Objective

Passwords are a fundamental part of online security. These codes protect our identities and serve as an authentication check for the services people use. Unfortunately with the increasing number of passwords required for various services, users can find themselves using passwords for multiple services. This reuse increases the risk for the user as a compromised account can unlock more services to the attacker.

Password manager is a program that helps users create unique passwords without the need to memorise these words. These created passwords are for services are stored on the user's machine so that the user only has to remember a single password to unlock their online services. Our aim for this program was to safely store passwords, on the user's computer so that they could be retrieved only by the password owner.

Program Features

Program design

The password manager's data structure is focused around the 4 key features:

Login:

The user types the password to unlock the program and load the file into the data structure of the program. The login would also create a new user if required.

Inputs from User: Login take two datasets from the user: Username and main password.

Inputs from file: Login function loads the saved data file into the program.

We chose to use the

Main Menu:

Implements the functions of creating and retrieving the passwords. The main menu navigates the program and implements the features.

Inputs: User choice for navigating between the program features.

New Password:

Retrieve Password:

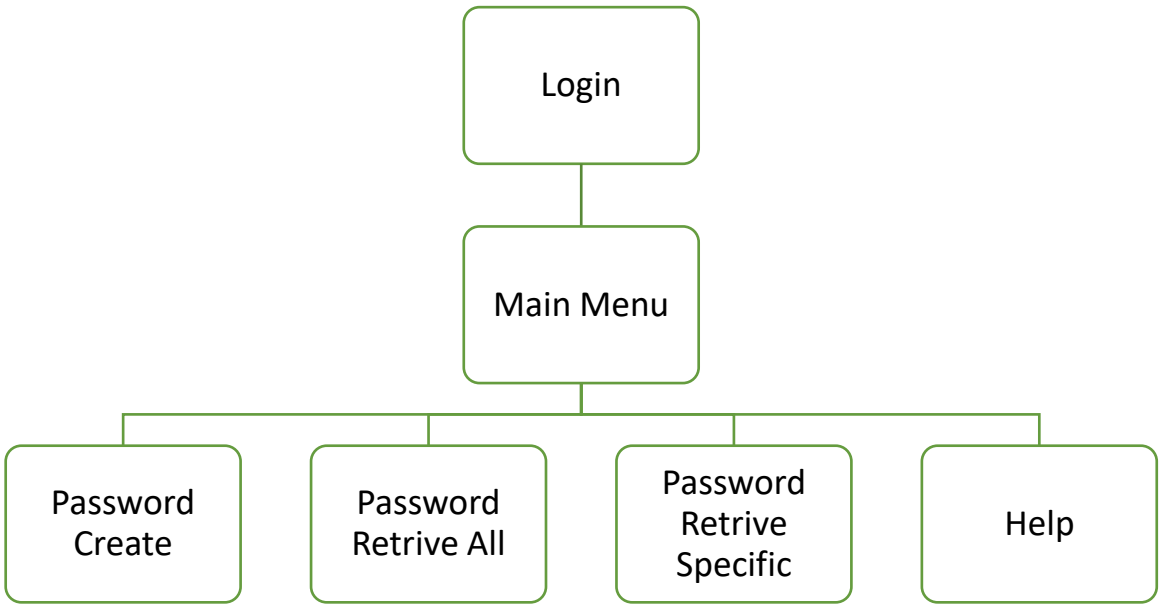
Program functions:

Encryption

For our program a key feature is the encryption of the main password. This is a secret which protects the program from unauthorised access to the users created passwords. To secure the main password the program uses a one way function. This function takes the users password and encrypts it using the random function with included salt. The output of the one-way function is very difficult to reverse. When the user logs into the system, the password given is then sent through to the one-way function and compared to the known string.

This type of encryption achieves the goal of authentication for the user into the system. It allows the salted one-way password to be saved in plain text on the hard drive. We chose this because it allowed us to be flexible with the how we saved our data. If we could retrieve the data though an unencrypt function then a bad actor could gain access to the stored passwords by reversing the encryption process.

Program Structure



Problems Occurred (Bad Pointers)

During code writing the user inputted String which was the name of the next service. Unfortunately the old password we thought was not terminated properly, it would re-write the first three letters of the next inputted string after the password was written. This issue only on the second password and all subsequent passwords written to file.

Firstly we tried to correctly terminate the old password, so that it would stop writing next user input to it. While debugging this issue, we could see that name was correct writing to file. This led us to believe it was a load error from the file. We considered that the load could be overwriting the original passwords. To bug fix we took out the load functions and used an array to see how the specific things being written as the password and next string.

During this implementation of the code we realised it was an errant pointer. The pointer caused doubling writing of the string for the user. This was because the number of bytes available to write the new password and the number of bytes being written did not match. To fix this we checked though our data structures and ensured that we were writing with the correct number of bytes. The issue was resolved after we became more aware of our memory allocation to array.

Problem Occurred (Multiple coders)

Due to the scope of our project we needed to use multiple coders to complete the task in an adequate time. We identified this as an issue at the planning stages of our project, and implemented a 2 step strategy to complete the works.

1. We broke the program up into parts with a defined data structure. This allowed us to code independently before integration of the program.
2. Kept a strong coherence to our designed structure.

During this coding phase we had to make a few changes to how data was accessed throughout our program. In our original plan we did not fully comprehend the data we should have passed to the encryption algorithm so that it would successfully encrypt the files. We also realised we would require the code to come together earlier than expected. We therefore implemented a number of smaller integrations rather than a large integration of the code at the end. This worked well because it allowed us to see the program take shape earlier and better focused our efforts on the parts of the program that did no work.