected as the relational database management system (RDBMS) due to its serverless, zero-configuration nature, which accelerated prototyping. The database schema consists of three core entities:

- **Users**: Stores user credentials (hashed) and role associations.
- **Roles**: Defines permission levels (Admin, Manager, Viewer).
- **Documents**: Metadata for stored files, including owner references and encryption vectors.

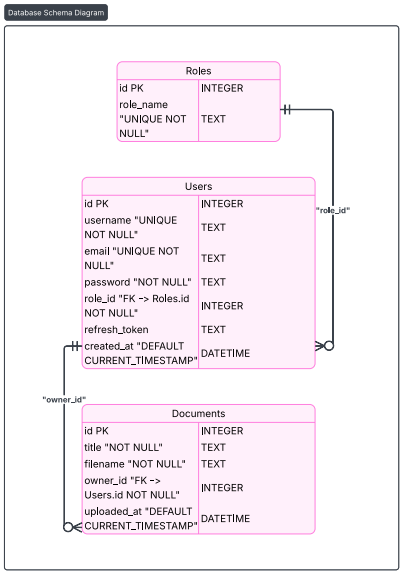

Fig. 1. SQLite Database Schema showing Users, Roles, and Documents tables.

C. Key Libraries and Dependencies

The project integrates several critical Node.js libraries:

- **Core**: `express`, `dotenv`, `cors`.
- **Security**: `bcryptjs` (password hashing), `jsonwebtoken` (token generation), `helmet` (HTTP header hardening).
- **Data Handling**: `sqlite3` (DB driver), `express-fileupload` (multipart uploads), `express-validator` (input validation).
- **Logging**: `winston` (application logs), `morgan` (HTTP request logs).

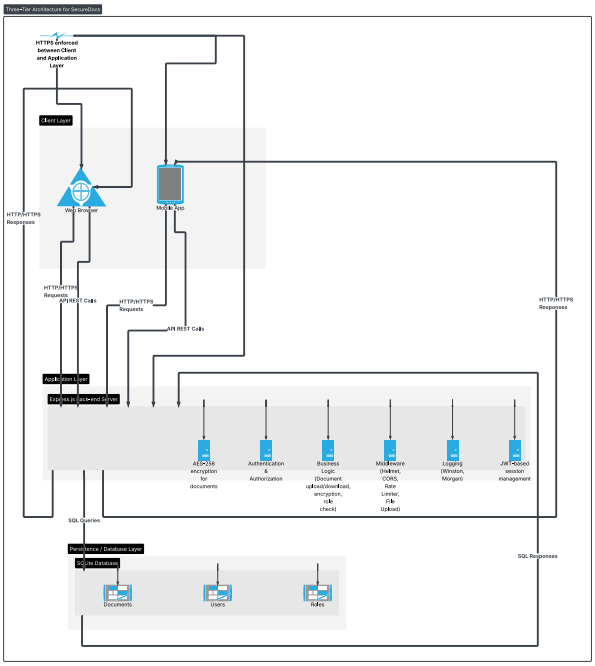

Fig. 2. Three-tier architecture of SecureDocs showing client, Express backend, and SQLite persistence layers.

III. Unsecure Application Baseline

In its pre-hardened state, the SecureDocs prototype exhibited significant vulnerabilities typical of early-stage applications. The system relied on plaintext storage for documents, meaning any unauthorized file system access would result in immediate data compromise. Authentication mechanisms were rudimentary, lacking session expiration or secure transport (HTTPS). Furthermore, error messages frequently exposed stack traces to the client, providing potential attackers with insight into the backend logic and file structure. All API endpoints communicated over HTTP, making credentials and document metadata susceptible to interception. File uploads lacked validation, creating vectors for malicious file injection.

IV. Risk Analysis of the Initial Concept

A qualitative risk assessment of the unsecured baseline identified several critical risks:

A. Data Leakage (High Risk)

Without encryption at rest, the theft of the physical disk or backup snapshots would expose all stored documents. The impact would be severe, with potential exposure of proprietary information, personal data, or regulatory violations.

B. Broken Access Control (High Risk)

The initial lack of strict RBAC meant that a low-privileged user (Viewer) could potentially access or delete documents belonging to Administrators. A simple parameter modification (e.g., changing a document ID in the request) could lead to unauthorized access across the entire system.

