Applied Cryptography

Final Project Design Document

Richard Greenbaum, Rudolf Hernandez

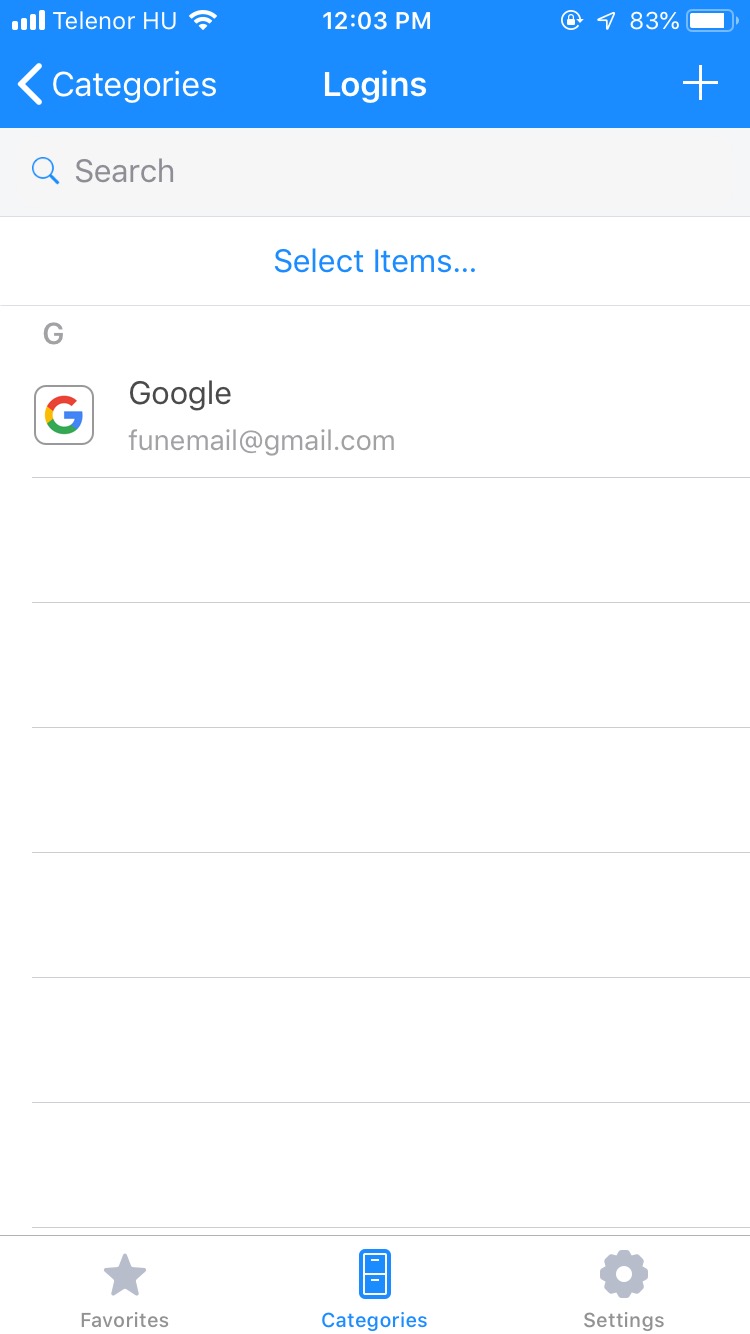
