CS 6303 Cyber Security Essentials for Practitioners

Final Project Deliverable 2

WiseLock App | Elliana Brown

Introduction

As digital technology becomes increasingly integrated into daily life, the need for effective cybersecurity practices grows. Password managers, which securely store and generate unique passwords, are widely recognized as vital tools for protecting personal information. However, older adults—often a vulnerable demographic—remain hesitant to adopt these solutions. This reluctance stems from factors such as limited exposure to technology, a lack of confidence in using digital tools, and fears surrounding the security of their sensitive information. Unfortunately, these barriers leave older individuals more susceptible to cyberattacks, such as phishing and credential theft, which can have devastating financial and emotional consequences. While existing password managers provide robust security features, they often fail to address the unique challenges and concerns of older adults. This project aims to design and implement a password manager tailored to this demographic, focusing on accessibility, ease of use, and transparency to foster trust and confidence. By prioritizing user-centered design principles, this tool seeks to empower older adults with a sense of control over their online safety and help bridge the gap between cybersecurity needs and technological adoption.

Background and Related Work

Previous research has explored the barriers older adults face when using password managers. Ray, Wolf, Kuber, and Aviv's study, *Why older adults (Don't) use password managers*, highlights the psychological and technical challenges that discourage adoption, including perceived complexity, distrust in automation, and fears of data breaches. Despite these reservations, their findings reveal that older adults are aware of the importance of strong passwords but often resort to unsafe practices, such as reusing passwords or writing them down. Other studies emphasize the cognitive load required to manage multiple complex passwords, particularly for older individuals who may already experience memory-related challenges. Existing password managers, while effective in enhancing security,

are frequently designed with tech-savvy users in mind, overlooking the specific needs of less experienced demographics. For example, interfaces that rely on jargon, minimal guidance, or excessive customization options can deter older users from engaging with the technology. My project seeks to build on these insights by creating a password manager that addresses these gaps, incorporating user-friendly design elements and educational features tailored to older adults. By combining the foundational security principles of existing password managers with a design philosophy focused on simplicity and reassurance, this tool aspires to make cybersecurity accessible to a broader audience.

Implementation

Backend

This project is built using Django, a Python web framework that simplifies the creation of robust and scalable web applications. Django is known for its many useful features like user authentication, database integration, and URL routing that are already included. For this password manager application, Django serves as the backend, managing user accounts and storing password data securely. The framework makes it easy to set up a secure system where users can register, log in, and manage their passwords through RESTful API endpoints. These endpoints are the communication bridges that allow the backend to interact with a frontend application or mobile app.

In this application, the custom user model extends Django's default user system to allow future flexibility, such as adding unique fields to users later on. A second database table is introduced to store passwords, each linked to a user account. For example, if a user adds a password to their list, the backend ensures it's saved under their account and not accessible to other users. These relationships between users and their stored passwords are defined using models. A model in Django is a blueprint for the database, making it straightforward to save, retrieve, and manipulate data.

The application also uses Django REST Framework (DRF), which makes building APIs simple and consistent. DRF takes care of many tedious details, like converting Python objects into JSON, a format that is easy for other systems,

such as the frontend, to understand. For example, when a user logs in, the backend checks their credentials and returns a JSON response with a special token. This token acts like a key that the user can use to prove they are logged in whenever they interact with the API.

The app includes several API endpoints for specific tasks. For instance, there is an endpoint for user registration (`/signup`) where new users can create an account by providing their username, email, and password. Another endpoint, `/login`, allows users to log in and get their token for accessing the app. For password management, there are endpoints to view a list of saved passwords, add a new one, or update an existing password. Each of these endpoints is secured, meaning only authenticated users with valid tokens can use them.

To ensure a smooth experience for the user, the project also includes serializers. Think of serializers as translators that convert complex Python objects into simpler formats, like JSON, for the frontend to use. For example, when a user requests their list of passwords, the backend uses a serializer to neatly format all the password information before sending it to the user. Conversely, when the user sends new data (like a new password to save), the serializer ensures the input meets all the app's requirements before saving it.

Finally, the app uses middleware called `CORS` (Cross-Origin Resource Sharing) to allow the frontend and backend to communicate even if they’re hosted on different servers. This is essential for real-world applications where the user interface and backend often live in separate environments.

