Requirements and Design

A. Introduction

CGJL

Team Members

Gabriel Undan

Luke McDonald

Jessica Ocampo

Christina Chen

Application

Simple Password Account Manager (SPAM)

Type: Password manager web application

Description:

A password manager similar to applications such as KeePass and LastPass. Instead of keeping track of multiple unique passwords, a password manager only requires the memorization of one master password used to access a password database.

Functional Requirements:

- User accounts
- Stores username, password, and website/description the credentials are intended for
- Passwords are not stored in the database in plain text
- Searchable by website/description or username
- Generates strong random passwords
- Allows users to modify or delete their saved passwords
- Logs accesses and password changes

Development Tools:

- HTML, CSS, and JavaScript
- Meteor JS framework
- MongoDB
- React
- Semantic UI
- IntelliJ

B. Requirements

Security and Privacy Requirements

Privacy Requirements

- User must be able to delete or modify their saved passwords
- User must be given a noticeable way to cancel the process in the middle of entering a password to be saved
- Developers/administrators should not be able to see the unencrypted passwords of users in the database
- User should be notified of when some user-initiated transaction was successful or not

Security Requirements

- Passwords must be not be visible during the transport between client and server on form submission and vice versa when not appropriate
- If found unnecessary, encrypted passwords should not be sent to any third party API that is used for development
- Critical user transactions on the application (getting passwords, storing passwords, deleting passwords, etc.) must be authenticated, and possible bypass should be prevented

Security Flaw Tracking Plan

For tracking known and stated security flaws, we will refer back to this section as development progresses to see if all of the 1) requirement items above and 2) all of the bug bars below have been or are being met and how. The questionnaire in the risk assessment section would also be updated accordingly. As we discover new security flaws during development, we will update our lists of requirements and bug bars with them. If we discover that just updating these lists and referring back to them is not sufficient enough for our security flaw tracking plan, then this plan will be modified as needed.

Quantity Gates (Bug Bars)

Privacy Bug Bars

End-User Scenarios

Critical

- Lack of notice and consent
    - Example: User presses some “Cancel” button on the form, and the password manager still stores what they’ve inputted up till that point
- Lack of user controls
    - Example: User is unable to remove any of their saved passwords in the event they choose not to store them with the password manager.
    - Example: If the user discovers they are reusing passwords across their accounts whose passwords they do not want stored with the password manager, they are unable to modify any of their saved passwords

Important

- Data minimization
    - Example: Encrypted passwords are transmitted from the database to some third party (ie. an API used by the password manager for development) without any need to

Moderate

- Lack of user controls
    - Example: There exists no obvious/visible feature to leave the form or to cancel the process of inputting the user’s password (ie. a “Cancel” button).

Low

- Lack of notice

    - Example: Users should be notified of when their password has indeed been successfully saved into the system, for when the process has been successfully canceled, etc.

Enterprise Administration Scenarios

Critical

- Lack of data protection
    - Example: Administrator is able to see the unencrypted passwords of users in the database without a need to

Security Bug Bars

Client

Critical

- Information disclosure
    - Example: Unencrypted passwords are visible during transport from the client to the server on form submission (ie. as query parameters)
- Unauthenticated access
    - Example: People are able to bypass authentication (ie. without going through the login form on the UI) and connect to an endpoint in the application that gives users their passwords

Risk Assessment Plan for Security and Privacy

Security and Privacy Questionnaire

Privacy

- Describe how users can delete or modify their saved passwords.
- Describe how users would cancel the process of submitting a password without closing the application, window, or browser tab.
- Describe how the administrators/developers are not able to know what a user's passwords are through the database.
- Describe how the user would know a transaction they initiated themselves failed or was successful.

Security

- Describe how passwords are not visible during transports.
- Describe all instances where encrypted passwords are transmitted to third-party APIs and why these instances could not be avoided.
- Describe the transactions that are authenticated and how.

Security Reviews and Threat Modeling Areas

The transactions that give out passwords, save passwords, and modify the passwords are the main areas that would need security reviews and threat modeling since sensitive information is being involved in those transactions.