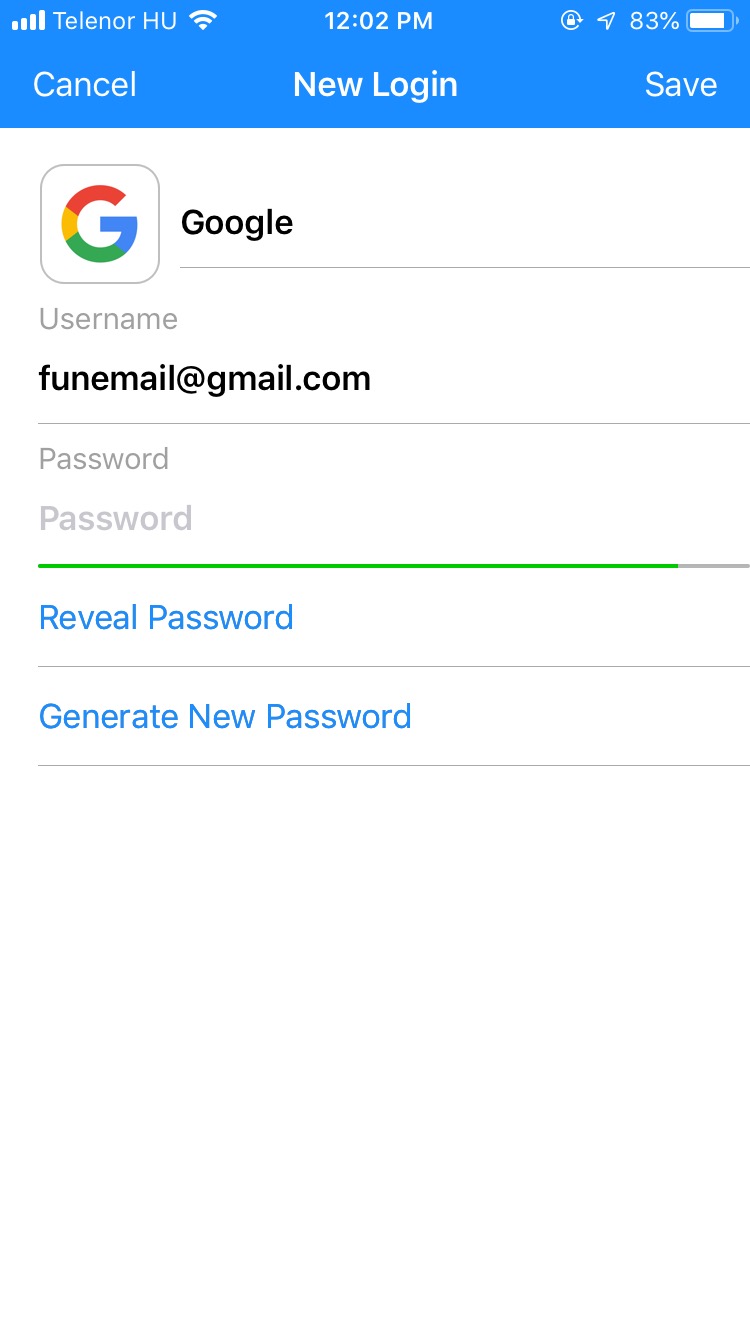
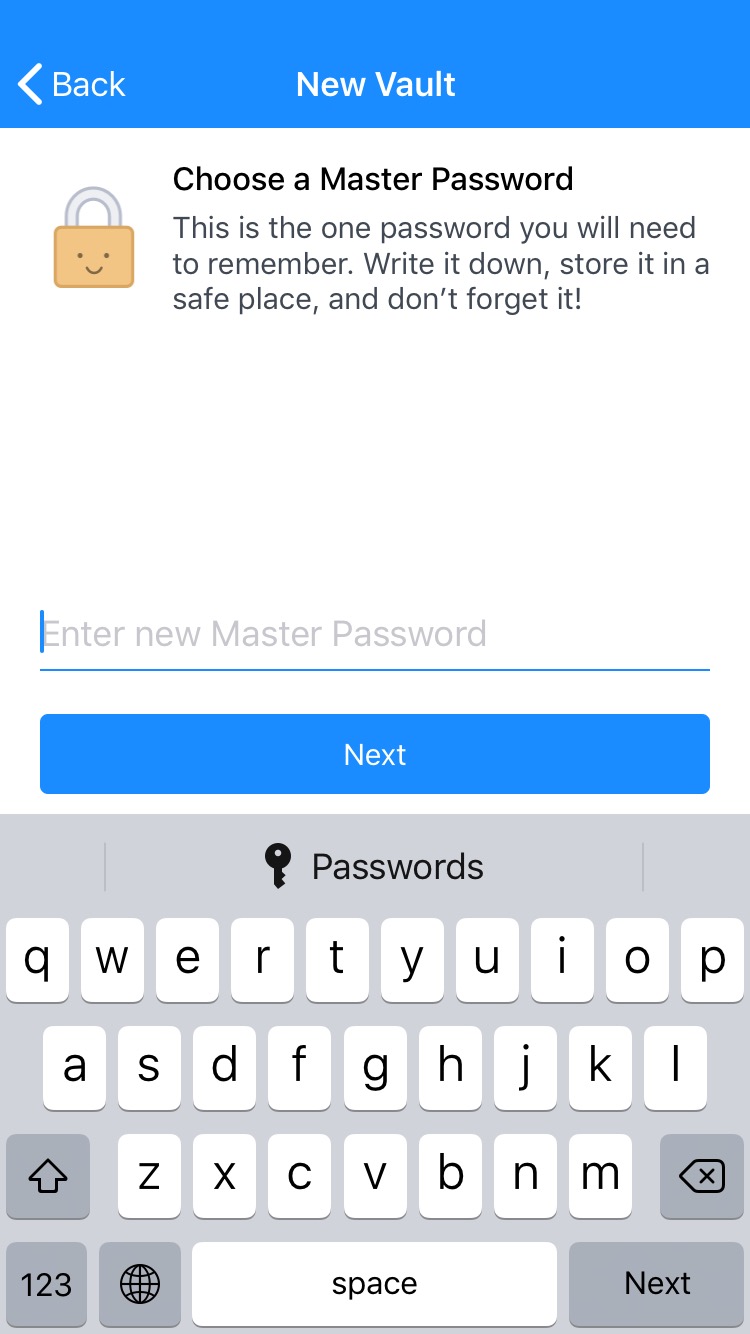
Overview of System Architecture and Design

