Password Manager

MAX GRATSCHEW

KALLE RAITANEN

GitHub [https://github.com/Gratschew/passwordmanager] repository, includes guide in ReadMe.md file.

Table of Contents

Vocabulary

Service data = data saved in database about services, includes service name, username and password. For example, “Facebook”, “Mikko Mallikas” and “Password!123”

2FA = Two-Factor authentication

General description

Password Manager - A password manager application that uses secure encryption algorithms to store user credentials. This was implemented as web full stack application. This is fully original code produced for this course. Project is full of little choices which make it more secure, and in this document, we list things that we think are the most important/worth noticing.

Structure of the program

Technologies used:

- Frontend: React
- Backend: Node.js, Express.js
- Database: NoSQL (MongoDB)
- Containerization: Docker & Docker compose
- Version control: GitHub
- Authenticator for 2FA
    - https://github.com/Authenticator-Extension/Authenticator
    - Chrome plugin can be downloaded from Chrome web store

Features:

1. Register/Login:
    a. **Description**: Logging in with username and password
    b. **Implementation**: password is salted and hashed with Bcrypt and a JSON Web Token (JWT) is issued.

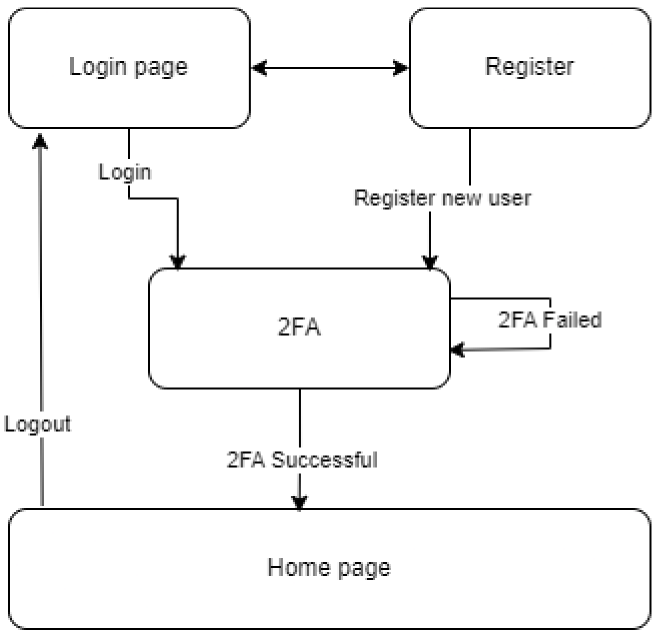
2. Password strength meter:
    a. **Description**: A password strength meter helps users create strong passwords that are resistant to brute-force attacks. Minimum length of the passwords is forced by the application.
    b. **Implementation**: Passwords shorter than 8 characters are considered to be weak [1] Password’s strength is measured using zxcvbn library created by Dropbox [2] and displayed to the user in the frontend.

3. Two-factor authentication:
    a. **Description**: 2FA for logging in to the application

  b. **Implementation:** TOTP based 2FA is set up on register. Backend issues a TOTP secret for the user on registration. The secret is confirmed by the user by giving the corresponding 2FA six digit code to the backend with the secret. The secret is then encrypted with an AES key derived from the user's password and saved to the database.

4. Saving user's service data:
   a. **Descritpion**: Service name, username and password will be saved to database as a single object.
   b. **Implementation**: Each object containing username, password and service name are encrypted using AES. Aes key is generated from user's password. That key is stored in memory only when user is logged in.

Program has basicly 3 pages: login, register and home page. Following flow chart describes how the program functions.



PICTURE 1: STRUCTURE OF THE PROGRAM

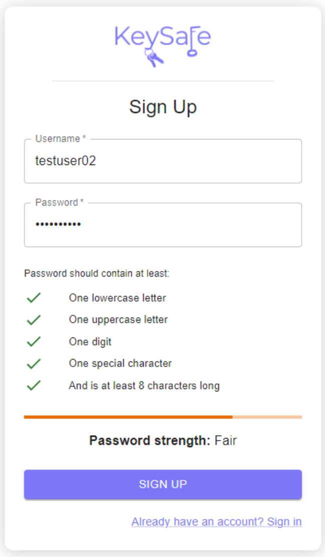
More of the security related features are presented in the next chapters.

Authentication

To use password manager, user needs to login using existing login credentials, or register to create new ones. 2FA is required.