C. Design

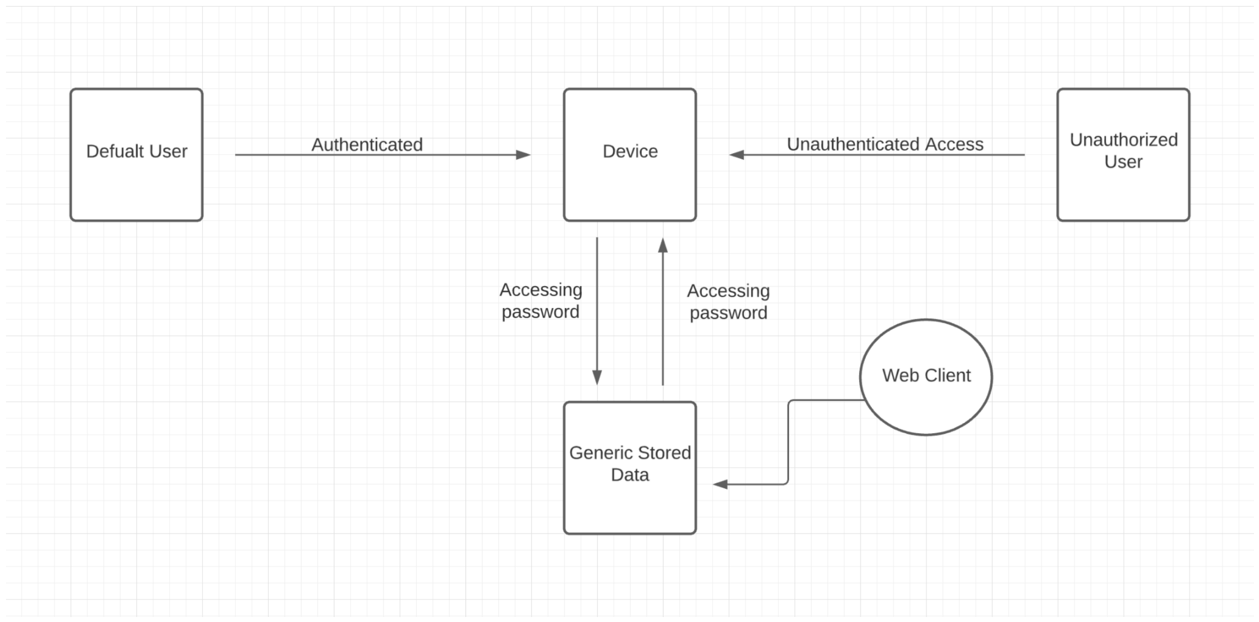
Design Requirements

- Authentication and Password Management Requirements
    - Minimum Requirements (e.g. character length, upper/lower case, numbers, special characters, password strength)
    - Master Password
    - Two Factor Tokens
    - Password Rotation (e.g. password is changed every 90 days)
    - Reuse of passwords across multiple sites
    - Notification Requirement (e.g. change in password or has been accessed)
- Password Encryption
    - Reassurance for security incase if database is accessed by unauthorized parties

Attack Surface Analysis and Reduction

When designing this application, our main concern would be accounts with access to the contents of the application and the application itself. If an unauthorized user were to gain access to the application and obtain the contents in it, we would fail at our goal of protecting a user’s passwords. Unauthorized users who obtain access may be capable of decrypting passwords that are locally stored. This is a problem that is similar to other password managers because of neglecting to remove locally stored data.

Threat Modeling



Identifying Threats

- **Default User**
  - Reuse of passwords
  - Using compromised passwords
  - Failing to secure device properly
  - Failed removal of password information from data storage
- **Web Client**
  - Passwords may be obtained through client
  - Introduction of threats may be created from interactions between client and user
- **Unauthorized Users**
  - Unsecured device set by default user
  - Access to user’s system (e.g. by brute force, tailgating)

Categorizing Threats

- **Default User -** Information Disclosure

- **Web Client -** Tampering
- **Unauthorized Users -** Tampering/Spoofing

Implementation

Link to Source Code and README:

https://github.com/CGJL/spam

A. Approved Tools

The following are tools that are required for development of the app and tools that are not specifically required but used by our team.

