Password Manager - Writeup & Documentation

Josh Qian, Ugo Ofor, Elizabeth Garcia, Claire Burkhardt, Michi Okahata, Claudia Schmdit, Beauttah Wanja

Overview

This password manager uses a **SQLite** database to store securely-encrypted passwords. The database is accessed by a **Flask** API that links the backend database and custom-created **cryptography module** with an **Electron/Node.js** front end that the user can access given that they provide a secure Master Password. This stack is packaged by **Forge** into a Windows executable that allows the user to launch the application and manage their passwords from a convenient desktop executable. The password manager allows users to setup a profile that, upon login, gives them access to securely-stored passwords. It analyzes this database of passwords for potential security issues, such as weak passwords or repeated passwords.

Please download the Windows installer file from our most recent release (v4) — under the Releases section of our repository. Follow the installation structions in the `windows-release` branch of the repository (or in the Documentation PDF), where there are setup instructions.

The Windows executable and overall stack is the most stable version of this project. Use this for running and evaluating the project. The Mac password manager — packaged in a DMG file — is a work in progress. The Mac project works well in some instances but crashes in others, largely depending on the individual machine it is run on. This has to do with whether or not the

machine settings allow the Flask API to properly connect with the database and whether the machine launches the DB in sync with the front-end packaging. The Mac version currently builds and generates a DMG, but the DMG does not always (reliably) launch properly. Refinement of the Mac branch - among other development items - is a part of future development goals. You can run the Mac version from the terminal. See the `mac_download` branch for setup instructions, debugging, and FAQ.

Motivation

Everybody wants a password manager - a streamlined way to organize highly secure passwords that are impossible to memorize due to volume and uniqueness. Many of us, however, may not trust a company like Apple or Google to store those passwords. There are three reasons to be wary of company-based password managers: first, storing passwords with a company may enable them to misuse our personalized information. Second, individuals may want to avoid storing any passwords on the web whatsoever due to the vulnerability inherent to cloud-based storage. Third, many professional password managers who try to mitigate these concerns operate on a subscription model.

We created a free-to-use password manager packaged as a downloadable application to avoid company interference, cloud-based security risks, and predatory pricing models. Users are able to download our app onto their local machines. After a successful download and set up, users will be directed to a user interface, allowing for easy access and creating a user friendly experience.

Each user will locally host their own container to store their database of passwords. The app follows a master-key workflow that imitates existing cybersecurity workflows to enable verification and validation alongside secure, encrypted storage of sensitive information. A master password will give users access to their entire password database.

Video Demo

Please refer to the following demonstration video.

[START HERE] Installation Instructions (Windows)

1. Download version 4 of our password-manager release on the github.
    a. https://github.com/cburkhardt27/password-manager-cs-370/releases
2. Follow the installer's setup installation steps.
    a. *You may need to instruct your computer to "Trust This File" prior to installation. This is not due to security design problems in the executable, but rather because we have not signed our code with a recognized code certificate that Windows might be expecting.*