We decided to use Token-based authentication, JSON Web Token (JWT). After successful login, token is stored in cookie (Secure cookie would be used, if the application was deployed to production to a https server), and used later to authenticate user, when making API requests.

User registers from the register page which can be seen in picture 2.



KeySafe

Sign Up

Username * testuser02

Password * ••••••••••

Password should contain at least:

✓ One lowercase letter
✓ One uppercase letter
✓ One digit
✓ One special character
✓ And is at least 8 characters long

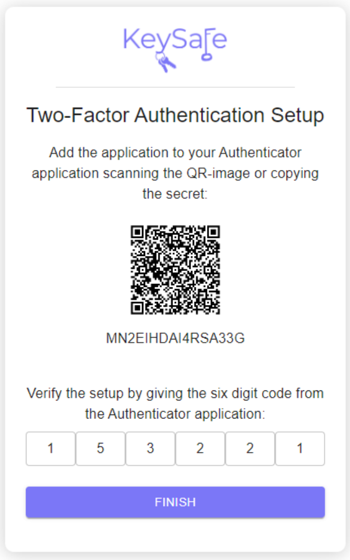
Password strength: Fair

SIGN UP

Already have an account? Sign in

PICTURE 2: REGISTER PAGE

After signing up, the 2FA is required to set up, as can be seen from picture 3.



KeySafe

Two-Factor Authentication Setup

Add the application to your Authenticator application scanning the QR-image or copying the secret:

MN2EIHDAI4RSA33G

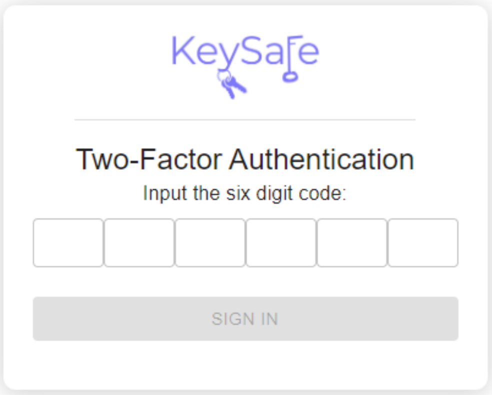
Verify the setup by giving the six digit code from the Authenticator application:

1	5	3	2	2	1

FINISH

PICTURE 3: 2FA SET UP

After 2FA is setup, the user is logged in. If the user doesn't set the 2FA and closes the application, the 2FA is forced to setup on the log in again. 2FA is required on every log in after it's set up, as can be seen from picture 4.



PICTURE 4: 2FA AUTHENTICATION

In case that attacker gets access to user's username and password, multi-factor authentication is the best solution due to it defending well against for example brute-force [7][8].

We decided to use Time-based One Time Password (TOTP) codes. User can use authenticator and require 6-digit code, which must be typed in order to proceed. If attacker cannot access the authenticator application, she/he shouldn't be able to log in.

Derivating encryption key from users' passwords

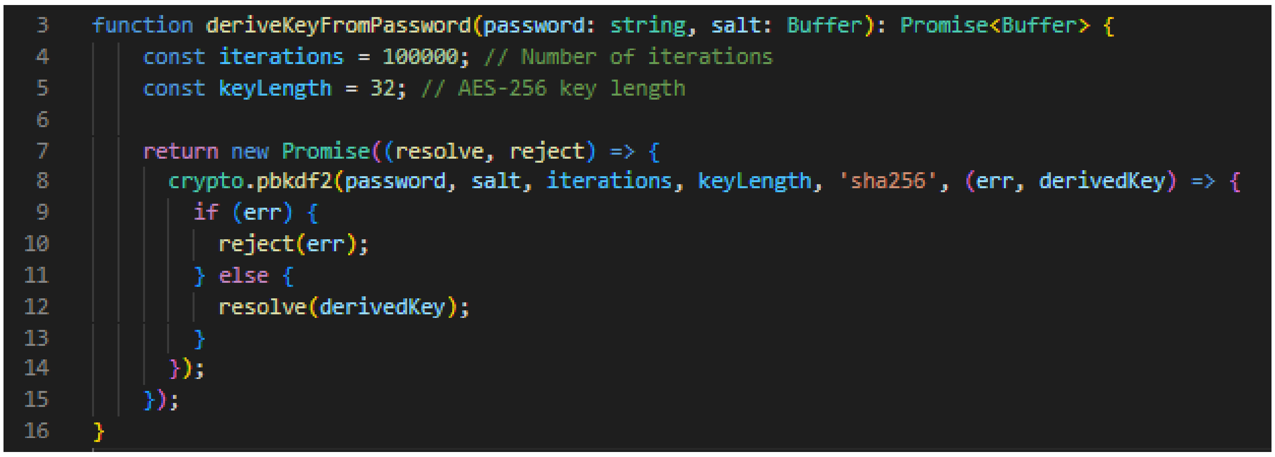
Encryption key which is used to encrypt service data, is derivated from the user’s password. Encryption keys are stored in memory only when user is logged in. There are few reasons for this design decision

