Password manager

COMP.SEC.300

Victor Zhou Jiang

Table of Contents

Password manager 1

1. Introduction 2
2. Desktop application 2
3. Browser application 2
4. Server 4
5. Database 4
6. Signing up 4
7. Logging in 5
8. Password generation 5
9. Password encryption / decryption 5
10. Master password 5
11. Testing 6
12. File structure 6
13. Vulnerabilities 7
14. Future development 7

1. Introduction

This is a project created for COMP.SE 300 course of Tampere University. It currently has three different components, browser application, server and database with desktop application being developed. Server and database are being run with docker with only server being accessible to the outside.

The program supports multiple users. User is able to ether store their already own passwords or generate a new password with given length.

In the readme.md are the instructions on how to run the whole system.

2. Desktop application

Original plan was to develop a desktop application using C++ and QT framework but there were too many problems on getting openssl to work with QT so the development was halted and switched to browser application.

3. Browser application

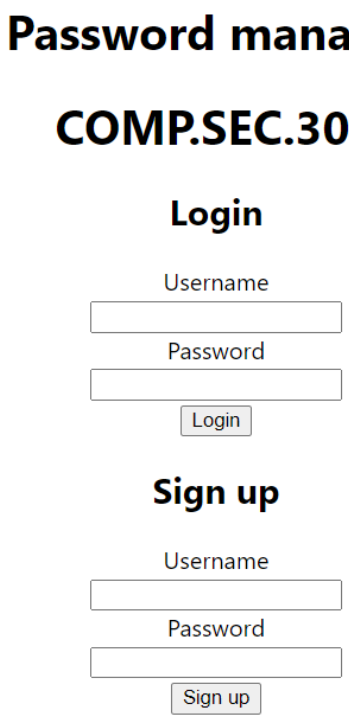
Browser application is created using Javascript and React framework to enable creation of single page application for smoother user experience. It communicates with the server through using HTTPS protocol thus ensuring that the messages are encrypted. Currently the application lacks css to make it pretty but it is intuitive to use.

With the application user is capable of creating a new user and logging in to see their passwords. They can save passwords and the site they are used for. The site value is not required. It's more to recognize where the password is used for. If the user desires they can delete any password they currently have stored.