In short, this application uses Django’s powerful tools to handle the complex logic of managing users and passwords securely. With features like REST API endpoints, custom user models, and password storage, it’s designed to work seamlessly with a frontend, providing users with an easy-to-use and secure password management experience.

Frontend

React Native is a framework that allows developers to build mobile applications using JavaScript and React. What makes it special is its ability to create apps that feel truly native on both Android and iOS without needing to write separate code for each platform. Instead of creating a mobile website, React Native

generates actual native app components, providing a smoother, faster experience for users. It's an efficient way to build apps because it allows developers to reuse code across platforms, speeding up the development process.

In this app, React Native is used to create a password management tool that helps users log in, sign up, generate strong passwords, and organize them with labels. At the heart of the app is a file called `RootLayout`. This file sets up the navigation system using a tool called React Navigation. React Navigation organizes the app into screens, such as Login, Signup, Home, Password Generator, and Password List. It uses a "stack navigator," which essentially creates a stack of screens that users can move through, with the ability to go back to a previous screen if needed. Each screen is represented as a React component, which keeps the app modular and easy to manage.

The Login screen is where users start by entering their email and password to access the app. This screen validates the inputs, ensuring that the email looks like a real one and that the fields aren't empty. If there's an issue, like invalid credentials, a pop-up modal displays an error message. Once users successfully log in, they're directed to the Home screen. If they don't have an account yet, they can navigate to the Signup screen, where they can create one by entering their details. The app checks that passwords match, meet minimum security requirements, and that the email is valid before allowing them to proceed. Once their account is created, they're redirected to the Login screen.

The Home screen acts as the hub of the app, welcoming users and giving them two options: they can either generate a new password or view their saved passwords. If users choose to generate a password, they navigate to the Password Generator screen. This screen allows users to enter a label, like "Email" or "Bank Account," and click a button to generate a strong, random password. The app connects to a backend server to create the password and displays it in a pop-up along with the label. This ensures the generated password is secure and unique. Users can then save the password to their account for future use.

If users want to review their saved passwords, they can go to the Password List screen. This screen fetches all saved passwords from the backend server and displays them in a list, showing both the label and the password. The list is dynamically updated whenever new passwords are added, so users always see

the most recent data. This feature is designed to make password retrieval simple and convenient.

The app is styled to be clean and user-friendly, with input fields that are easy to type in and buttons that clearly guide users through each task. Pop-up modals provide feedback, such as showing errors when something goes wrong or displaying generated passwords. Overall, the app uses React Native's capabilities to create a seamless experience, making it a practical tool for managing passwords securely. By combining React Native with backend functionality, this app ensures both a polished user interface and robust performance behind the scenes.

Security Principles Addressed

The key functions implemented in this app focus heavily on ensuring the security and integrity of user data, particularly sensitive information like login credentials and generated passwords. These functions handle everything from validating user input to storing passwords securely, addressing multiple aspects of security in a mobile application. Let's break down each function and its role in enhancing the overall security of the app.

Verifying User Input

The function responsible for verifying user input plays a critical role in preventing malicious or erroneous data from entering the system. This function ensures that the inputs, particularly strings such as email addresses and passwords, adhere to expected formats and constraints before being processed further. For example, when users input their email address, the function checks whether it follows a valid pattern (using regular expressions to ensure that it looks like a proper email). This reduces the likelihood of invalid or malicious data being submitted, which could otherwise lead to errors or security vulnerabilities like SQL injection or malformed data being sent to the backend. Similarly, the password verification process could ensure that passwords meet a minimum length requirement and contain a mix of characters, enhancing password strength and reducing the likelihood of weak passwords being chosen. In this case, input validation works as an initial line of defense by rejecting weak or improperly formatted data early on.

Checking Login Information Against Saved User Database

Once the user's email and password are verified locally for correctness, the system then checks the credentials against the saved users database. This step involves securely retrieving the stored credentials (typically hashed and salted passwords) and comparing them with the provided login information. Instead of storing plain text passwords, which would pose a significant security risk, the app likely stores the hashed versions of passwords. Hashing is a process where the password is transformed into a fixed-length string that cannot be reversed to retrieve the original password, even if the hash is compromised. When the app checks the provided email and password against the database, it doesn't compare the actual password but rather compares the hash of the provided password with the stored hash, adding a layer of security.

