Tampereen yliopisto

COMP.SEC.300 SECURE PROGRAMMING
Encrypted note application documentation

TABLE OF CONTENTS

.

1. INTRODUCTION

NoteOnline is a web application for creating and storing personal text notes. It can be used for example quick memos or could work as a password manager system as well. The application features a login system that enables users to access their notes from any device with browser access (provided of course proper login credentials).
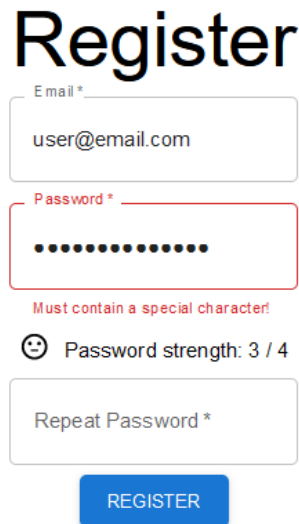
The main point of the application is that the user generated notes are not stored in plaintext, but instead are stored in an encrypted format. Only the user that has created the note, can see its contents. This makes it possible to use the application for storing even more sensitive content, like passwords.

The application is publicly available on GitHub under the MIT licence [1]. Installation instructions and configuration settings are provided in the readme file.

2. LOGIC AND ARCHITECTURE

2.1 Application logic
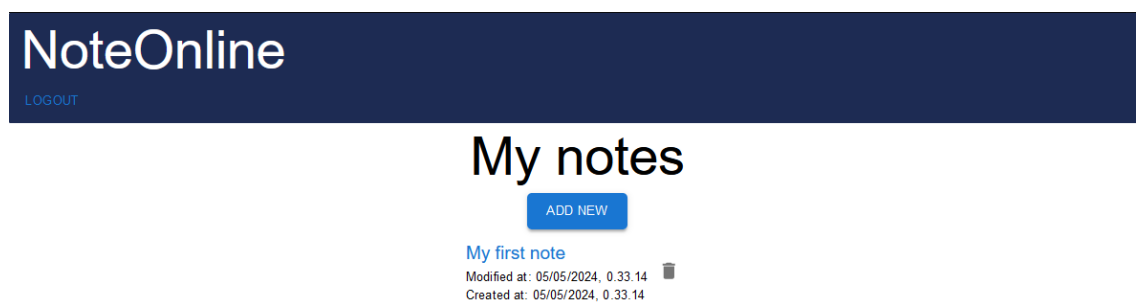
The application is a very simple single page web application. For unregistered users, only accessible points are Login and Register pages. Login is done with an email-password combo. Figure 1 displays the registration view:

Figure 1. *User registration page.*

After a successful registration, a login is required. Once logged in, the notes page is shown. This is illustrated in Figure 2.

Figure 2. *Main page, user notes view.*

New notes can be created from the ADD NEW button. Existing ones can be edited by clicking the name of the existing note. Either option will open the edit view, which is shown in Figure 3.

NoteOnline

LOGOUT

←

Name

My first note

Note content

My first content!

17 / 5000

SAVE

Figure 3. *Note creation/editing view.*

Communication of application state is done using message popups.

2.2 Architecture

The application is divided into a separate frontend and backend entities.

2.2.1 Frontend

Frontend for the application is a web app made with React. React is a component-based web development library [2]. The project components are divided into *Pages* and *Components*. *Pages* are the whole views that the user sees, when navigating to any path (for example "base-address/notes"). *Components* are individual components that form the page contents, like navigation bar, notification popup. Each component handles its own logic.

The frontend also contains a separate state management library Redux [3]. Redux is used to maintain the application state on the user's end. Application state contains five different substates.

- **Auth:** This state holds user email and their roles in the application. For unauthenticated users, the role is set to *ROLE_NONE*. State is used for access control on the frontend: users with *ROLE_NONE* can only access home, login and register pages. Users who have a role (i.e., are logged in), can't access these pages. Auth also includes the session token, which is included in state-modifying requests.

- **Notes**: This state holds the identifier data for each note the user has access to. Identifier data includes note id, owner id, date created, date modified and header of the note. Note content is not included. The state is populated when the user logs in.

- **Active note**: This state contains all information of one note, including decrypted contents of the note. This state is populated whenever the user opens a note on the application.

- **Notification**: This state is used to display notifications for the user, which can either be received from the server, or the frontend app. This state consists of the state type (*SUCCESS*, *ERROR*) and message text.