For our final project, we will be implementing a secure password management application as an Android keychain app. The code will be written in Kotlin (very similar to Java) and the application will be built using Android Studio. In this section we will give a general overview of the application, covering the different activities the app will have and the functionality that the app will provide. Later sections will discuss the cryptographic primitives used to achieve maximum security in greater detail.

Upon opening the application for the first time, the user will be presented with a fairly simple activity prompting them to input a master password and then to confirm (retype) that master password in a separate text field. The user will also have the option for the app to generate a secure master password for them. If the user inputs two master passwords that do not align, the text fields will be emptied and the user will be prompted to input the passwords again. If the user is a returning user, as stored in Android Shared Preferences, the user will be prompted to input their master password for verification. After the user has successfully created or provided their master password, they will be taken to a scroll view activity that displays all of their accounts and usernames. Each account will have a button that says “reveal password” which, when clicked, will display the password on the screen until a hide password button has been clicked. This screen will also have a “new account“ button which will take the user to a create account activity. This activity will prompt the user for an account name (Gmail), a username (matthew456), and a password (cryptoisgreat24). It will also give the user the option to randomly generate a secure password in the same way as for the master password. After all steps have been completed, the account will be added to the account list in the scrolling activity. The user will also have the ability to delete and edit accounts. Time allowing, we will allow the user to automatically input a stored password on the corresponding web page or app.

During the creation of a new account, users will also be given the option to set a reminder for when they should change their password. So, for example, a college student who is storing a password on the app could set a reminder to notify him/her to change this password every semester since colleges usually require we do that anyway. Time permitting, we intend on making this completely functional in-house so that the user does not have to navigate to a 3rd-party site and have to manually change their password on the site as well as on their keychain. If a user were to change their password on a 3rd-party site, the app should send a notification to them asking whether or not they would like to automatically incorporate this change on their keychain as well. Although the core functionality of the app will be to securely store and generate passwords, we also plan on offering users the ability to store data that is just as important as their passwords, such as their credit card numbers or other relevant bank information, essentially creating a one-stop shop for users to store their most crucial information.

There already exist several keychain apps on the app store. Here are three screenshots from one such app that are very similar to the three activities we plan on creating.



Attacker Model

A cryptographic attacker model is composed of the presumed goals and capabilities that an attacker of a system would have. For our application, the goals of an attacker include obtaining the master password, obtaining the password-based encryption key, obtaining a specific account password stored in the app, and predicting the output of the secure random passwords provided by the app. The main attacker capability that we will assume when creating our application is that the attacker will gain access to the memory of the device at some point in time, and that they will be able to identify the location in memory of the passwords stored by the app. For this reason all passwords stored in memory will be encrypted, and the amount of time that unencrypted passwords spend in memory will be minimized. We also assume that the attacker knows the cryptographic primitives that we are using to encrypt the passwords, allowing him or her to mount an offline dictionary attack and verify the correctness of a candidate master password. Thus the security of our application cannot be reliant on the secrecy of our algorithms.