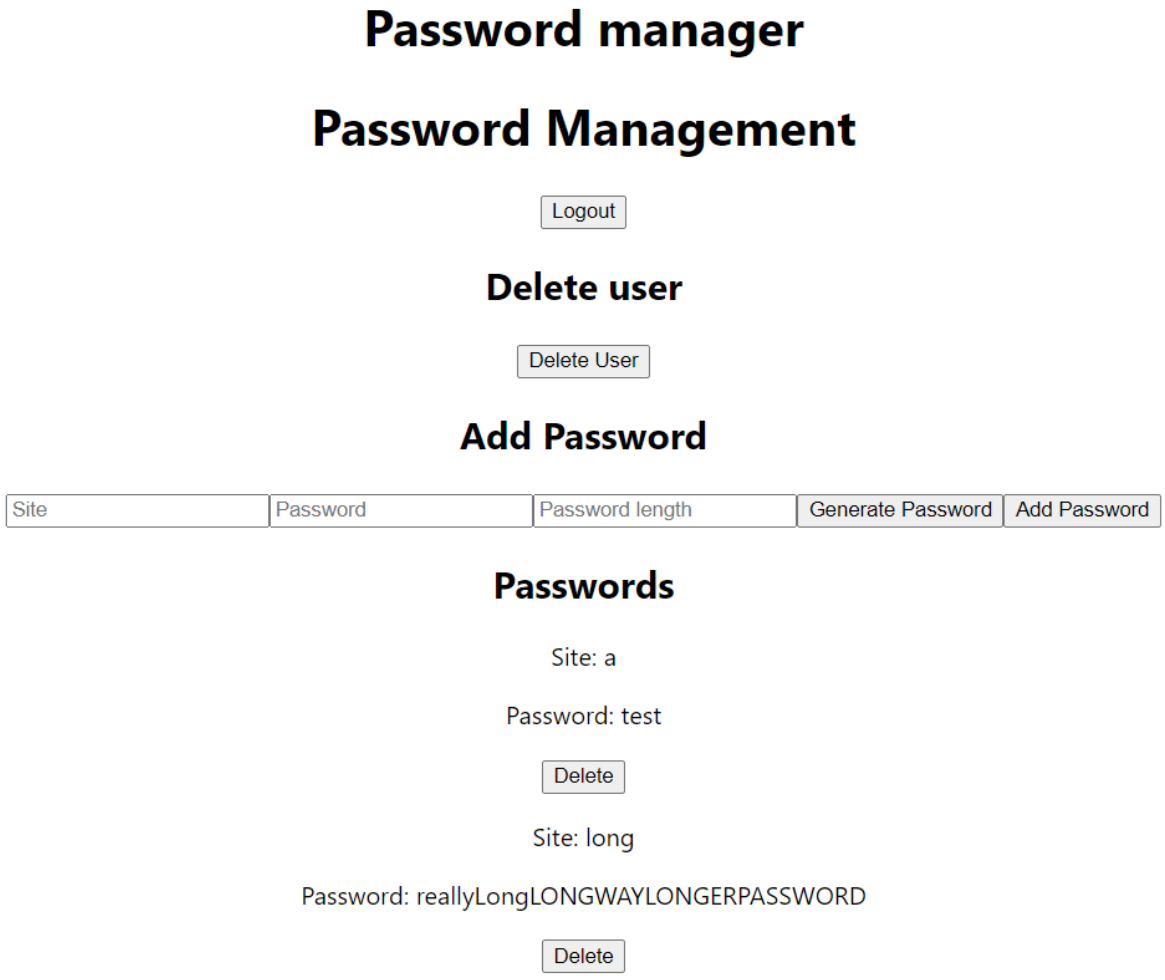
The application can also generate a password for the user with given length. This password is generated using cryptographically secure random generator. User can also delete their account thus deleting everything that has is stored in there.

As this is a SPA that uses JWT, CSRF shouldn't be an issue.

Picture 1 is screenshot of the login/sign in screen. Picture 2 shows the account page where user can do everything that was mentioned before.



Picture 1 Login/Sign up screen



Picture 2 Password management page

4. Server

The server is created using C# and ASP.NET framework. Most of the endpoints are secured with JWT to ensure that the client is who they say they are and only able to access their data. Only the register, login and getSalt are no authorization required endpoints. Get salt is used to get the salt for master password for login. Hosting is done with docker to ensure that the server can be used with any operating system. Currently the CORS has been set to only accept request from localhost:3000. Which should mitigate XSS.

Servers swagger can be accesses from https://localhost:8081/swagger/index.html. Please read the readme.md on how to run the server as you cannot run it by just cloning the repository.

5. Database

PostgreSQL is used as the database for this project. It's run through docker with docker compose. It's not exposed so no one outside of the docker network can access the system.

The current datatabase has two tables: Users and Password. Picture 3 shows the tables.

```
mydatabase=# \d "Users";
                              Table "public.Users"
     Column     |  Type   | Collation | Nullable |            Default
----------------+---------+-----------+----------+-------------------------------
 UserId         | integer |           | not null | generated by default as identity
 Username       | text    |           | not null |
 DerivedKeySalt | text    |           | not null |
Indexes:
    "PK_Users" PRIMARY KEY, btree ("UserId")

mydatabase=# \d "Password";
                              Table "public.Password"
      Column       |  Type   | Collation | Nullable |            Default
-------------------+---------+-----------+----------+-------------------------------
 PasswordId        | integer |           | not null | generated by default as identity
 UserId            | integer |           | not null |
 EncryptedPassword | text    |           | not null |
 Salt              | text    |           |          |
 Site              | text    |           |          |
 IsMasterPassword  | boolean |           | not null | false
 IV                | text    |           |          |
Indexes:
    "PK_Password" PRIMARY KEY, btree ("PasswordId")
```

Picture 3 Database structure

6. Signing up

The user creates an account by inputting a username and master password. After that the client will generate a salt using crypto.generateRandomValue() for hashing the password using sha256. This hashed password is then saved in the database and it's only used for logging in.