- **AppState**: This state describes the application responsiveness state. Can be either *LOADING* or *READY*. State is mainly used to disable user interaction when application is loading content and displaying a load bar.

Communication with the server is done using thunks, which are asynchronous methods executing multiple actions. Logic for these can be found in *src/store/actionCreators/thunks*. Thunks invoke actions using the dispatch method. Actions are in *src/store/actionCreators*. Based on dispatched actions, state is modified through reducers (located in *src/store/reducers*).

User interface is using Material UI component library to provide more stylized elements compared to default HTML elements [4].

2.2.2 Backend

Backend of the application is made with Spring Boot in Java [5]. In addition, the following libraries are used:

- Spring Security: authentication and access control [6].

- jwt-api: user login and credentials management [7].

- BouncyCastle: Hashing and key generation [8].

- PostgreSQL: Database [9].

The backend is separated into models, controllers, and services. *Models* represent the data and relations between them. These also act as schemas for the database. *Control-*

lers handle incoming HTTP requests and return appropriate responses. Controllers invoke validations and other business logic. *Services* handle more complex business logic, like file encryption. Note encryption is done in *NoteService.java.*

Security and other configurations are in the config folder. The core of the security pipeline is *SecurityConfig.java*, which defines a filter chain for accepting requests. *JwtUtils.java* handle logic related to Jwt tokens. *PasswordEncoder.java* includes configuration for the password algorithm. *AuthEventListener.java* is used for listening failed login events and locking user accounts with too many failed attempts.

Utils directory contains general business logic for validation and encryption. *EncryptionUtils.java* includes encryption methods and algorithms. *LoginUtils.java* includes login validation logic and generating encryption keys and tokens. *NoteUtils.java* validate incoming note content.

DTOs are used for transferring data between server and client. They form stripped down versions of the database models: user passwords, salts, IVs and such are not needed on the frontend, despite being stored in the database.

Program static configuration file can be found in resources/application.properties. This configuration file contains database connection credentials, Jwt settings and login limits.

3. SECURITY SOLUTIONS

3.1 Password policy

Strong password policies are enforced in the application. Minimum of 10 characters, maximum of 64 character. Passwords must contain one uppercase letter, one lowercase letter, a number and special character. These rules are checked client- and server-side. Password upper limit exists to prevent long password DoS issues.

The application also includes a password strength meter. This was made using *zxcvbn-ts* library [10] which is recommended by OWASP [11]. This library can detect the use of common phrases like *password*, *qwerty* and will also detect patterns like *1234554321* and *qazwsxedc*. Scoring is used to determine the strength of the password: password must have strength score 3 or higher to be accepted, in addition to the roles mentioned earlier.

3.2 User access control

A rate limiter has been implemented for the login system. After five failed login attempts, a one-minute cooldown is added, where the user credentials are not check with a login request. After 10 failed attempts, the cooldown time is one hour for each failed request. The counter will reset when a successful login is made. This is done to prevent brute force attacks against known user accounts.

Login system does not provide information on what goes wrong with a failed login. The response is the same whether the email is wrong, password is wrong, or login limit has been reached. This was implemented to ensure that attackers cannot check which emails are in use in the application. On the other hand, this could be a usability issue in the sense that account lock is not explained to a regular user.

User authentication after a successful login is handled through a JWT token. JWT tokens (JSON Web Tokens) are a standardized method of transferring claims between two parties [12]. JWT token is generated from user email and password when logging in. Subsequent requests made to the server have the JWT token included in them. The server checks the validity of the token when a request to a protected endpoint is made (protected endpoints are /user and /notes).

The JWT token is stored as an HTTP only cookie on the client. This means that the cookie is not accessible from JavaScript. This will help protect against XSS (Cross Site

Scripting) attacks [13]. However, this approach has the risk of enabling a CSRF (Cross Site Request Forgery) attack.

CSRF (Cross Site Request Forgery) is an attack, where a malicious entity can trick an authenticated user to perform unwanted actions on a trusted site [14]. A separate session token (X-CSRF-TOKEN) is used to protect against CSRF attacks. This token is a 32 byte cryptographically random sequence, which is generated when a user logs in. Unlike JWT tokens, this token is stored in browser state. Thus, it gets refreshed each time the page is reloaded. The token should be included in each state-modifying request made to the server. The server does not accept requests with empty or invalid session tokens.