3. Launch the program from Desktop. **Run the file as an administrator.** The password manager should launch!

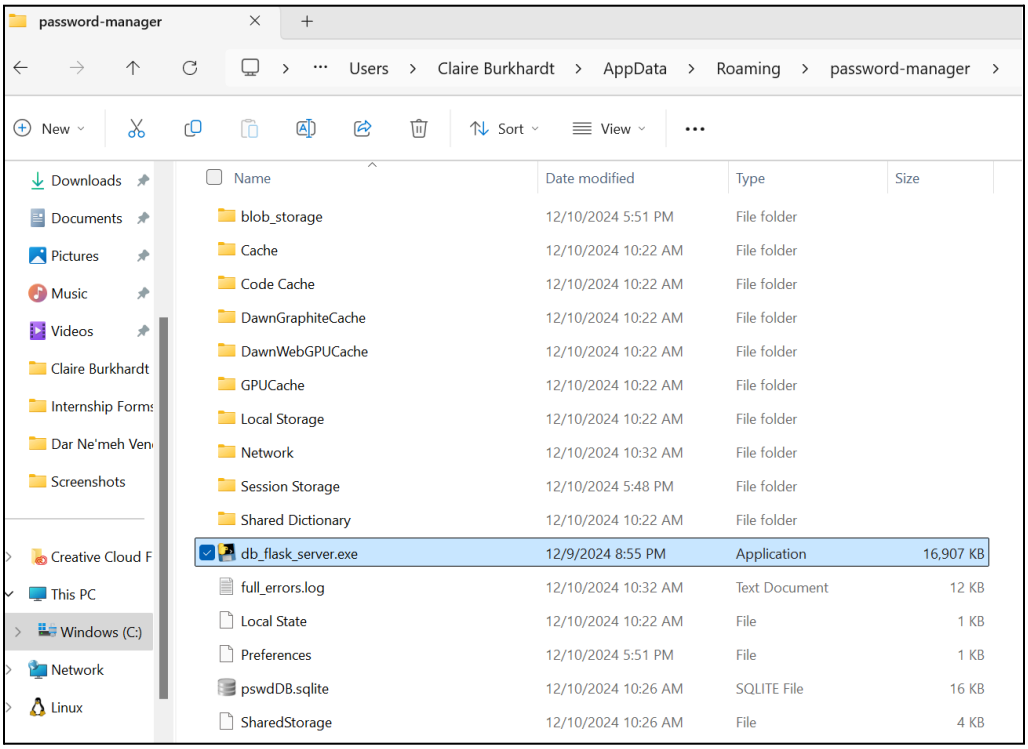
If the program window does not launch, this may be due to the file permissions configuration of your individual Windows computer. Our program launches an API server (Flask) that connects the UI to the SQLite database. Depending on the individual Windows version your computer runs on, Windows may or may not give the user permissions automatically to simultaneously launch the server alongside the front end.

There are a variety of fixes:

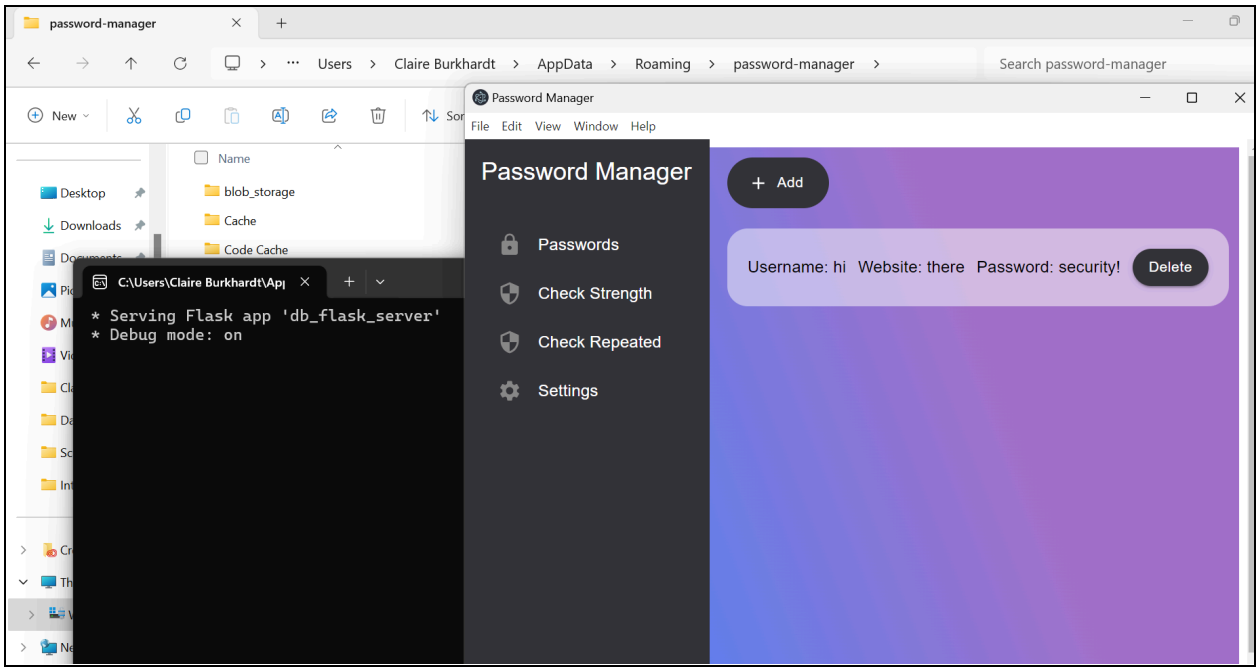
1. **Make sure you are running the password-manager program as administrator.** This should ensure all components have the correct permissions to launch
2. **Manually launch the Flask Server executable stored in App Data.**
    a. Navigate to