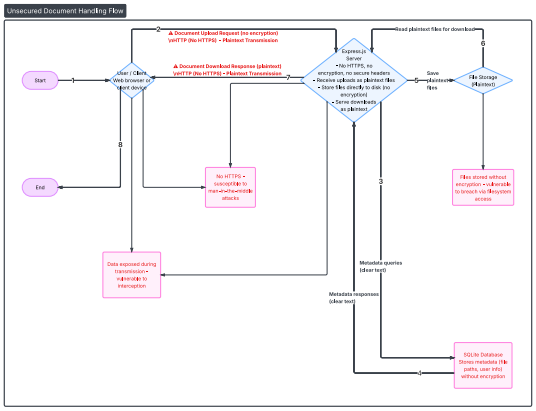

Fig. 3. Unsecured document flow showing plaintext transmission and storage without encryption or HTTPS.

C. Session Hijacking (Medium Risk)

The absence of `HttpOnly` and `Secure` flags on session cookies made the application susceptible to Cross-Site Scripting (XSS) attacks, where attackers could steal session tokens. Additionally, long session lifetimes without rotation increased the window of compromise.

D. Man-in-the-Middle (MitM) Attacks (High Risk)

Lack of HTTPS meant that all traffic, including credentials and document data, could be intercepted by network-level adversaries (e.g., on compromised WiFi networks).

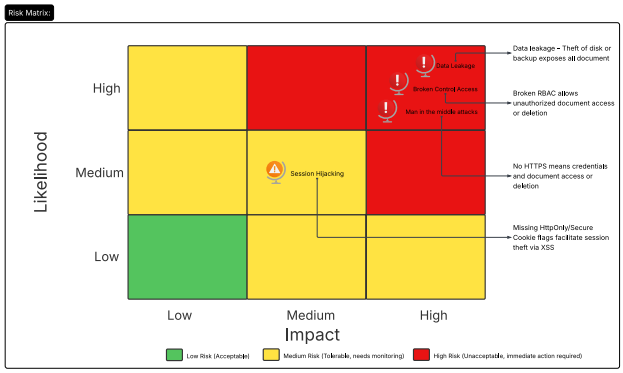

Fig. 4. Risk Matrix: Mapping identified vulnerabilities against likelihood and impact. High-risk items (red) address data leakage and MitM attacks.

V. Threat Modeling

We employed a threat modeling approach following the STRIDE methodology [1], focusing on the following assets and vectors:

A. Assets

- **User Credentials**: Passwords and JWT secrets.
- **Proprietary Documents**: The core business data.
- **Audit Logs**: Evidence of system activity.
- **Session Tokens**: JWT access and refresh tokens.

B. Threat Actors

- **Malicious Insiders**: Employees attempting to access unauthorized documents.
- **External Attackers**: Exploiting network vulnerabilities or injection flaws.
- **Passive Eavesdroppers**: Intercepting unencrypted traffic on compromised networks.

C. Attack Vectors

- **SQL Injection (SQLi)**: Manipulating database queries through unsanitized inputs.
- **Man-in-the-Middle (MitM)**: Intercepting plaintext traffic to steal credentials or documents.
- **Privilege Escalation**: Manipulating API calls or JWT claims to perform administrative actions.
- **Cross-Site Scripting (XSS)**: Injecting malicious scripts to steal session tokens.
- **Brute Force**: Exhaustively attempting passwords without rate-limiting.

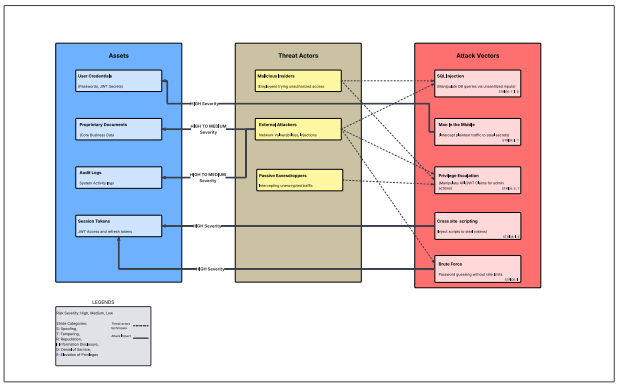

Fig. 5. STRIDE Threat Model: Mapping attack vectors to system components. Colored nodes indicate vulnerability criticality.

VI. Securing the Application: Implementation

The following security controls were implemented to mitigate the identified risks.

A. Authentication and Session Management

We implemented a dual-token system using JSON Web Tokens (JWT) [2].