Tool	Version	Type/Purpose
git	2.30.0	version control
HTML5	---	framework/language
React	17.0.2	framework/language
JavaScript	ECMA 2015 (ES6)	framework/language
Semantic UI React	2.0.3	framework/language
MeteorJS	2.2	framework/language
MongoDB	4.4.4	framework/language
Intellij IDEA	2021.1.2	editor/IDE
npm	7.16.0	package manager
ESLint	7.28.2	static analysis
Iroh	0.3.0	dynamic analysis

B. Deprecated/Unsafe Functions

The following are functions that have been deprecated in the tools at the versions listed in the above section. The functions included here are what we are planning to or may use.

React

1. Deprecated: `componentWillMount()`

    a. Replacement/Solution: Use the constructor for all use cases that were associated with the deprecated function
    b. New alias: `UNSAFE_componentWillMount()`
    c. Provided reason: Unsafe to use with certain async rendering features
2. Deprecated: `componentWillReceiveProps()`
    a. Replacement/Solution: `getDerivedStateFromProps()`
    b. New alias: `UNSAFE_componentWillReceiveProps()`
    c. Provided reason: Unsafe to use with certain async rendering features
3. Deprecated: `componentWillUpdate()`
    a. Replacement/Solution: `getSnapshotBeforeUpdate()`
    b. New alias: `UNSAFE_componentWillUpdate()`
    c. Provided reason: Unsafe to use with certain async rendering features
4. Deprecated: Use of `javascript:` in URLs
    a. Replacement/Solution: Use React event handlers
    b. Provided reason: Easy to accidentally include unsanitized output in an HTML tag

MongoDB

1. Deprecated: `db.cloneCollection()`
    a. Replacement/Solution: `mongoexport`, `mongoimport`

C. Static Analysis

For our project, we will be using ESLint. We're specifically using the AirBnB style to follow. Due to this project using JavaScript as the main language, ESLint is the most popular check style framework. It offers a lot of configurations that could allow us to create our very own style, however AirBnB's configuration is so popular in the industry that it has become the standard.

It seems to be relatively easy to use each time since running would just require `npm run lint` with npm. The output provided includes the files of the errors, reason, and the locations in the files. In addition, we haven't tried to modify the original that was given with the Meteor React template, but looking at the layout of the .eslintrc configuration file, it seems straightforward to modify with new rules as needed.

Verification

A. Dynamic Analysis

We are planning to use a third-party tool for JavaScript called Iroh (https://github.com/maierfelix/Iroh). The basic steps to using this tool are copying and pasting a section of the code to be tested as a string variable, instantiating a `Iroh.Stage()` object and passing in that string to the constructor, calling the `addListener()` property of that object, and choosing a runtime event and a property from Iroh's API (https://github.com/maierfelix/Iroh/blob/master/API.md) to pass into that listener. This runtime event and property is related to the nature of the code (ie. a function, an if statement, a loop, etc.) that is to be tested and the point of execution of that code (ie. on enter of the if statement, on leave of the if statement, etc.), respectively. Output generated by Iroh from testing the section of code can then be received and used in things such as `console.log()` statements.

Though it was a bit of a challenge and took a bit of time to try and learn how to use this tool at first, the layout of its API makes this tool relatively easy to understand mostly. One scenario where using Iroh actually helped out in finding a logic error during development was when a section of the code that uses falsey values, such as null, was selected for testing. `[variable1] && [variable1] != [variable2]` was tested when it was used as an `if` condition. This was through using the `IF` runtime event and its `enter` property. If `variable1` was `null`, then the condition is `null` rather than the expected `false`. After discovering this, code that uses this logic was modified as appropriate. Another scenario where this tool helped out was when it allowed us to discover a misplacement of a return statement within a function's nested control structure. The runtime event `FUNCTION` and its properties `enter` and `return` were used for this.