The client will also generate an another salt that will be used to create derived key for passwords. This salt is also saved in the database.

If the username already exist the client will promp the user that it already exist. As the client doesn't care about what the master password looks like, it's up to the user itself to create a strong password.

7. Logging in

To log in the user needs to input their username and master password. After that the client will fetch the salt for the master password. It will then salt and hash the password and send it to back to server. The server will then compare if the hashed password matches the one stored in the database. If it does the user is granted an JWT access token and redirect to their page. As the master password is used to encrypt/decrypt other passwords it is kept in memory until user logs out or refresh the page.

Incase of username not existing or user inputting a wrong password the client will create a promp that says "Login failed".

8. Password generation

In the password management page user has the ability to generate a password with wanted length. This password is generated using cryptographically secure random number generator which is build in in Javascript.

9. Password encryption / decryption

When password is being added to the system the client first encrypts the password with a key that is derived from the master password. Thus making sure that only the person with master password can see the password in its decrypted form. The encryption uses AES-GCM with IV being generated during the encryption. This IV is also stored in the database.

When user logs in to their page the system will fetch all the passwords the user has. These are decrypted first with the same derived key as mentioned above before showing them to the user.

10. Master password

Master password is the heart of the whole system. If this gets compromised the attacker can access everything that the user has stored. Sadly there was no better way to do this and so it's on user to create a strong and secure password.

This password is used to log in and the derived key that is got from this master password is used to encrypt/decrypt the passwords. Derived key is created in the following way.

1. The system will first create a salt when user first registers.
2. This salt and the master password are encoded to bytes.
3. The encoded password is then imported as a raw key. This is to get CryptoKey object.