`:C:Users/<*YourUserAccount*>/AppData/Roaming/password-manager/db_flask_server.exe`

    b. Double click on the db_flask_server.exe file. While this server is running, re-launch the password-manager desktop application. The password manager should launch!



`:C:Users/<YourUserAccount>/AppData/Roaming/password-manager/db_flask_server.exe`



Success!

3. If your password manager still does not launch out, **please reach out to our friendly and accessible support team!** You can reach us at claire.burkhardt@emory.edu and @michi.okahata@emory.edu. We provide speedy customer service and real-time debugging :)

Terminal-Based Launch & Build Instructions

While this is documented here for the sake of comprehensiveness, ***please refer to the READMEs of the appropriate github branch.*** *Those are better formatted and include better problem resolution strategies. Windows | Mac*

1. Clone the Repository (link to Github Repo: https://github.com/cburkhardt27/password-manager-cs-370.git)
    a. Option 1: Clone this GitHub repository and checkout the proper branch:
        i. git clone https://github.com/cburkhardt27/password-manager-cs-370.git
    b. Windows (link to branch: https://github.com/cburkhardt27/password-manager-cs-370/tree/windows)
        i. git checkout windows
    c. Mac (link to branch: https://github.com/cburkhardt27/password-manager-cs-370/tree/mac_download)

        i. git checkout mac_download
    d. Option 2: Download as a zip file
    e. Download this repo as a zip file and extract to desired location

2. Setup your Python Environment
    a. Windows
        i. python -m venv win_venv
        ii. win_venv\Scripts\activate
        iii. pip install -r requirements.txt
    b. Mac
        i. python3 -m venv mac_venv
        ii. source mac_venv/bin/activate
        iii. pip3 install -r requirements.txt

3. Install Front-End Dependencies
    a. npm install

4. Package the Application Pages and Main Process
    a. Windows
        i. npm run pack:r
        ii. npm run pack:m
    b. Mac
        i. npm pack:r
        ii. npm pack:m

5. Start the Application
    a. Windows
        i. npm run start:e
    b. Mac
        i. npm start:e

6. Build the Application for Desktop (applies to Windows only)
    a. npm run make

Advanced Instructions / Installation

Build and Setup Scripts: Scripts are managed through package.json, which uses Electron Forge for building the desktop application.
To view other build/setup scripts: `npm run`
These include:

`build:dev` & `start:dev`: Development environment for React/Electron renderer. Not recommended for pages interacting with the backend.
`start:e`: Starts Electron.
`pack:m`: Webpack for main.js and preload.js. Outputs to the pack folder.
`pack:r`: Webpack for React/renderer pages. Outputs to the pack folder.
`package`: Creates an Electron executable.
`make`: Builds the installer.

Technical Details

Frontend: user interface built with React and Electron to run as a desktop application

- React: Provides responsive and interactive user interface
- Electron: Wraps the frontend in a desktop application environment
- Webpack: bundles the React code for production

Backend: Python based backend that manages logic and interacts with database

- Python: Handles backend logic including encryption, password management, and API endpoints
- Flask: Provides API layer for frontend-backend communication

Database: SQLite database that stores user data

- SQLite: stores credentials and encrypted passwords securely

Library Dependencies

Encryption:

- Bcrypt, string, secrets, Crypto.Cipher, base64, hashlib, pathlib, os.

