Introduction

Team Name: Wallacea

Team Members: Scott Yuk, Zolbo Tomita, Keaton Wong

Application Title: SafePass

Type of Program: Web-based Application

Program Functions: Our program will be a password manager. This will allow the user to create an account and save multiple usernames, passwords, and to what site/application. There will also be a function to create a secure password where you can alter certain requirements like having special characters, upper and lowercase letters, numbers, and length.

Development Tools:

- HTML
- CSS
- JavaScript
- Database (SQL or MongoDB)
- GitHub

Requirements

1. **Security and Privacy Requirements**

   Log based system that is dynamic, that's able to store personal information into a database. The logging system would have specified keys that can only access specific information when needed to access a person's information. Along with a lock and key style system time stamps and cookies would be saved each time an attempt is made to log in, for a certain period of time. This would give a greater chance at recovery if the database were to get hacked.

2. **Quality Gates**

   Currently, we only have a rough idea of what we are going to do so we cannot go into detail about the privacy and security quality gates.

   Privacy Bugs

   Critical:

   - Lack of notice and consent
   - Lack of data protection
   - Lack of internal data management and control
   - Insufficient legal controls
   - Improper use of cookies

   Important:

   - Lack of notice and consent
   - Lack of data protection
   - Data minimization
   - Improper use of cookies

   Moderate:

   - Lack of data protection
   - Data minimization
   - Improper use of cookies
   - Lack of internal data management and control

   Low:

   - Lack of notice and consent

Security Bugs

Critical:

- Elevation of privilege

Important:

- Information disclosure
- Spoofing
- Tampering

Moderate:

- Security assurances

Low:

- Not enough protection

3. **Risk Assessment Plan for Security and Privacy**

Implementation of API's would be a way of risking the security and privacy of our users personal information, so finding specific weaknesses within any of the API's we may implement would be a start to ensuring the strength of our application. Along with API's the database we use to store all of our user's valuable information would be another potential point of attack. As we want SQL as our database, we may need to worry about things like injections or other vulnerabilities that come with using an SQL database.

Design

1. **Design Requirements**

   For our specific application we would want user information key encryption within the database that can only be decrypted by our own method or API. This would ensure the user's sensitive information is protected even if a leak or attack on our website is made and some information is stolen, the people breaking in would have a hard time trying to decrypt code. Along with providing different ways for the user to access their information like two-step authentication. Also allowing users options like choosing to store their information on hard drives if they wish or storing it with our applications database. Having a combination of tight security with our encryption idea and allowing the user to have more autonomy with how they wish to secure their information would be the best mix.

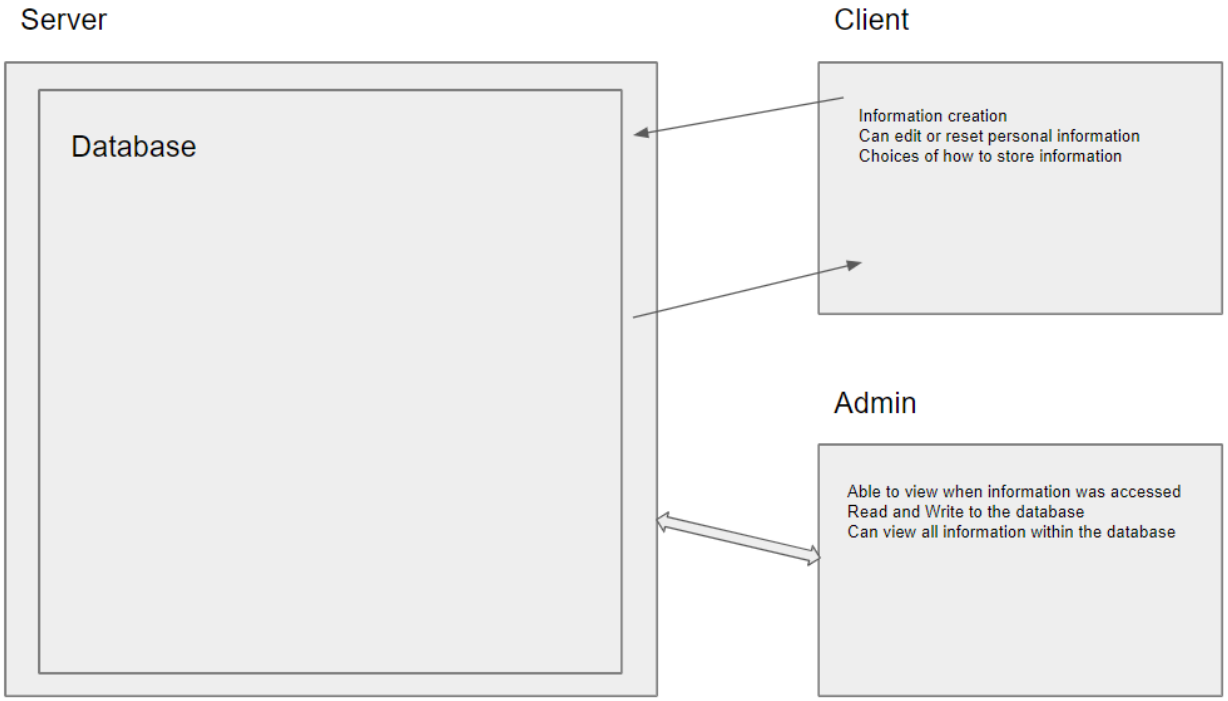
2. **Attack Surface Analysis and Reduction**

   Admin level: This user will have access to read and write to the database and see time stamps of when personal information is attempting to be accessed along with the cookies that would be stored with the time.