CSRF attacks are mitigated by a few additional factors. Firstly, no modifications are done with safe methods (GET, HEAD, OPTIONS). Updates are done with POST, PUT and DELETE. This eliminates the possibility to click state-modifying links in the application. In addition, all cookies in the application (JWT token and encryption key) have their SameSite attribute set to 'Strict'. This attribute disables sending cookies to cross-site requests [13]. CORS settings on the server also only allows the origin of the frontend. The application does not use any client-side redirects. The user is only ever able to see their own content, so there is also no risk of other users generating malicious inputs.

Whenever a user enters the page, a request is made to /authstatus endpoint. This state checks the validity of the JWT token and encryption key from the request. If these are empty or invalid, user is required to log in. If tokens are still valid, they are refreshed, and a new session token is granted. Figure 4 illustrates the refresh process.

**Check status
(Refresh or re-enter web page)**
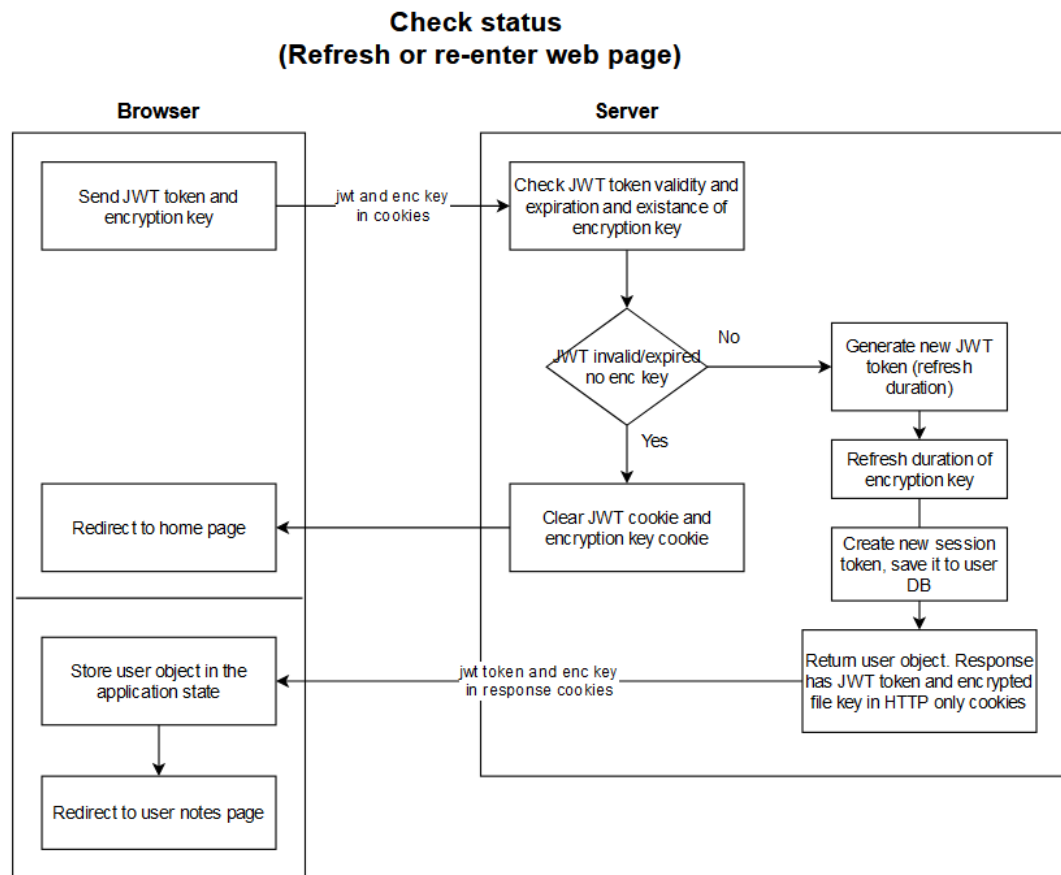
Figure 4. *Actions when user re-enters the page.*

This status refresh logic means that users need not to re-login each time they refresh or re-enter the page unless their tokens are expired.