A week after choosing our tool and doing our initial analysis, the status of our dynamic analysis is up-to-date as far as fixes go. We just continued doing the same: updating existing ones as the functions were changed and providing fixes along the way for whatever issues that were found. One new scenario where Iroh helped us detect a potential bug was accessing properties, like the `.filter()` property for JavaScript arrays, on null values. When accessing properties on null values, the application didn't seem to continue the execution of the program. We had not run into this situation before applying dynamic analysis on this portion of the code, and so, it had gone unnoticed. The fix we implemented was just a simple ternary statement to access the property only if the value was not null and to return an empty array if it was. Similarly, for each element in the array looped through with `.filter()`, we were accessing one of its properties and thus found that we also needed to provide a null check for those.

B. Attack Surface Review

Between the time of writing the Approved Tools of the Implementation section of the report and writing this section, it does not seem like there have been any updates or vulnerabilities reported for those tools.

C. Static Analysis Review

Regarding the status of our static analysis at the time of writing this, after running the tool we chose, the type of errors that are pending fixes are `no-undef`, `no-eval`, and `no-unused-vars`. These errors are related to the Iroh package however, and the sections in our code that use it follow the same format that is displayed in the samples provided on their GitHub page. In the next assignment, we will be attempting to find a solution to solve these ESlint errors by finding an alternative to `eval()` for running dynamic analyses and possibly using aliases for the JavaScript import statements to solve the first and third mentioned errors.

D. Fuzz Testing

1. SQL Injection
    a. We tried this attack by entering vulnerable SQL statements such as:
        i. "admin'--" into the login form to try and get admin access.

        ii. “' or 1=1--” to avoid logging in
        iii. “' GROUP BY **table.passwords** HAVING 1=1 --” to display all passwords

    None of these attempts worked because the technical stack of our web application does not use SQL based database backend, we use MongoDB which is NoSQL. This also wouldn’t work because of the dependencies we use to handle Form inputs automatically sanitizes inputs.

2. Cross-site Scripting (XSS)
    a. To attempt a XSS attack we inputted HTML element tags and JavaScript tags into a handful of input fields such as:
        i. “<script>alert(“Test”)</script>”
        ii. “<h1>Test</h1>”

    These attack attempts did not work due to our web application technical stack automatically escaping special characters inputted into forms to display them as strings and not as actual tags when they’re eventually loaded into the page.

3. Encryption Key Leak
    a. Since our application relies on encryption and we do not save it in our database. It is generated on every login and saved to the client. We ensured that it is stored in the client where it is not publicly accessible.
        i. It is not found in the Local Storage of our browser, session, cookies, cache.
        ii. Our web application framework Meteor allows us to store things locally on the client as a mock MongoDB database and the way to access it is through development tools such as: Meteor DevTools. However, we do not publish the data for the client to access on it’s own, the encryption key is not publicly accessible.

Release

A. Incident Response Plan

Privacy Escalation Team

- Escalation Manager - Luke McDonald (lukemcd9@hawaii.edu)
    - Monitors incidents and escalates what incidents need to be taken care of urgently. Also ensures that fixes are made in a timely manner depending on the severity of the incident.
- Legal Representative - Christina Chen (cchen2@hawaii.edu)
    - Monitor incidents that could potentially lead to legal issues. They will handle whatever legal affairs we need to deal with.
- Public Relations Representative - Jessica Ocampo (jnocampo@hawaii.edu)
    - Communicate with end-users the incidents that we are dealing with. Ensure our users that we are working on fixes and our timelines for when they will be fixed.
- Security Engineer - Gabriel Undan (gundan@hawaii.edu)
    - Find solutions to incidents whether through the community or ourselves. Perform adequate testing to ensure that incident is dealt with and plan for release.

In case of emergency, contact any of the Privacy Escalation Team's emails.

Procedures in case of an incident

1. Investigate report of an incident and bring it up with the Privacy Escalation Team.
2. Escalation Manager will assess the severity of the incident and assign a priority to resolve it depending on the severity.
3. Public Relations Representative will inform end-users of the incident and what the team is doing to respond to it.
4. The Legal Representative will look into any legal repercussions that we could potentially have due to the incident.

