Connie Huang (connieh) | **Stephen Suen** (ssuen) | **Sashko Stubailo** (sashko)

6.170 P3.3 | RUMI - FINAL VERSION

Our cost sharing app, **Rumi**, is designed to facilitate the process of splitting costs with *roommates*, assuming that each roommate periodically makes a purchase that benefits at least one other roommate. Costs we are expecting include utilities, rent, or apartment essentials (food, cleaning supplies, furniture, etc).

Rumi does not handle the transfer of money but rather it does take care of *tracking* expenses, we are less concerned about attacks to our system that would initiate monetary transfers without user consent. The worst breach we could face is likely from an existing user on Rumi who either accidentally or intentionally malformed data that is tracked in the Transaction History.

As our main goal is to keep an honest log of expenses and transactions between a group of housemates, our priority is to make sure that the data that is stored is validated, sanitized and safe from future malicious edits.

Security Policy

- All users:
    - Can view expenses and payments
    - Can add new housemates
- Individual users:
    - Can add as many expenses as they would like and portion them among their housemates accordingly.
    - Can add a payment to another user
    - Can edit or delete expenses and payments they originally created.
- No one:
    - Can delete a household
    - Can delete a user from the household
    - Can edit the Household Activity Log

Threat Model

- Users can add new housemates to a household without approval of existing housemates. This new housemate could enter faulty expenses.
- Users have free reign to edit their own expenses or even delete them entirely.

Security Concerns and Solutions

Problem: Malformed data

One of the most important security concerns for any application is that someone might submit data that they shouldn't be able to. This could be as simple as someone creating a malformed object that will later crash the page when it's loaded, or as bad as a user being able to edit other users' content. Because our data deals with real life money, we need to ensure that data cannot be submitted (intentionally or accidentally) that could modify the log of the household expenses.

Solution: Data validation before it reaches the database

We validate all of the data input by users on the server right before it's entered into the database. This means that any time any method is used to create a new household or a new expense, the object is thoroughly checked for all possibilities of security concerns or poor structure. With these checks in place, we don't have to worry about validation on the client side for data input, or about sanitizing data when it's displayed. All of these things are done in one server method.

Problem: Losing track of expenses, either accidentally or maliciously

Although there is validation in place to make sure a user submits well formed data, there is the possibility that a user may accidentally delete an expense while browsing older expenses. There is even the possibility that a roommate decides to "edit" some older expenses in their favor with the hopes that their other roommates won't notice.

Solution: Logging changes in the Household Activity Log

Because users will eventually use this service to repay each other in real life, we not only have to safeguard our users from technical errors, but also from potential fraud by their roommates. To deal with this issue, each household keeps a household activity log that saves any changes made to the household so that nobody can change, delete, or add a transaction without someone noticing. There is no mechanism for deleting data from this log, so it's a reliable history of the household's data.

Testing

In order to test Rumi, we tried setting up a number of different testing frameworks, including the unofficial test runner of choice, RTD, and frameworks ported over from Node.js such as Mocha. RTD, while supporting unit tests, was far too expansive for the needs of the project and has little documentation online. The Mocha package for Meteor was no longer up-to-date and was incompatible with the latest version of Meteor. Finally, we settled with the Laika testing framework, which was originally designed for end-to-end testing, not unit testing. Nevertheless, Laika had more documentation available, allows for testing in both client and server environments, and leverages existing libraries like Node's assert. Laika works by injecting server-side code, but doesn't seem to be able to access all the methods we wrote, so there were huge limitations in this process. As much as the framework allowed, we wrote tests for the user and household models in the style of unit tests, discretizing individual functions as much as possible. This was done in both the server environment as well as in a combined server + client environment (to test such features as login).