3.3 Note encryption

Application users are granted a personal master encryption key upon a successful login. This key is derived from the user password, using argon2. Argon2 is a key derivation function that is currently OWASPs top recommended method for storing passwords [15]. Similarly to the JWT token, the encryption key is also stored as an HTTP only cookie on the client. User login flow is shown in Figure 5.
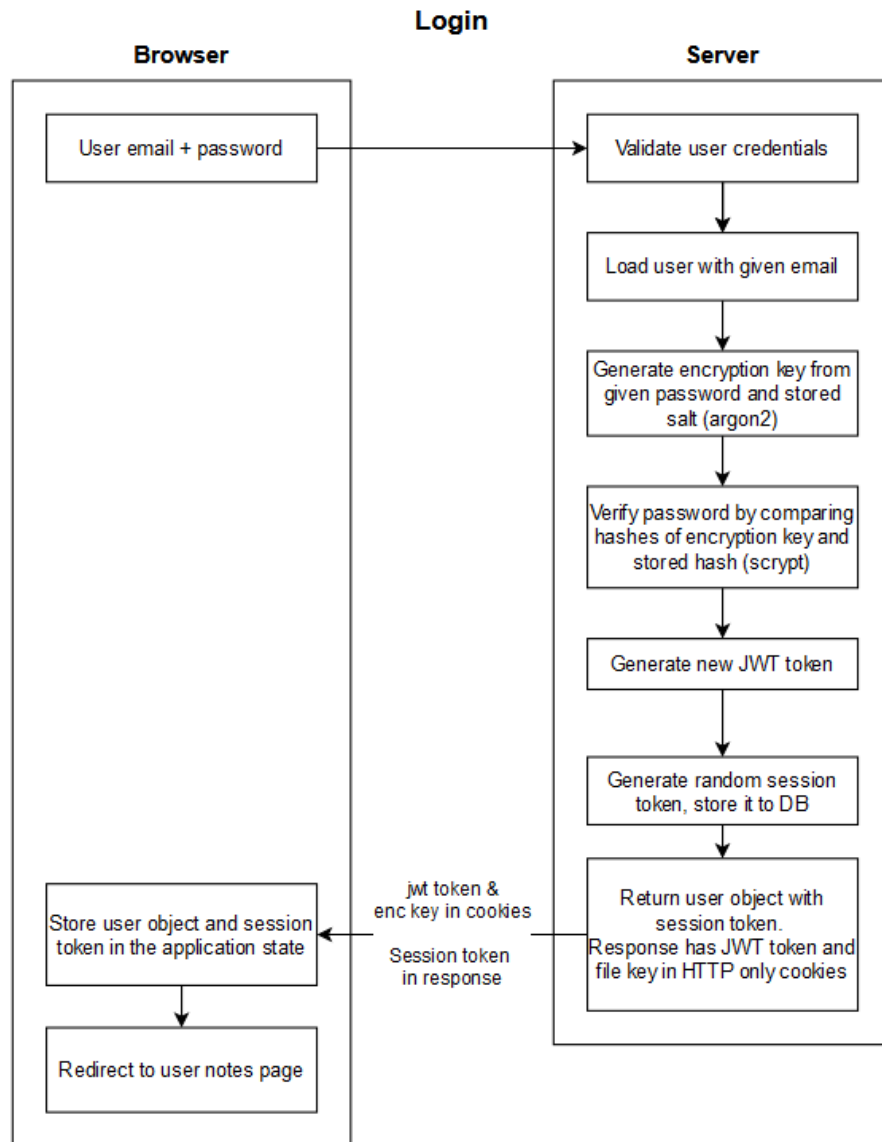
Login

Figure 5. *User login flow.*

File encryption is done using AES-CBC, with a crypto-random initialization vector and salt. For each file, a separate encryption key is generated. This key is used to encrypt the file contents. Then this file key is stored by encrypting it with the user's master encryption key. Figure 6 illustrates the note encryption process. Figure 7 illustrates note decryption process.

Create note / update note

Browser

Server

Unencrypted note data (header, content, owner)

— Encryption key in cookie →

Generate crypto-random file key (16 bytes) + iv (16 bytes)

↓

Encrypt note header/content text with file key (AES-CBC)

↓

Generate "file key encryption key" from user provided encryption key and random salt (PBKDF2WithHmacSHA256)

↓

Encrypt file key with "file key encrypion key" (AES-CBC)