SQLite Database:

- Sqlite3, flask, os, request, json, jsonify, flask-cors.

Frontend:

- React, MUI(material, icons-material, system), PasswordItem, react-router-dom, electron (Node.js), NPM, webpack, styled-components.

API Documentation

validate_master_password(): User authentication via master password

- When the user first downloads the application and it, they must create a username and master password to gain access. This username and master password is stored in a master password table in the database. Subsequently, whenever the user wants to login into the app, they have to input their login information. This verifies the identity of the user and allows the user to utilize the other app functions.

  Example usage:

  Input:

Username: username
Master Password: mp
Outcome: successful login

add_password_entry(): Add a new password entry to the database

- This app feature takes input from the user–username, url, and passwords–and then inputs that information into the vault as a password entry. The app encrypts the password and then inserts the encrypted password along with the url and username into an already initialized table(solely meant for password storage) in the database/vault. Then the user receives a password stored successfully message.

  Example usage:
  Input:
  Username: username
  Password: password
  Url: website.com
  Output: Password stored successfully

get_password(): Retrieve stored passwords from the vault

- This feature takes input from the user (a url), retrieves the password associated with that input, and displays that password and its corresponding username to the user. This function retrieves the encrypted password and username, associated with the url, decrypts the password, and returns the decrypted password and username. If the password entry does not exist then a password not found error message is shown to the user.

  Example usage:
  Input:
  Url: website.com
  Output:
  Username: username
  Password: password

delete_password(): Delete password entry

- This feature allows the user to delete a password entry. When prompted, the user inputs the url and username of the password they want to delete. The password entry is then deleted from the vault and the user receives a prompt that the password was successfully deleted. If the password entry was not originally in the vault, then the user will receive an error message.

  Example usage:
  Input:
  Url: website.com
  Username: username

Output: Password successfully deleted

display_all_passwords(): Retrieves all passwords, usernames, urls in database

- This feature allows a user to see all their passwords they have stored. When called, the function retrieves the username and password associated with each url in the database.

  Example usage:
  Inputs: None
  Outputs:
  Url1: website1.com
  Username1: username1
  Password1: password1
  ... for all stored data

get_repeated_passwords(): Retrieves passwords that have been used multiple times across different urls in the database

- This feature is used to see which passwords are being used multiple times across different urls. This is a security feature that allows users to see when a password is being overused and thus at higher risk.

  Example usage:
  Inputs: None
  Outputs:
  Url1: website1.com
  Username1: username1
  Password1: password1
  ... for all repeated passwords

generate_random_password(): creates a random password

- This feature allows a user to create a random password. It generates a password of length 12 made up of a random assortment of letters, numbers, and punctuation (special characters).

  Example usage:
  Inputs: None
  Outputs:
  Password: Somevalidpassword123

Tech Documentation: Application Architecture & Methods

SQLite Database Functions:

Connect_db: This function takes the input DB_NAME and uses the sqlite3 function, connect, to connect to a sqlite database with the name DB_NAME. If this database does not exist, then one is created with the name DB_NAME. The established connection generates a connection object. That connect object can be used to create a cursor object. Both objects are returned The connection and cursor object will be necessary when executing database queries.

Create_password_table: This function takes the connection object as an input. First a query is used to create a table to hold password entries.The table has columns for an id, usernames, urls, passwords, and timestamps for when the entry is created and updated. A try and except block is used. The try block checks if the connection is established; if the connection is established, the query is executed using the cursor. In the except block, any errors are caught and printed. Lastly the cursor is closed.

Create_master_password_table: This function takes the connection object as an input. First a query is used to create a table to hold the master password. The table has a column for the username and the hashed master password. A try and except block is used. The try block checks if the connection is established; if the connection is established, the query is executed using the cursor. In the except block, any errors are caught and printed. Lastly the cursor is closed.