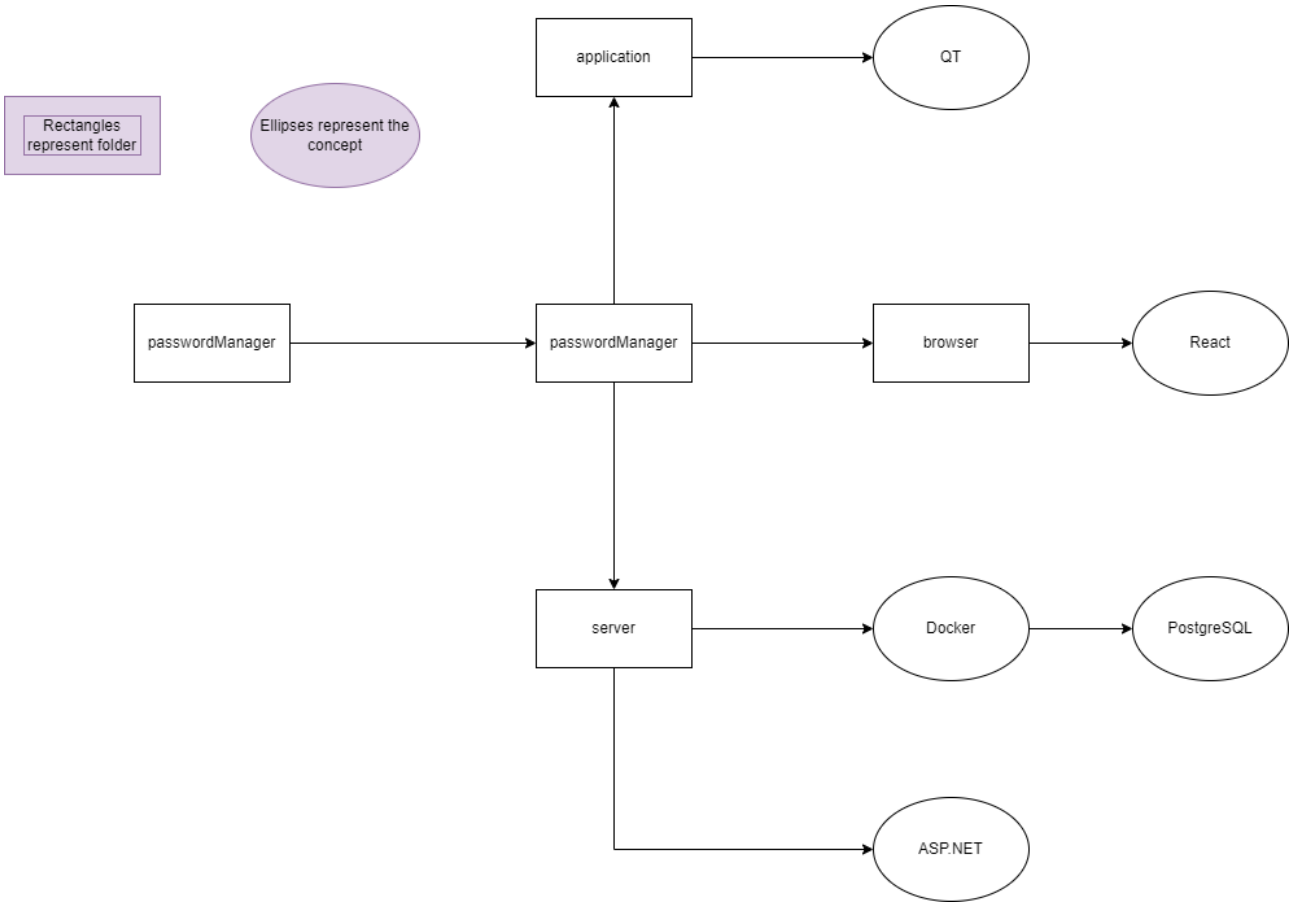
A derived key is created from the imported key using the PBKDF2 algorithm. PBKDF2 stands for Password-Based Key Derivation Function 2, which is a key derivation function. The derived key is generated using the salt, 1000 iterations of the algorithm, and the SHA-256 hash function.

11. Testing

Because of the problems with desktop application, it took more time than planned which left no time to implement automated test. The whole system has been tested using manual testing focusing mainly on exploratory testing. Some bugs were found but these were fixed. The biggest bug was with access JWT token not being accepted but that was also fixed as soon as it was detected. Other bug was password encryption and decryption because I had some problems when creating the derived key. This caused the password being encrypted but the key couldn't decrypt the password.

12. File structure

Picture 4 shows the structure of files. Inside the server folder is everything needed to run the server and database. In the browser folder is everything needed to run the browser application. In the application folder is going to be everything needed to run the desktop application which is in the works.



Picture 4 File structure

13. Vulnerabilities

There are couple of vulnerabilities that I realized after the final meeting

- Because you are using username to sign up, the attacker can use that to see if such username already exists. This leads to them knowing what usernames are in the system. This could be solved by using email and email confirmation for sign up.
- With the previous vulnerability it may lead to bruteforce attacks. This could be mitigated by adding timeouts after certain amount of wrong password guesses.

14. Future development

- Currently the JWT access token has the duration of 30 minute after that the user must log in again to do anything. This is because the system doesn't use refresh tokens. This could be implemented in the future.
- Email login could also be implemented as that is way safer than username. With email it is harder for attacker to know if given email has already been used. This is because with email there would be email confirmation for account creation. Currently when you try to create use a username that already exist the system tells you that which is a security vulnerability.
    - Reason for this not being implemented was the time restrains on learning email confirmation.
- Ability to modify existing passwords and master password.
- Ability to check the strength of the password. Using github.com/zxcvbn-ts/zxcvbn to check.
- Ability to import/export passwords.
- Limiting password length.
- UI improvements could also be made.