↓

Save encrypted header/content, encrypted file key, salt and iv to DB

↓

Store response as active note in application state

← Note object in response body —

Respond with plain text note

Figure 6. *Note creation/update process.*

Read note

Browser

Server

Request note with id

— Encryption key in cookie →

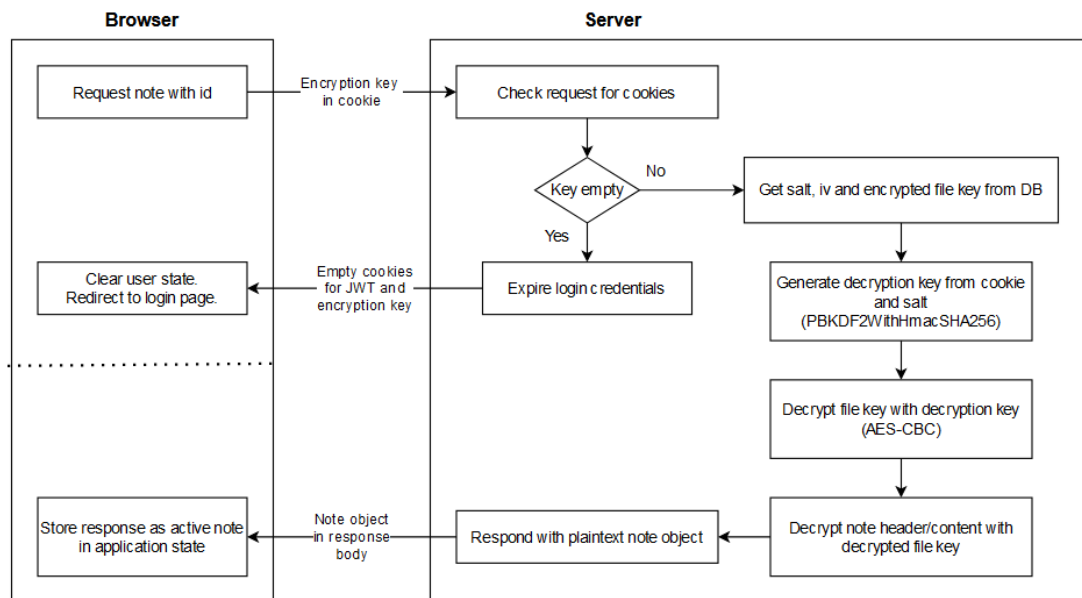Check request for cookies

↓

Key empty — No → Get salt, iv and encrypted file key from DB

↓ Yes

Clear user state. Redirect to login page.

← Empty cookies for JWT and encryption key —

Expire login credentials

Generate decryption key from cookie and salt (PBKDF2WithHmacSHA256)

↓

Decrypt file key with decryption key (AES-CBC)

↓

Store response as active note in application state

← Note object in response body —

Respond with plaintext note object

←

Decrypt note header/content with decrypted file key

Figure 7. *Note decryption process, user accessing a note in the application.*

Using separate file keys makes it possible for users to change their password later: when a password is changed, only the file encryption keys need to be re-encrypted.

Note content size is limited to 5000 characters. Note header/filename is limited to 64 characters. This is to prevent excessively large files from being added to the database.

3.4 Frontend

React uses a virtual DOM for rendering the HTML. The frontend elements are created using JSX, which is a syntax extension for JavaScript. This means that raw HTML is not generated by the user at any point in the application. JSX outputs elements using auto-escaping, meaning that anything, for example embedded script tags for elements, would be treated as regular strings. [16] Escaping user input helps prevent cross-site-scripting attacks (XSS), illustrated in Figure 8.

Figure 8. Embedding script to input fields does not execute script.

Since the users can only see their own content, there is no risk of clicking malicious links or embedded content on the site.

3.5 OWASP Top 10

OWASP Top 10 is an awareness document for web application security. It describes the most critical security risks found in web applications [17]. This section highlights the latest 2021 listing and mentions strategies this project has taken to avoid these risks.

Broken access control

- Application contains role-based access control. On frontend, unauthenticated users can only access home, login and register pages. Authentication credentials

are checked for every request that contains note data. In addition, application only returns notes that have their owner id equal to the users id.

- All added users are always defaulted to 'USER' role. Admin roles can only be assigned through the database. Therefore, the risk of elevated privileges is minimal.

Cryptographic failures

- Application uses currently secure encryption and hashing algorithms: argon2 for key generation, scrypt for password hashing and AES-CBC for encrypting notes. Initialization vectors and salts are generated using cryptographically random byte generators (using Java's SecureRandom number generator) [18].

Injection

- User email must be in a specific format (checked both on front and backend). Passwords are hashed before contacting the database, making malicious SQL code injection practically impossible.