Add_master_password: This function initializes a hashed master password and username variables using the setup_user_master_pass() function from the encryption functions file. An insert query to insert into the master password table is made. First a connection is made using connect_db. Then an if statement is used to check if the connection is properly established; if the connection isn't properly established then the function returns. A try and except block is used. Within the try block, the query is executed with the hashed master password and username inputs using the cursor object. The except block catches any errors. Lastly the connection and the cursor is closed.

Get_master_password: This function first creates a retrieve query. Then a connection is made to the database. A try and except block is used. Within the try block, the cursor object is used to execute the retrieve query. Cur.fetchone used to assign the retrieved information to a result variable. If the variable has no value then the master password table is empty and false is returned. The except query is used to catch any errors Then the cursor and connection is closed and the username and hashed master password is returned.

Add_password_entry: This function takes username, url, and password strings as inputs. First this function gets the master password and username and uses the username to encrypt the password input. A check query is made to check if the password entry input is already in the vault. An insert query is made to insert the input data into the password table. Then a connection is made to the database and connection and cursor objects are created. This function uses a try

and except block. Within the try block, the check query is executed. If the input password entry already exists in the database then function is returned. Next, the insert query is executed using the cursor object. The except block is used to catch any errors and then the cursor and connection object is closed.

Get_password: This function takes a url as an input. A select query is made to retrieve the password and username associated with the given url. A connection is established to the database;if the connection was not established, the function is returned. A try and except block is used. Within the try block, the query is executed using the cursor object. Cur.fetchone is used to assign the retrieved data to a result variable. If the result is not empty, the password, which is encrypted, is decrypted using decode_vault_password function from the encryption functions file. The get master password function is required to use the decode vault password function. Then the decrypted password and username are decrypted. If the result variable was empty, an error statement is printed and the function is returned. The except block catches any other errors. Lastly, the connection and cursor objects are closed.

Delete_password: This function takes a username and url as inputs. Then a query to delete the password entry at the input username and url is created. A connection is established to the database;if the connection was not established, the function is returned. A try and except block is used. Within the try block, the query is executed using the cursor object. The except block catches any errors and then the cursor and connection objects are closed.

Display_all_passwords: This function first makes a query to retrieve all the passwords and their associated usernames from the password tables. A connection is established to the database;if the connection was not established, the function is returned. A try and except block is used. Within the try block, the query is executed using the cursor object. Cur.fetchone is used to assign the retrieved data, which is all the passwords in a list, to a result variable. For each password in the list, the decode_vault_password function from the encryption functions file is used to decrypt the password. The get master password function is required to use the decode vault password function. The decrypted password and its associated username are added to another list. Then this list is returned. If the result variable was empty, an error message is printed and the function is returned. The except block catches any errors and then the cursor and connection objects are closed.

Get_repeated_passwords: this function returns every password, and its associated username, that is repeated. A try and except block is used. Within the try block, display all password functions are called which returns a list with every decrypted password and its username. The first for loop uses a count list variable to keep count of the presence of each password. The second for loop uses that count list and for each count that is greater than one, the corresponding password entry is appended to a new list variable. If this list variable is empty, a no repeated

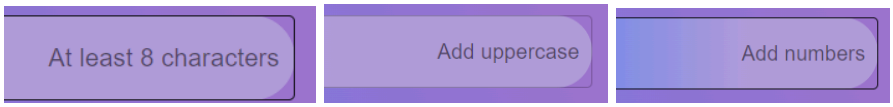
passwords message is printed and the function is returned. Else, the list is returned. The except block catches any errors.

Flask Endpoints:

Flask endpoints are defined for each of the SQLite database functions defined above with a corresponding method specification to GET, POST, or DELETE. These endpoints are called from the Electron application with Axios, an HTTP client.

CheckUp Features:

CheckUp, CheckUpStrength, and CheckUpUnique are check up functions that are utilized by the frontend of the program. When a user initially creates their account, generating a username and master_password, the master_password is tested against the checkup strength function to make sure the password is strong. Strong passwords are passwords that have at least 8 characters, an uppercase and a lowercase letter, a number, and a special character and unique passwords are passwords that have not been used previously by the user. These functions also test each password entry, for strength and uniqueness, inputted by the user. The following prompts are shown when entering a password.