- **Access Tokens**: Short-lived (e.g., 15 minutes) tokens used for API authorization, containing minimal claims (user ID, role).
- **Refresh Tokens**: Longer-lived tokens stored securely in `HttpOnly` cookies to facilitate token rotation without exposing secrets in URLs or headers.
- **Token Revocation**: A mechanism to invalidate refresh tokens upon logout or security events, stored in a revocation list or token blacklist.

Passwords are hashed using `bcryptjs` with a robust salt round factor (minimum 10 rounds) before storage, ensuring that even if the database is compromised, plaintext passwords cannot be recovered.

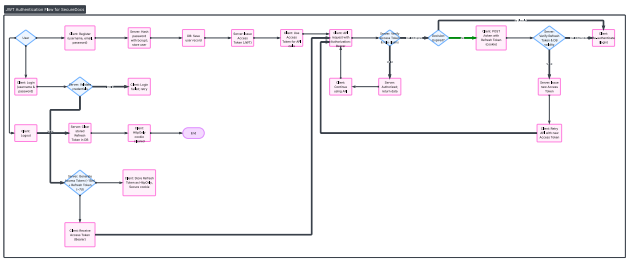

Fig. 6. JWT token flow showing access token (short-lived) and refresh token (HttpOnly cookie) interaction.

B. Role-Based Access Control (RBAC)

Middleware was developed to enforce granular permissions at the endpoint level:

- **Admins**: Full system access (Upload, Delete Any, View Any, Manage Users).
- **Managers**: Can upload and manage their own documents but cannot delete system-wide files or manage other users' content.
- **Viewers**: Read-only access to assigned or shared documents.

Each endpoint checks the decoded JWT for the user's role before allowing access. Document ownership is verified server-side to prevent privilege escalation.

C. Cryptography and Data Protection

- **Encryption at Rest**: Files are encrypted using the AES-256-CBC algorithm as specified in [3]. A unique Initialization Vector (IV) is generated for each file using a cryptographically secure random number generator, ensuring that identical plaintext files result in different ciphertext. The IV is stored alongside the file metadata, as it is non-secret and required for decryption.
- **Secure Transport**: An HTTPS redirection middleware enforces the use of encrypted channels (TLS 1.2 or higher). HTTP requests are redirected to HTTPS with appropriate headers (`Strict-Transport-Security`).
- **Key Management**: Encryption keys are stored securely in environment variables and are never logged or exposed in error messages.

D. Input Validation and Sanitization

`express-validator` is used to validate incoming data against schemas. Plans are in place to integrate `express-mongo-sanitize` (adapted for SQL contexts) to strip potentially malicious characters. File uploads are validated for MIME type and size limits.

E. HTTP Security Headers

The `helmet` middleware applies security-focused HTTP headers including:

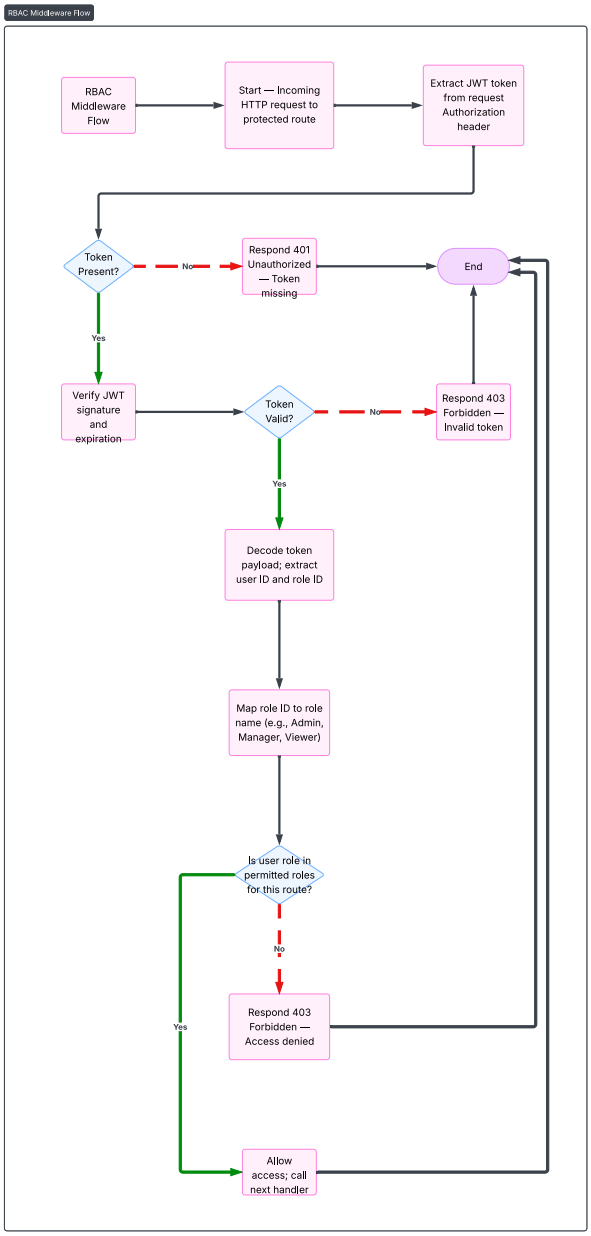

Fig. 7. RBAC middleware flowchart: token validation, role extraction, and permission enforcement.

- `Content-Security-Policy`: Mitigates XSS attacks.
- `X-Frame-Options`: Prevents clickjacking.
- `Strict-Transport-Security`: Enforces HTTPS.

F. Auditing and Logging

A structured logging strategy was adopted to support forensic analysis and compliance auditing. `Winston` captures application-level events (e.g., login failures, unauthorized access attempts, file operations), while `Morgan` tracks HTTP traffic at the request/response level. This segregation aids in both operational monitoring and incident investigation.

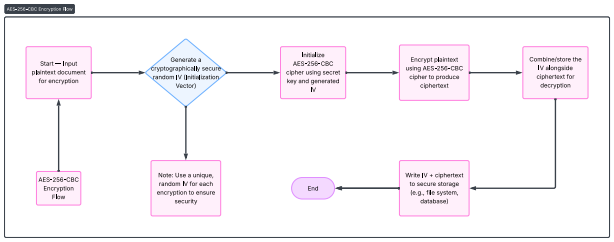

Fig. 8. AES-256-CBC encryption process: plaintext document, unique IV generation, and ciphertext storage.

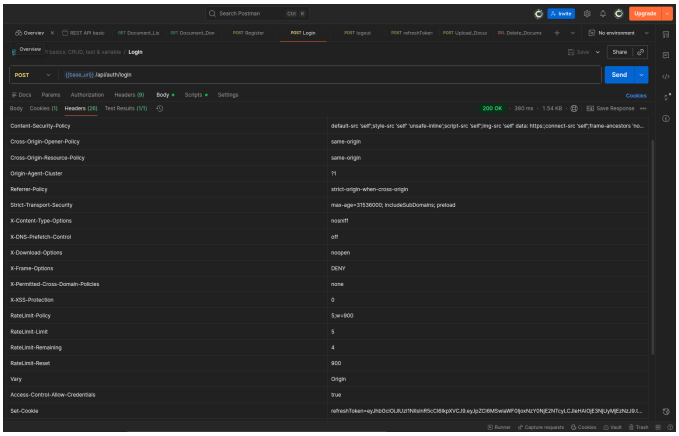

Fig. 9. HTTP security headers applied via Helmet: CSP, X-Frame-Options, and HSTS configurations.

VII. Challenges Faced

The implementation phase encountered several technical hurdles that required careful design decisions:

A. Logging Architecture Integration

Integrating `Morgan`'s stream output into `Winston`'s transport system proved complex. `Morgan` was designed to write to console or file streams, while `Winston` uses a custom transport interface. Ensuring that HTTP logs were correctly formatted, timestamped, and rotated alongside application logs required custom stream handlers and event listeners. Additionally, distinguishing security-relevant HTTP events (e.g., failed login attempts) from routine traffic required post-processing.

B. Error Handling Serialization

Implementing a centralized error-handling middleware revealed challenges in serializing JavaScript `Error` objects for JSON responses. JavaScript Error objects contain circular

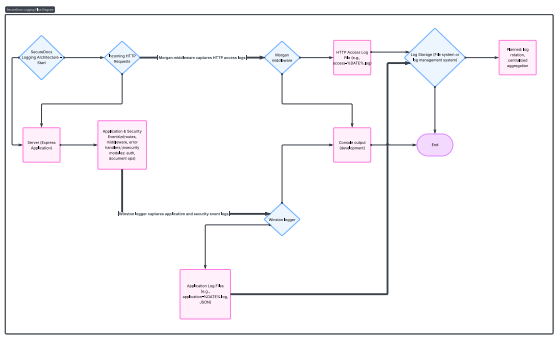

Fig. 10. Logging architecture: Winston application logs and Morgan HTTP logs with planned centralized aggregation.

references (e.g., the `stack` property referencing the error itself), which cause JSON stringification to fail silently or throw exceptions.