- Raw SQL queries are not performed on the backend. Communication with the database is done through an ORM. In addition, user specified data includes only the filename and content of a note. Both are encrypted before making contact with the database, which eliminates the possibility to inject SQL queries.

Insecure design

- Data is always validated on both frontend and backend.

- Some limits for number of login attempts and note content are implemented.

Security misconfiguration

- The application uses latest stable versions of Spring Boot & Spring Security (backend) and React (frontend).

- No default admin accounts are present in the application. Database uses a secure password.

- CORS policy only allows requests from the frontend address.

Vulnerable and outdated components

- The project was started with the latest stable versions of all technologies that were used.

- Spring Boot uses version 3.2.3. Reported vulnerabilities for Spring Boot have only affected older versions of the framework. Spring security has reported vulnerabilities, but these are for features that are not used in this application. [19]

- React version 18.2.0 had no reported vulnerabilities [20]. Redux version 9.1.0 had no reported vulnerabilities [21]. axios had reported vulnerabilities for older versions than the one in use (1.6.7) [22].

- When installing the frontend, npm will report vulnerabilities with the packages. This is discussed in Chapter 5.2.

Identification and authentication failures

- A rate limit for login is implemented, making brute-force attacks more time consuming. After 10 failed attempts, the attacker would need to wait 1 hour after each failed attempt.

- Application uses currently secure hashing and encryption algorithms.

Software and data integrity failures

- For project management, the application uses Maven on the backend and npm on the frontend. External packages that are managed through these systems have been manually verified to be correct. The application does not use CDNs or other external sources to fetch data from, nor does it have any automatic update/version checking.

Security logging and monitoring failures

- Application logging is very minimal. Spring Boot allows more detailed logging by using a debug mode logger but this is turned off in the application.

Server-side request forgery

- Only defined API routes are recognized by the backend system (routes are defined in backend as controllers). User input is never used as instructions to do anything (like fetching content from a user specified URL). All text input the user can feed the server is either validated with a regex (email) or hashed/encrypted.

4. TESTING

Manual testing has been done extensively throughout the development of the project. User registration and note creation in particular have been under constant iterations. For example, a late bug was discovered where the application would not accept note content longer than 175 characters due to a wrong data type being used in the database.

CSRF testing was performed by removing required headers from the requests and attempting to do state-modifying actions. All such actions returned a 401 response by doing as a result. Figure 9 shows the result of attempting to delete a note with a logged in user, without a session token.
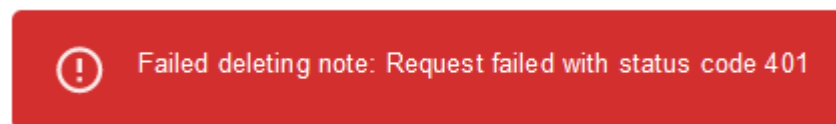

Figure 9. *Attempting to delete a note without a session token.*

Limit testing was done for password inputs, file name inputs and note content inputs. It was verified that checks are working on the frontend: requests are not sent if data is invalid. Backend checks were tested by removing the limits from the frontend and resending the requests. The requests failed as well as expected. Some basic script embedding was also attempted (see Figure 8).

Account login cooldown was tested by sending continuous requests with a valid email, but with an incorrect password. The server started rejecting all login attempts after five failures. The cooldown was set to one minute. After the cooldown period, giving proper credentials allowed a login.

Various passwords combinations were tested to see if the password strength checker recognizes commonly used passwords. A common password list was used to help with this [23]. In addition to directly using common passwords, some variations were tried with replacing letters with numbers, capitalizing letters and adding numbers. Figure 10 displays an example of a typical response.
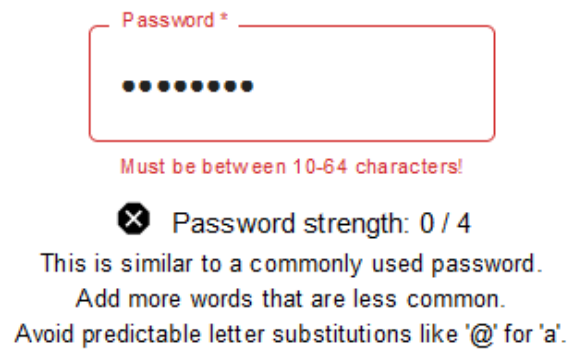
Figure 10.   *Testing password strength with input "Passw0rd"*

Some other bad practices like patterns and repetition were also used in password testing. The checker also caught some of these but failed for example with a pattern *qazokmwsxijn* (letters below each other from left to right). Using common names would also trigger a warning.