   Consumer level: This user has the most basic access and will only allow them to create an account and save passwords.

   Common Vulnerabilities with Password Managers: There are not a lot of vulnerabilities with password managers because of the way they are created. Due to zero-knowledge architecture it makes it so that only the user knows what is stored in the password vault. This prevents server-side attacks from affecting the user's password data. By trying our best to implement our functionalities with the zero-knowledge protocol we can ensure that our users' PII is unreachable by potential attackers.

3. Threat Model



Implementation

1. **Approved Tools:**
    a. Visual Studio Code version ~~1.78~~ → 1.79.2
    b. Bootstrap version 5.3
    c. MongoDB version 6.0
    d. HTML version 5
    e. JavaScript version 13
    f. Node.js version >= 10 and <= 14

2. **Deprecated/Unsafe Functions:**
    a. remove() - deprecated to comply with the MongoDB CRUD specification. This change was aimed to provide a consistent API for CRUD operations for all MongoDB drivers.
        i. deleteOne() and deleteMany() should be used when wanting to remove specific information from the MongoDB database.

3. **Static Analysis:**
    ESLint version 2.4.0
    We are deciding to use ESLint as our static analysis tool as it's a convenient and helpful plugin tool to find some weaknesses in our code. Since our group is mainly using JavaScript, ESLint will help us identify certain issues that are repeated within the code as it will be immediate feedback for us to change before committing code. Past experience with ESLint also helped keep code readable to everyone in a group and outlines specific problems that can be overlooked. Overall a useful tool for keeping code well organized and generally a good tool for giving warnings of potential issues.

Verification

1. **Dynamic Analysis:**

   Our group is mainly using JavaScript as the main coding language for our program and with some research we found that JSHint and JSLint in combination with ESLint would be a good combination to be the dynamic tool we use to ensure that our code would have low security vulnerability. Initially we wanted to use JSHint as our dynamic tool as it's protective but less restrictive which is perfect for our needs. Upon trying to implement JSHint we found out that it was deprecated and no longer recommended for use, and JSLint was also not as highly recommended but still a good analysis tool and so we decided to use JSLint. The implementation of JSLint has given a few coding warnings, but no major vulnerabilities were presented in our code.

2. **Attack Surface Review:**

   Upon checking all the approved tools we are using for our project, there have been no updates for any of the tools used. Even though the time frame between the releases of the latest versions of the tools being used and this update was short, it is important to keep an eye out for updates. In previous classes, keeping tools used in a project up to date hasn't really been focused on, but now that we are strongly keeping security in mind while creating our own project that uses multiple different tools, we will be sure to make a greater effort to do so.

3. **Fuzz Testing:**

   For our fuzz testing we attempted to hack our website in a few different ways. We attempted to hack our website by trying to access the admin level account first without an account and logged in as a regular user. Trying to access the admin page without an account proved to be ineffective as it would always show the login page of the website. Next, trying to access the admin page as a regular user also proved to not work, as trying to change the URL or update the status of the user only refreshed the page and didn't allow access to the admin page. Our next plan of attack for our website was to create another user account with an email that already exists. Our test proved to work as when

attempting to create an account with an already existing email shows the user an error message. Our last way of attacking our website was to do SQL injections anywhere we could on our website. The testing consisted of trying to hack into an already existing user and trying to get information from the database. When trying to access another user's account with SQL injections proved to not work as a user can't use symbols when creating an account for the first time, and when trying to input a symbol when trying to login in showed an error message to the user. When trying to access the database we attempted to do SQL injections anywhere on the page and ultimately couldn't get anything as either seemingly nothing would happen or blank pages would appear.

4. **Static Analysis Review:**

   Once a running version of our website was made, we went to fuzz test our website. After fuzz testing the website we ran ESLint, which presented a few warnings about expectations. When reviewed we changed and updated the code in an attempt to get rid of the warnings and fuzz tested again. On the second attempt of fuzz testing the results were the same and once the testing was done we ran ESLint again. The second update of ESLint still presented a few warnings, but as they are minor warnings we decided to leave them be as it doesn't affect our website.

5. **Dynamic Review:**

   Using JSLint to see vulnerabilities in our code, within our group after compiling code we would run JSLint to find vulnerabilities within our code and try to fix them. We would often get a lot of errors of missed things and prop type errors. Doing this ensured that the code we were making had the least amount of vulnerabilities before testing. Overall JSLint was a good way of seeing that the logic of our code was good and that checked our code for basic errors.