Generating New Password

Password generation in the app is another crucial security function. When the user requests a new password, the app generates a strong, random password designed to be difficult to guess or crack. This process likely uses a cryptographically secure method, ensuring the passwords are sufficiently random and resistant to attacks like brute force or dictionary attacks. Rather than relying on easily guessable words or patterns, the password generator uses a mix of letters, numbers, and special characters to produce passwords with high entropy, making them more secure. After the password is generated, it's hashed before being stored, ensuring that even if an attacker gains access to the app's database, they cannot retrieve the actual password. Hashing the generated password, just like with user login credentials, ensures that only the hashed version is saved, minimizing exposure.

Saving Passwords Locally

Once a new password is generated, it is saved locally within the app. This function stores passwords in a secure local storage mechanism, preventing unauthorized access to the password data on the user's device. It's important that the app encrypts this data before saving it locally to prevent local attacks, where someone with access to the physical device might try to extract stored

passwords. Storing passwords locally also enhances user experience by allowing easy retrieval without needing to re-enter login information.

Updating Password Option

The ability to update passwords securely is a vital feature in any password management app. When a user decides to change their stored password, the app must ensure that the new password is both valid and securely stored. After a user inputs their new password, the system must hash and securely update the saved password in both the local storage and on any server-side databases where it might be saved. The app could prompt the user for their current password to confirm their identity before allowing the change. This step prevents unauthorized users from modifying passwords if they gain temporary access to the app. Just like the initial password generation, the app ensures that the new password is hashed before being saved, maintaining security best practices throughout the app. By updating passwords using hashed values, the system guarantees that even if the local or server-side storage is breached, the actual password is never exposed in plain text.

Overall Security Principles

Together, these functions address several key principles of security: **data validation, encryption, hash security, and access control**. By validating inputs before processing, the app minimizes the risk of bad data or attacks like SQL injection. The use of hashing and salting ensures that sensitive user data, such as login credentials and passwords, is stored securely, even in the event of a data breach. Password generation techniques also play an important role in creating strong, unpredictable passwords that make it difficult for attackers to guess or crack them. Storing and retrieving passwords securely from local storage ensures that user data is protected from unauthorized access, and the ability to update passwords securely prevents attackers from changing credentials without proper verification. Together, these functions ensure that the app follows best practices for managing user data securely, creating a trusted environment for users to store and manage their sensitive information.

Results

The connection between the backend and frontend of my project is currently not functioning as expected. This issue could stem from several potential causes. First, there may be a mismatch in the endpoints or URLs being used by the frontend to communicate with the backend. If the frontend is pointing to an incorrect or nonexistent API endpoint, requests will fail. Another possible issue is the lack of proper CORS (Cross-Origin Resource Sharing) configuration on the backend, which can prevent the frontend from accessing backend resources when running on different servers or ports. Additionally, if the backend is not correctly handling requests (such as missing API routes or server-side errors), the frontend will not receive the expected responses. Lastly, network issues or incorrect authentication tokens being passed between the frontend and backend could be causing the problem. I've tried debugging the API calls, checking the network logs, and ensuring both systems are properly configured to help identify the root cause of the disconnect but nothing seems to be working.

Conclusion and Future Work

In conclusion, while the project demonstrates the core functionality for password generation, user authentication, and local storage management, there are still key areas that need further refinement. The backend and frontend are currently not communicating as expected, and resolving this issue will be crucial for the seamless operation of the application. Additionally, ensuring secure password handling, implementing better error handling, and improving user experience are essential steps for strengthening the overall application.

For future work, I plan to address the connectivity issue between the frontend and backend by troubleshooting API calls, verifying correct URL routes, and resolving any potential CORS issues. Further testing and validation of user input, password generation, and storage will be performed to ensure robust security. Moreover, integrating user feedback and improving the UI/UX design will enhance the overall user experience. Long-term goals include adding features like password encryption, improving the mobile interface, and expanding the app to handle more complex password management tasks, such as categorizing passwords and syncing across devices.

Citations

H. Ray, F. Wolf, R. Kuber, and A. J. Aviv, "Why older adults (don’t) use password managers," *Proc. USENIX Security*, 2021.