5. KNOWN LIMITATIONS AND SECURITY ISSUES

5.1 CSRF concerns

The application uses per-session tokens for CSRF prevention. Per-request tokens should be preferred over this choice since there is less time for an attacker to exploit a stolen token [24]. Additionally, the system implemented here is custom made, which should always be avoided when an existing solution already exists. Spring Security offers built-in protections against CSRF attacks [25]. However, this configuration is turned off in the application. The reason for this is that the application had issues with validating the CSRF token, causing numerous failed requests when updating notes. Unfortunately, I did not have enough time to figure out this issue, which is why per-session tokens were used instead. Fixing this issue would be a high priority before the application publicly.

5.2 npm reports vulnerabilities

When installing the frontend with npm, the installation will report 2 moderate and 6 high vulnerabilities. Vulnerabilities reported are Inefficient Regular Expression Complexity and *PostCSS line return parsing error*. These issues originate from react-scripts and are due to using the Create React App build tool.

It is likely that these issues are false positives that are related to how npm audit works. Create React App is a build tool, so it does not create a running Node application. The issues in the build tool would not transfer to the published application, as the published application wouldn't include these tools. [26] As the issues are related to nth-check and css generation, this is not a security issue in the deployed application, as neither generates any content at runtime. Furthermore, the suggested fix in npm is to downgrade to a much older version, which would most likely be a bigger security issue.

5.3 Time based attacks

Currently there is a clear time difference for fail methods when logging in. If the account is locked, the server response is immediate. For an invalid email, time is increased because of the need to query the database. If the email is valid, login time is increased as the password must be checked. While the application does not expose possible account data with its responses, an attacker could make some guesses based on the processing time. For example, try the same email six times: if the sixth response comes immediately,

the email is in use in the application since it was just locked. An arbitrary delay should be added on server side to fix this issue.

5.4   HTTPs over HTTP

Currently the application uses unencrypted HTTP over HTTPs. This means that all data sent between the client and server is unencrypted. This is a concern when sending passwords and unencrypted notes in requests, as transferred data is easily obtainable via a man-in-the-middle attack. HTTPs support was left out due to time constraints. Proper HTTPs encryption should be added for a publicly available version of the application.

5.5   Lack of logging and monitoring

Unnecessary logging can be a security issue. As was mentioned in chapter 3, the application does not generate explicit log information. However, logging and monitoring features could also be used to detect for example DDoS attacks and brute force login attempts [27]. Some sort of automatic alert system would be a solid addition to the application. Some request IP blocking solutions were investigated for the application but did not make it to this project on time.

5.6   User registration and password recovery

There is no verification of ownership in emails. Meaning that users do not receive verification emails that an account with their name has been registered with the service. This means that you can currently add a non-existing email address to the application, if the format is valid. Emails are used similarly to unique usernames in other applications. The decision to use emails instead of usernames was made because this ensures the possibility for creating a password reset service.

Current architecture of the application supports the possibility to change the password. As encryption is based on encrypting file keys, only the file keys need to be re-encrypted when a password changes. However, the current architecture does not support a password recovery service. Since file keys are encrypted using a key derived from the password, there is no way to decrypt notes when the password is lost. This is not a security vulnerability but is an issue of data availability and user experience.

5.7   Admin role

The application support both admin and user roles. This is because the backend authentication framework (Spring Security) is a role-based authentication system. Admin accounts can see note identifier data (name, date created, owner) but do not have access to the actual note content. Admins can also remove and modify user data. Currently there is no way to register admins, other than manually adding them to the database. Therefore, there is no risk of elevated privilege, unless the attacker would have editing rights to the database. In which case, whether the attacker has admin rights or not, would not matter. However, the admin role is pointless in practice, as all its current operations can be done via database queries in PostgreSQL. Dormant functionality should be minimized, as it provides increased attack surface for malicious actors and reduces maintainability.

6. FURTHER DEVELOPMENT

Like mentioned in Chapter 5.6, there is currently no way to recover an account if a password is forgotten. Some method of account recovery should be added if the application was to be made publicly available. Further improvement could be to enable logging in using for example the user's Google account using OAuth.

Currently the application only supports text files. This could relatively easily be extended to support other data types, like images. Basically, any content that is feasible to store in a byte array format (and is relatively small in size) could be encrypted with this application.