5. Security Engineer will develop a solution to the problem in a timely manner and release it to the community.
6. After the fix is released, the Security Engineer will perform a forensic investigation on the incident and see if there are any vulnerabilities still in the application and apply patches to fix them.
7. If there was any data loss, ensure that we have backups to restore from.
8. Make copies of all system logs that dealt with the incident in case it is needed for any legal issues.

B. Final Security Review

In our Static Analysis Review section above, we have mentioned our use of ESLint to make sure that all of our code adheres to our coding standards and rules. The only exception to this still remains to be the implementation of Iroh, our dynamic analysis tool. It uses `eval`, though according to the GitHub documentation, this appears to be how the tool is intended to be used. However, since this is only a dynamic analysis tool that doesn't impact function, code for running Iroh can be removed without affecting the rest of the application. Doing so would eliminate our last ESLint errors.

As mentioned previously, our use of Iroh as a dynamic analysis tool has helped us to check our runtime logic and processing of variables. It can be easy to overlook an error when visually checking over code, but Iroh has allowed us to quickly identify bugs (e.g. checking for `null` instead of `false`) in our code that may have been harder to pinpoint. During this final check we have made sure that our code that handles user data (i.e. pages for adding and editing passwords) is handled and validated correctly, while also accounting for erroneous input and attempted attacks such as XSS and SQL injection.

From the user's perspective, the application functions as intended. Users are able to create an account, add passwords to that account, view their passwords, and edit them. Looking at our feature set and the quality gates we have set, we have met most of them. The exception to the rule involves the visibility of the encryption key. If this application were deployed, the server it is hosted on should have HTTPS, which prevents the plain-text, unencrypted passwords from being seen by other parties that manage to get the packets. However, as mentioned previously, the user's encryption key is stored locally on their machine. We have tried looking for it in places such as the browser's local storage or browser memory (using Process Hacker) and have not successfully found where on the client machine this data is stored. However, it has to live somewhere. If an attacker is able to find it, it's possible that they could use it against a client. However, the risk is low since the attacker would have to compromise both the application's database as well as a client machine to get both an encrypted password and its corresponding key.

Considering this evaluation, we think that our application would fall under FSR passed with exceptions. Though our app is not vulnerable to common attacks that could leak user data, it does have the code that calls Javascript's `eval` function, which goes against coding standards and triggers ESLint. This could potentially be used in malicious ways. However, this code is non-critical to the application's function as it was used for dynamic analysis. It can simply be removed and the application will function as normal.

C. Certified Release & Archive Report

Latest Release

Version 1.0.0: https://github.com/CGJL/spam/releases/tag/1.0.0

Features

- User registration and login
- User can store endless amounts of accounts with username, password, associated url, and extra information if needed
- When passwords are saved to our database the data is encrypted with a unique encryption key for every user.
- When viewing passwords they are decrypted with the user's specific encryption key
- Users can edit the account information that is saved

- Users can view all passwords they have saved and filter through them

Future Development Plans

- Two-factor authentication
- Password reuse check
- Password rotation
- Modified passwords notification

Technical Notes

Installing and running the app

1. Clone the repository or download a .zip of the source code at https://github.com/CGJL/spam.
2. Install npm and Meteor to your machine

- Meteor: https://www.meteor.com/
- npm: https://www.npmjs.com/

3. Open up a command line session, change the working directory to that of the repository or source code, and "cd app".
4. "meteor npm install" to install the dependencies needed to run the app.
5. "meteor npm run start"
6. Visit localhost:3000 in your browser to start using the app.

Accessing the local MongoDB server

1. Ensure the app is running first, open up a new command line session, and change the working directory to the same one as above.
2. meteor mongo to access the local MongoDB server that is automatically created when running the app.

Development Tools

- HTML, CSS, and JavaScript

- Meteor JS framework
    - Meteor template from http://ics-software-engineering.github.io/meteor-application-template-react/
- MongoDB
- React
- Semantic UI
- IntelliJ