Final Security Review

1. **Incident Response Plan:**

Keaton - Legal Representative and Public Relations Representative.

Having both roles of legal representative and public relations representative the duties would be to ensure that the website fits within all legal spaces and provides support if a legal issue occurs. Along with legal assistance, public relations and communications would have to be built with networking opportunities.

Scott - Escalation Manager.

The main focus of this role is to prepare for any potential issues that may occur with the website and how to deal with those issues. Having to create ways to detect and resolve issues that occur and supervise teams while issues are occuring.

Zolbo - Security Engineer.

Testing and analyzing code for potential threats and weaknesses. Once finding a weakness, necessary steps can be taken to resolve the issue or a plan of defense can be created if the issue cannot be resolved.

Our application will have two contact emails and an emergency phone number created for users. One email would be for general inquiries of the website and general support. The second email would be for submitting flaws/bugs of the website users may experience. The emergency phone number would be a 24/7 number for any emergency the user may be facing, as quick notification and action would be needed for the user if an issue is occuring, like a hack of a user's information.

In the case of an incident, we would go through a three step process to ensure it is handled properly. Identification, Containment, and Recovery and Remediation.

Identification

During this step we will identify the incident based on impact and severity by determining how many users were affected and what type of data was compromised. We would also have to monitor and check logs to detect and track suspicious activity.

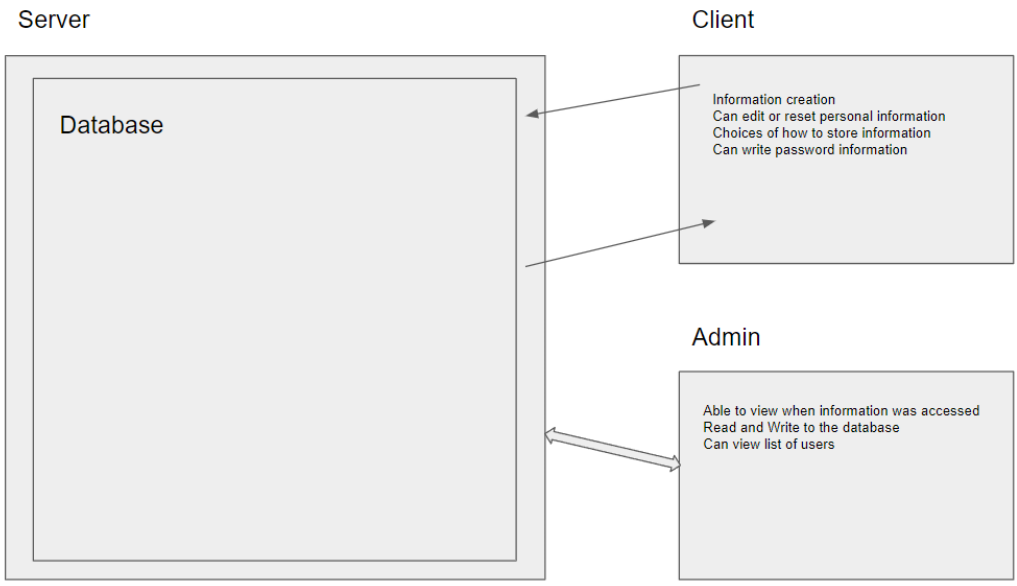
Containment

After determining the impact, the escalation manager would then figure out what to do to make sure that no further damage can occur by securing and isolating our system.

Recovery and Remediation

Due to our service being a password manager. If our application were to be compromised we would need to notify our users so that they can take the proper precautions. The legal and public relations representative would then inform affected users about the incident emphasizing the importance of taking the necessary precautions like changing passwords. The security engineer would then implement robust measures to strengthen the application and conduct a review of the incident detailing what was learned and identify areas of improvement.

2. **Final Security Review:**



Through our application, it is possible to register for an account and store passwords so that you won't forget them. Our threat model was adjusted slightly. Previously it stated that admins were able to view all information within the database. This is a security risk so we are limiting their view to just the list of users. Our static and dynamic analyses passed without errors. We conducted penetration testing on our application and tried various ways people could get into an account like account escalation and SQL injection.

Due to this we have concluded that the grade of our application is FSR pass with exemptions.

3. **Certified Release & Archive Report:**

   SafePass is currently at version 1.0.0 and is in development. It can be found here.

   Summary of Features

   - Secure login system
   - Store and edit passwords

   Future Development Plans

   - Strong password creator where requirements can be edited
   - Auto-fill passwords
   - Application accessible through website

   Installation

   1. Install Meteor
   2. Clone our repository onto your local machine
   3. Within your IDE of choice, open the cloned repository
   4. Go to the terminal and make sure you're in the correct directory and run the following commands.
      - meteor npm install
      - npm install
      - meteor npm run start
   5. The installation of SafePass is now finished and you can now access the application at https://localhost:3000/