- This way each user has different encryption (expecting that they have different passwords)
- Encryption keys are stored only in memory, when user is logged in
- User's password are strong and good fit for password derivation due to strict requirements when creating password
- The data can only be decrypted when the user’s password is known (The user is the only one who can decrypt it)

Key derivation was implemented using Password-Based Key Derivation Function 2 (PBKDF2) from nodes ‘crypto’- library. Function was chosen due to

- Recommended in exercise group
- Made for our purpose
- Can be used with digest algorithm

Even though there are newer options like Argon2, PBKDF2 is still a computationally expensive function and good fit for this project. We decided to use SHA-256 as digest function.



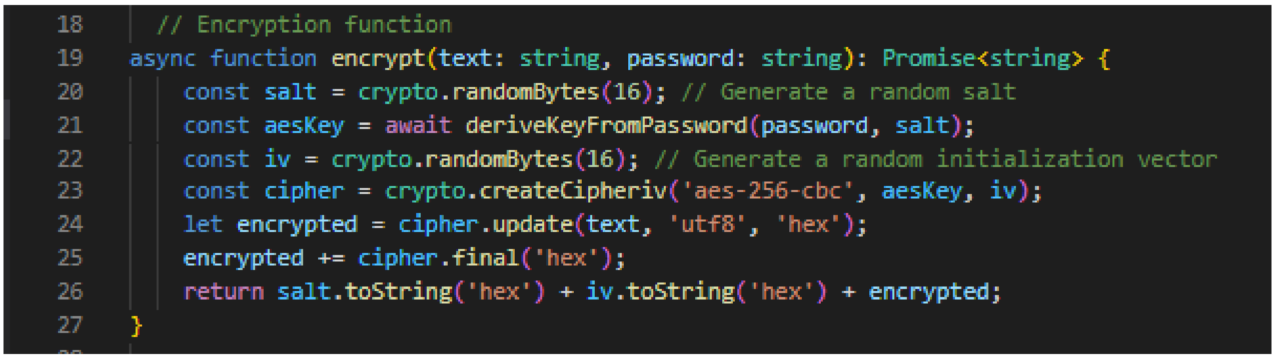
```
function deriveKeyFromPassword(password: string, salt: Buffer): Promise<Buffer> {
    const iterations = 100000; // Number of iterations
    const keyLength = 32; // AES-256 key length

    return new Promise((resolve, reject) => {
      crypto.pbkdf2(password, salt, iterations, keyLength, 'sha256', (err, derivedKey) => {
        if (err) {
          reject(err);
        } else {
          resolve(derivedKey);
        }
      });
    });
}
```

PICTURE 5: DERIVATING KEY FROM PASSWORD

Encrypting/decrypting service data

Service data is encrypted as an object. Aeskey is derivated from user's password. Algorithm used is AES-256