CheckUpStrength utilizes the display_all_passwords api and specifically checks for certain features (length, capitalization of letter characters, numbers, special characters, etc) of password input to make sure they are strong. CheckUpUnique utilizes the get_repeated_passwords api to find repeated passwords. There is a check strength page, that shows any weak passwords, and a check repeated page, that shows any repeated password, on the program.

Cryptography:

The Cryptography is implemented using specialized Python libraries. Using bcrypt, the program hashes a secure (8+ characters, upper and lower-case, special characters, and numbers) master password that the user creates during the setup phase of the program. This hash is a slow-encryption hash that neutralizes common intrusion techniques like rainbow tables. This hash is used to validate any access to the vault: the database of encrypted passwords. Access will not be provided if the user cannot submit a master password whose hash matches the hash of the actual master password. The master password hash is also used alongside PBKDF2—a key derivation function—and AES—an encryption algorithm—to securely encrypt the vault's sub-passwords. These sub-password are stored in the database in an encrypted state for security.

They are then decrypted upon user retrieval if and only if the master password the user provides is identical to the one stored (via hashing, not in plaintext) during the setup phase. Thus, this double hashing/encryption allows for secure and best-practice storage of sensitive passwords.

Github Version Control

Please reference our main repository here:
https://github.com/cburkhardt27/password-manager-cs-370

Other branches on the above repo represent prior iterations of the codebase. We started developing in brain but quickly branched into `build-new`. Then we refactored our database into sqlite, prompting the sqlite branch and eventually sqlite-new. We added another-one during integration. When we decided to redo the front end, this got hosted on a spike in this github repository (https://github.com/michi-okahata/toy-password-manager). We integrated those changes back into this repo under windows and mac_download. In an attempt at cleaning up a now-out-of-date main page, we decided to make the main branch our documentation page..

Future Work

Front End Feature update:

While the essential front end features complete, there are some advanced UI improvements we'd like to implement. These include `update-username` and `update-password` functions, which would give the user more flexibility in managing their passwords. Update username would take a url and two usernames, the original username and the new username, as the input. The associated password entry would be retrieved and the original username would be changed to the new username. Update password would take a url, a username, and new password as the input. The associated password entry would be retrieved, the password would be decrypted then changed to the new password, and then the password would be encrypted. We also want to add a `generated-random-password-feature` button to the user interface when creating new passwords. This function was implemented in the API but has not yet been added to the front end. This will allow users to generate a random password whenever they want to input a password entry. Lastly, during the Project Fair, users recommended hiding the plaintext passwords that are displayed in the front-end after they are decrypted. This is a good suggestion for privacy reasons; we'd like to hide the actual password from the screen unless the user hovers or clicks on the password box.

Windows Exe packaging update:

We'd like to continue improving the smoothness of installation and packaging. In particular, we'd refactor our database into a Node.js-native architecture that would allow us to avoid potential bugs with the Flask server and IPC setup. This would simplify the installation pipeline and reduce the likelihood of user errors.

MacOS DMG packaging and IPC Updates:

Refactoring our database would also simplify MacOS packaging. While we can successfully generate a MacOS .dmg file, the packaged version of the program doesn't properly connect with the Flask server. Thus it cannot access the database. In the Mac distribution of this project, the Flask API does not always launch in-sync with the front end. Different versions of main.js (which manages simultaneous launch/setup of Flask and Electron and the IPC between them) work for different developers MacOS computers. This was frustrating, because it meant that we'd successfully make the project run on one MacOS machine, then realize we'd introduced bugs onto another developer's machine.

Future work will include fixing this particular bug so that all components work smoothly alongside each other when packaged. More specifically, we would choose to manage more of the stack locally within the node.js framework by saving passwords within node.js rather than the SQlite database. This would make our IPC integration with Flask API more straightforward. Thus, rather than working across four independent development tools, we would simplify our stack. This would allow more streamlined development and bug-fixing when generating our .dmg Mac file.