REFERENCES

[1]   NoteOnline source code, 2024. Available: https://github.com/Polystyreeni/Note-Online

[2]   React, 2024: Available: https://react.dev/ (Referenced 5.5.2024)

[3]   React Redux, 2024. Available: https://react-redux.js.org/ (Referenced 5.5.2024)

[4]   MaterialUI, 2024. Available: https://mui.com/ (Referenced 5.5.2024)

[5]   Spring Boot, 2024. Available: https://spring.io/projects/spring-boot (Referenced 5.5.2024)

[6]   Spring Security, 2024. Available: https://spring.io/projects/spring-security (Referenced 5.5.2024)

[7]   JWT: Libraries for Token Signing/Verification, 2024. Available: https://jwt.io/libraries (Referenced 5.5.2024)

[8]   Legion of the Bouncy Castle. Available: https://www.bouncycastle.org/java.html (Referenced 5.5.2024)

[9]   PostgreSQL JDBC Driver, Maven Repository, 2024. Available: https://mvnrepository.com/artifact/org.postgresql/postgresql (Referenced 5.5.2024)

[10]   zxcvbn-ts, 2024. Available: https://zxcvbn-ts.github.io/zxcvbn/ (Referenced 5.5.2024)

[11]   OWASP Authentication Cheat Sheet, 2024. Available: https://cheatsheet-series.owasp.org/cheatsheets/Authentication_Cheat_Sheet.html (Referenced 5.5.2024).

[12]   M. Jones, Microsoft, J. Bradley, Ping Identity, N. Sakimura, NRI, JSON Web Token (JWT), Internet Engineering Task Force (IETF), 2015. Available: https://datatracker.ietf.org/doc/html/rfc7519 (Referenced 5.5.2024)

[13]   MDN Web Docs, Using HTTP cookies, 2024. Available: https://developer.mozilla.org/en-US/docs/Web/HTTP/Cookies (Referenced 5.5.2024)

[14]   OWASP, Cross Site Request Forgery (CSRF). Available: https://owasp.org/www-community/attacks/csrf (Referenced 5.5.2024)

[15]   OWASP, Password Storage Cheat Sheet. Available: https://cheatsheet-series.owasp.org/cheatsheets/Password_Storage_Cheat_Sheet.html (Referenced 5.5.2024)

[16]   React, Introducing JSX, 2024. https://legacy.reactjs.org/docs/introducing-jsx.html#jsx-prevents-injection-attacks (Referenced 5.5.2024)

[17]   OWASP Top Ten, 2021. Available: https://owasp.org/www-project-top-ten/ (Referenced 5.5.2024)

[18]   Java Platform, SecureRandom. Available: https://docs.oracle.com/ja-vase/8/docs/api/java/security/SecureRandom.html (Referenced 5.5.2024)

[19]   Spring Security Advisories, 2024. Available: https://spring.io/security (Refer-enced 5.5.2024)

[20]   Snyk Vulnerability Database, react@18.2.0 vulnerabilities, 2024. Available: https://security.snyk.io/package/npm/react/18.2.0 Referenced (5.5.2024)

[21]   Snyk Vulnerability Database, react-redux vulnerabilities, 2024. Available: https://security.snyk.io/package/npm/react-redux (Referenced 5.5.2024)

[22]   Snyk Vulnerability Database, axios vulnerabilities, 2024. Available: https://secu-rity.snyk.io/package/npm/axios (Referenced 5.5.2024)

[23]   NordPass, Top 200 Most Common Passwords, 2023. https://nord-pass.com/most-common-passwords-list/ (Referenced 5.5.2024)

[24]   OWASP, Cross Site Request Forgery Prevention Cheat Sheet, 2024, Available: https://cheatsheetseries.owasp.org/cheatsheets/Cross-Site_Request_For-gery_Prevention_Cheat_Sheet.html (Referenced 5.5.2024)

[25]   Spring Security, Cross Site Request Forgery (CSRF), 2024. Available: https://docs.spring.io/spring-security/reference/servlet/exploits/csrf.html (Refer-enced 5.5.2024)

[26]   D. Abramov, npm audit: Broken by Design (2021). Available: https://overre-acted.io/npm-audit-broken-by-design/ (Referenced 5.5.2024)

[27]   OWASP Top 10, A09:2021 – Security Logging and Monitoring Failures (2021). Available: https://owasp.org/Top10/A09_2021-Security_Logging_and_Monitor-ing_Failures/ (Referenced 5.5.2024)