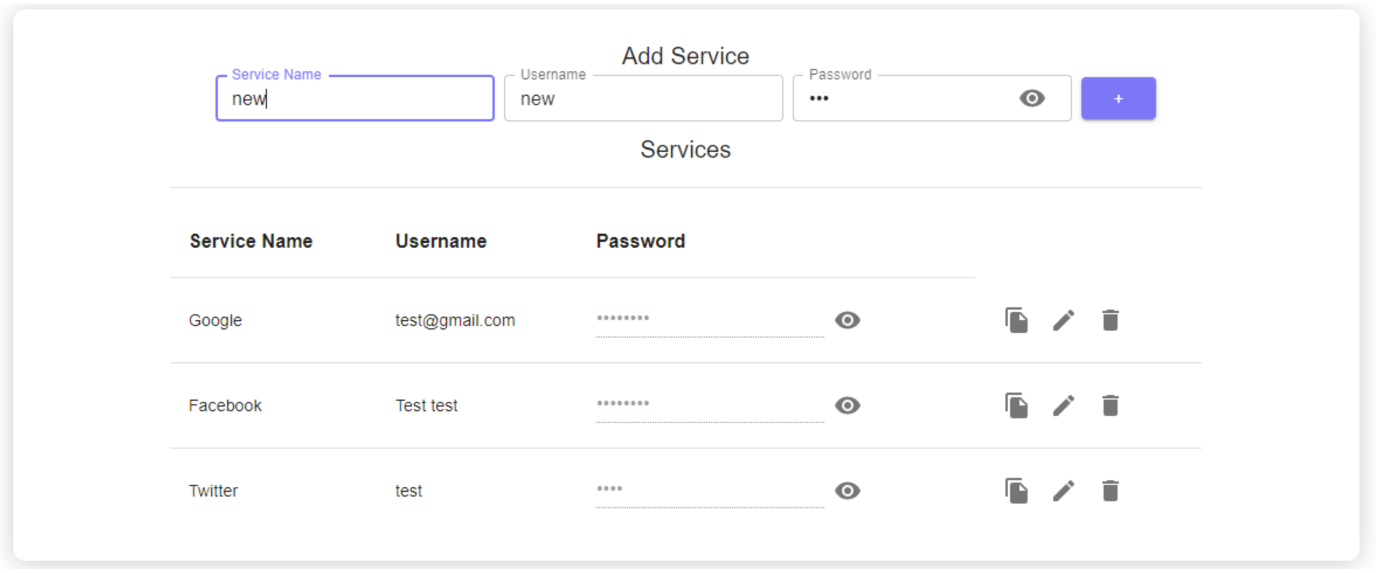


```
// Encryption function
async function encrypt(text: string, password: string): Promise<string> {
    const salt = crypto.randomBytes(16); // Generate a random salt
    const aesKey = await deriveKeyFromPassword(password, salt);
    const iv = crypto.randomBytes(16); // Generate a random initialization vector
    const cipher = crypto.createCipheriv('aes-256-cbc', aesKey, iv);
    let encrypted = cipher.update(text, 'utf8', 'hex');
    encrypted += cipher.final('hex');
    return salt.toString('hex') + iv.toString('hex') + encrypted;
}
```

PICTURE 6: ENCRYPTING SERVICE DATA BEFORE SAVING TO DATABASE

Sensitive data isn't shown in plain text

Password manager stores a lot of passwords, so it's mandatory for users to get password. By default, all passwords are invisible. Users can see passwords by clicking visibility- icon button or by using copy to clipboard button. This way it's possible to use password manager without ever showing passwords. This way password isn't compromised by shoulder surfing in public places or possible screen recording.



Service Name	Username	Password
Google	test@gmail.com	••••••••
Facebook	Test test	••••••••
Twitter	test	••••

PICTURE 7: ALL PASSWORDS ARE INVISIBLE IN HOME PAGE

Rate limit

Both Login and Two-factor authentication verification endpoints are rate limited. This means login/ 2FA attempts are maxed to 15 per IP. Window for those attempts is 15 minutes. This was implemented to prevent brute-force attacks, mitigation of credential stuffing attacks and to protect against username enumeration. Also server load and resource consumption gets reduces on the side. Rate limiting is something that has also been brought by OWASP and should be implemented [3].

Error & Exception handling

Error & Exception handling were implemented throughout the application. Those aspects are important in order to prevent information leakage and maintaining system integrity while giving the user proper guidance in case of errors.

Database

We decided to go with MongoDB to implement a secure database for the project. Both NoSQL and SQL databases are as secure as they are configured and we had more experience on configuring MongoDB. We configured MongoDB to use authentication, which means a connection can only be made using credentials.

Docker

We used Docker Compose to simplify the development and communication between the backend and the database. Docker or Docker Compose itself do not make application more secure but when it comes to deploying the application to production, the configurations become important and then can make the application more secure. For example image management, host security and secrets management are aspects that must be taken in to consideration when hosting the application for production. This is also something OWASP’s Top Ten list includes, in A05:2021-Security Misconfiguration [4]. Even though we didn’t go production with the application, we tried to configure the Docker-Compose and Docker files as securely as possible, using minimal and secure base images.

Testing

Manual testing was done during and after the development of this project. We made pull requests on every feature so we could review each other's code before merging the feature branch to the main branch. This kept code quality higher and catches few bad practices and bugs. Also, two separate testing sessions were hosted. The latter testing session was done when the product was finished and bug tickets were created of the found bugs on the project’s GitHub page. Also improvement aspects were found during testing and those aspects are written in the final chapter of this document.