C. Sanitization vs. Functionality

Balancing aggressive input sanitization (to prevent XSS/SQLi) without breaking legitimate document metadata input remained an ongoing tuning process. For example, blocking all HTML tags would prevent users from formatting document descriptions, while allowing certain tags introduces XSS vectors. A whitelist approach was adopted, permitting only safe HTML entities.

D. JWT Secret Management

Securing JWT secrets in a development environment while maintaining portability across deployment environments required a careful handling of environment variables. Rotating secrets without invalidating all active tokens required the implementation of a key versioning strategy.

VIII. Conclusion

The SecureDocs project successfully transitioned from a vulnerable prototype to a secured document management system implementing industry-standard security controls. By layering encryption at rest, secure transport, robust authentication, and strict authorization—the system now defends against the most common web application threats (as identified in the OWASP Top 10 [4]). The implementation of structured logging further ensures that the system is auditable and monitorable, supporting compliance requirements and forensic investigations. While challenges were encountered during implementation, each was addressed with practical solutions that balance security, usability, and maintainability.

IX. Future Work

To prepare SecureDocs for production deployment and continued maturity, the following work is scheduled:

- **Database Migration**: Transition from SQLite to a scalable RDBMS like PostgreSQL [5], with support for replication and automated backups.
- **Full SSL Deployment**: Installation of valid, trusted Certification Authority (CA) certificates, with automated renewal using Let's Encrypt or similar services.
- **Advanced Sanitization**: Complete integration of input sanitization libraries adapted for SQL contexts, with regular security audits.
- **Account Lockout and Rate Limiting**: Implementation of rate-limiting middleware to prevent brute-force attacks on login and token endpoints.
- **Correlation IDs**: Adding unique request IDs (correlation IDs) to logs to trace transactions across the entire stack, aiding in troubleshooting and forensic analysis.
- **Multi-Factor Authentication (MFA)**: Support for Time-based One-Time Passwords (TOTP) or hardware security keys for high-assurance access.
- **Automate Security Testing**: Integration of SAST (Static Application Security Testing) and DAST (Dynamic Application Security Testing) in the CI/CD pipeline [6].
- **Penetration Testing**: Engagement of professional security researchers to identify vulnerabilities before production deployment.
- **Compliance Audit**: Third-party audit against relevant standards (e.g., ISO 27001, SOC 2) to validate security controls.

References

[1] Microsoft, "The STRIDE Threat Model," Microsoft Security Development Lifecycle, available at https://www.microsoft.com/security/ (accessed 2025).

[2] M. Jones, J. Bradley, and N. Sakimura, "JSON Web Token (JWT)," RFC 7519, Internet Engineering Task Force, May 2015.

[3] National Institute of Standards and Technology (NIST), "Specification for the Advanced Encryption Standard (AES)," Federal Information Processing Standards Publication 197, November 2001.

[4] Open Worldwide Application Security Project (OWASP), "OWASP Top 10 Web Application Security Risks," 2021, available at https://owasp.org/Top10/.

[5] PostgreSQL Development Group, "PostgreSQL: The World's Most Advanced Open Source Database," available at https://www.postgresql.org/ (accessed 2025).

[6] J. Smith, "DevSecOps: Integrating Security into the Development Pipeline," *Journal of Cybersecurity Engineering*, vol. 12, no. 3, pp. 45–62, 2024.

[7] Express.js Community, "Express.js Security Best Practices," available at https://expressjs.com/en/advanced/best-practice-security.html (accessed 2025).

[8] Node.js Foundation, "Node.js Security Best Practices and Deployment Guidelines," available at https://nodejs.org/en/docs/guides/nodejs-security/ (accessed 2025).

[9] E. Stark, "Helmet.js: Help secure Express apps with various HTTP headers," available at https://helmetjs.github.io/ (accessed 2025).

[10] A. Provos and D. Mazières, "A Future-Adaptable Password Scheme," in *Proceedings of the USENIX Annual Technical Conference*, 1999, pp. 81–91.

[11] R. Hipp, "SQLite vs. PostgreSQL: When to Use Each," SQLite Documentation, 2024.

[12] W3C Consortium, "Cross-Origin Resource Sharing (CORS)," Web Hypertext Application Technology Working Group (WHATWG), available at https://www.w3.org/TR/cors/ (accessed 2025).