Security Requirements

**1. The master password should be strong**

Keychain applications provide utility to their user by reducing the number of passwords that they need to keep track of. While the ability of one single password to unlock countless other passwords is the primary utility of the app, it also creates a great cryptographic risk to the user, because it exposes the stored passwords to an attacker if they are able to guess just the master password. We will have several application components in place to protect the master password. The first is that we will require that the master password be at least 8 characters long. This ensures that the state space of possible passwords is very large. Assuming that each character can consist of an upper or lower case letter, a number, or a special character, the state space of an 8 digit password is 91^8 passwords long.

The master password will not be stored directly in the phone’s memory. Instead, a hash value will be computed by hashing the master password with the Kotlin hashCode function and then repeatedly hashing the output of the function for half a second. When the user tries to log in at a later time, the hash of the input password will be computed in the same way and then it will be compared to the stored hash value. If the values are equal, then the user inputted the correct master password. Otherwise they did not. Note the increase in security that is gained by repeatedly hashing for half a second, protecting even fairly weak passwords. As an example, assume that the user created an 8 digit password that consists only of lowercase letters, and that the attacker has this information. This will result in a brute force attack on 26^8 possible passwords. For an attacker to generate and verify each possible password, they would need (26^8)/2 seconds, or more than 3300 years. The odds that the attacker would arrive at the correct password in the first 10 years is less than 0.3%.

**2. All passwords should be encrypted**

We will encrypt the user’s data (i.e. account names, usernames, and passwords) using the recommended Advanced Encryption Standard (AES). Since AES uses a substitution-permutation network to encrypt data (with a given key), it replaces bytes from one table with the bytes from another, and as such creates permutations of data. The AES key will be 256 bits long and will be derived from using a Password-Based Key Derivation Function (PBKDF2) on the user’s master password. Using a 256-bit key with AES makes for a key space that has size 1.1 \* 1077, which would make a brute force attack on the key infeasible even by a supercomputer. Since PBKDF2 hashes a password with a salt many times over, even if two users happen to use the same password, the key will be unique. This prevents one key, if published online, from potentially being used to expose the data of multiple users.

In order to generate the salt, we use the Kotlin cryptographically strong pseudo-random generator function SecureRandom(), and set the salt to be sufficiently long such that it would be computationally infeasible for an attacker to try all possible passwords (i.e. due to the Birthday paradox, choosing salt length to be 64 leads to an expected collision after 232 (or 4 billion) passwords). We then create a password based encryption object (in Kotlin, this is known as a PBEKeySpec object), passing as input the user’s master password, the pseudo-randomly generated salt, the iteration count (NIST guidelines recommend a minimum of 10,000 iterations[[1]](#footnote-1), but due to security-performance tradeoffs, we will opt for a smaller number of iterations), and the length of the key (in our case, 256). We then pass this object into a secretKeyFactory object which outputs the AES symmetric encryption key, while also allowing us to specify the cryptographic algorithm we would like to use (in our case, PBKDF2WithHmacSHA256).

Now that we have our AES CBC key, we use the SecureRandom() function again to initialize an IV, and use PKCS7 padding. To encrypt a password, we will use AES CBC with the encryption key and a random IV. We will then store the encrypted password as well as the unencrypted IV. We will be able to decrypt this value using the same key and stored IV at a later time. Note the using CBC encryption with a different random IV for each password ensures that, if the user uses the same password for several accounts, the encrypted passwords will all be different. Thus the attacker will not know that the user has duplicate passwords, and no information will be leaked.

**3. Our password generation should be secure**

Password auto-generation will be performed using the SecureRandom() function mentioned earlier. Users should also have the option to configure password auto-generation to the requirements they are seeking. Suppose, for example, that a website or native app does not allow long passwords, or the dash symbol, or any other property that the default password auto-generation algorithm performs; the user should be able to put some constraints on this algorithm so that a secure password is still generated for them (within the confines of the restrictions they place on the auto-generation function). If the user selects to have a secure password generated for them, they will get to choose the length of the password as well as whether or not each of the following groups of characters can be included in the password: lower case letters, upper case letters, numbers, and special characters. We will then generate a password by adding the selected character arrays together and randomly selecting len(password) indices from that array. Generated passwords will be encrypted in the same way as user inputted passwords; using AES CBC with the encryption key derived from the master password and a randomly generated IV that is unique to that password.

1. <https://pages.nist.gov/800-63-3/sp800-63b.html#sec5>. See Section 5.1.1.2 [↑](#footnote-ref-1)