Libraries

Libraries that were used in this project were chosen carefully by determining the number of users and the activity for the libraries. We only chose libraries that had the most users and were the most maintained and fit our needs. It’s important to prevent vulnerable and outdated components as it’s written in OWASP’s Top 10 A06:2021-Vulnerable and outdated components [5].

The used libraries are listed in the table below.

Library	**Purpose**
bcryptjs	Hash user’s password using Bcrypt
Cookie-parser	Parse UserID from JWT
cors	Enable Cors for the backend
crypto	Encryption and decryption using AES
dotenv	Access Environment variables set by Docker-compose
express	Implement backend’s API endpoints
Express-rate-limit	Implement rate limit for login and 2FA verification endpoints
jsonwebtoken	Implementation for JWT
mongoose	Use MongoDB from Node.js
otplib	Implement 2FA with TOTP
qrcode	Create QR-code from 2FA secret
axios	Making API-calls from the frontend
Js-cookie	Using cookies on the frontend
redux	State management on the frontend
zxcvbn	Checking password strength

OWASP Top Ten

OWASP Top Ten consists of the Top 10 web application security risks [6]. In the table below, it's presented how we were able to evade those risks with our implementations.

Web Application Security Risk	**KeySafe's solution**
A01:2021-Broken Access Control	Every service object is encrypted with an AES-key derived from user's password. This means the services are always in an encrypted state in the database and therefore can only be decrypted by the real user knowing the password. Anyone can create an account and there are no superior roles, which means the attacker can't access the decrypted sources in anyway.
A02:2021-Cryptographic Failures	Encryption and Decryption is done with the best practices utilizing the most common and reliable library built in Node.js called crypto.
A03:2021-Injection	We didn't implement sanitization but since Every service object is encrypted with an AES-key derived from user's password and the services are only accessed from a GET request, there is no input, and the services are always safe. Anyone can create an account and there are no superior roles, which means any injection wouldn't be beneficial for the attacker.
A04:2021-Insecure Design	The best attempts were done in order to create a secure design using the best coding and security principles.
A05:2021-Security Misconfiguration	We tried to configure everything with best practices and create generic error messages not revealing any sensitive information.
A06:2021-Vulnerable and Outdated Components	We only chose libraries that had the most users and were the most maintained and fit our needs.
A07:2021-Identification and Authentication Failures	Identification and authentication was made to be as strong as possible by forcing strong passwords, 2FA and using the best practices in JWT, storing the token in a (secure) cookie.
A08:2021-Software and Data Integrity Failures	Authentication, secure error handling and security testing were done. Data validation and sanitization is something that could've been implemented for better protection against Software and Data Integrity Failures.
A09:2021-Security Logging and Monitoring Failures	We didn't implement any logging or monitoring to the software.
A10:2021-Server-SideRequest Forgery	Sensitive endpoints are restricted by login, latest versions of libraries are used. Rate limiting also helps mitigating the impaft of

	potential SSRF attacks by limiting the number of requests from a specific IP address. We also use CORS whitelisting for development purposes, but in production the cors can be configured so that only a certain URL can make requests.

Improvements and known issues

The following list includes possible improvements and known issues for the application:

- Peppering
- Checking service password strength (like master passwords strength)
  - Increases users overall security in other services
- Common password (e.g., password123!) is not allowed
- Checks and warnings if stored passwords are leaked in data breaches
- When deleting a service, add confirmation modal
- Master password can be changed
- Instead of showing service name, username and length of the password, show only Service name
- 2FA code could be pasted
- Password cannot be recovered
- Users' password cannot be changed.
  - In case user’s credentials get leaked, there is no way to change password
  - it would require decrypting and re-encrypting all service objects with the newly derived AES key from the new password.
- User cannot be deleted
- User should be notified, when 2FA login fails, message should include time.
  - By email
  - Next time login

Sources

[1] https://pages.nist.gov/800-63-3/sp800-63b.html

[2] https://github.com/dropbox/zxcvbn

[3] https://github.com/OWASP/API-Security/blob/master/2019/en/src/0xa4-lack-of-resources-and-rate-limiting.md

[4] https://owasp.org/Top10/A05_2021-Security_Misconfiguration/

[5] https://owasp.org/Top10/A06_2021-Vulnerable_and_Outdated_Components/

[6] https://owasp.org/www-project-top-ten/

[7] https://cheatsheetseries.owasp.org/cheatsheets/Multifactor_Authentication_Cheat_Sheet.html

[8] https://owasp.org/Top10/A07_2021-Identification_and_Authentication_Failures/