HashPass: Password Manager Chrome Extension

ECE 49595 Final Proposal (Spring 2025)

Shivam Sharma sharm572@purdue.edu

Xavier Callait xcallait@purdue.edu

Samarth Girish girishs@purdue.edu

HashPass Repository

Product Description

Our project is a Chrome extension aimed at addressing a significant gap in online security: the difficulty of managing secure, unique passwords across countless websites without requiring users to remember multiple complex credentials. While there are many password managers available, our extension goes beyond simple storage by generating site-specific passwords dynamically, reducing reliance on stored data, and adding unique security elements. This innovation is particularly valuable as it allows users to maintain secure, complex login information without the cognitive load associated with traditional password managers, enhancing both security and user convenience.

The extension simplifies password management by letting users create and remember just one easy-to-remember password. Instead of storing complex passwords, it dynamically generates secure, unique credentials for each website by combining the user's chosen password, personal identifiers like email addresses, and truly random numbers generated by the CPU. This combination forms a unique password chain, which is then hashed with advanced cryptographic algorithms. As a result, each generated password is specific to the website and nearly impossible to replicate, offering robust protection against password theft and cross-site vulnerabilities.

To achieve this level of security, our project leverages AWS cloud services for information storage. AWS RDS securely stores non-password-related user information such as email addresses, usernames, domain names, and phone numbers. The stored data will all be encrypted using standard AES in order to protect the data in case of a data compromise. Our compute function, hosted on AWS Lambda, retrieves truly random numbers, hashes the password chain, and manages all API requests. These requests are routed through AWS API Gateway, ensuring secure interactions with external sources and shielding the backend from unauthorized access. This architecture provides the reliability, scalability, and security required for our extension's seamless functionality.

Despite the complexity behind the scenes, the user interface is designed for simplicity. When users log in, they enter their memorable password into a browser-based extension pop-up, which instantly regenerates the unique hashed password for that specific site. The generated password is never stored, further minimizing the risk of exposure. This simplicity in UI design means users experience a streamlined login process without the technical details, while our extension manages security in the background.

Additionally, to address recovery needs and enhance security, we plan to integrate two-factor authentication (2FA) as a feature for passkey recovery. 2FA will operate through an SMS-based verification method. Upon initial setup, users will link their phone number or preferred authentication app to the extension. If the user has forgotten their passkey, the 2FA will verify the user’s identity upon password reset.

This adds another level of complexity to the UI of the application. A new component called “SMS Verification” would need to be included. This will open a new window on the user’s phone and communicate with the popup component on the website verifying that the correct user is attempting to access the password.

By merging the simplicity of a single memorable password with advanced cryptographic security and innovative environmental randomness, our project aims to provide an easy-to-use, secure, and reliable password management experience. This extension enables users to enjoy a safer online presence without the burden of remembering complex passwords, setting a new standard in password management by combining effortless protection with robust security.

Competitive Analysis

Most password managers, such as LastPass and 1Password, rely heavily on auto-fill vaults, which store and manage passwords in encrypted databases. Users access these vaults manually or through browser integrations, where saved passwords are automatically filled into login fields. Our password manager addresses these limitations by offering popup-based login handling, engaging only when needed, without constant vault access, reducing background processes and improving speed. Additionally, it introduces enhanced password transformation, dynamically hashing weak or reused passwords during login. Additionally, the addition of 2-Factor Authentication for passkey recovery will provide another layer of security in the event that the plain-text password has been compromised. The product will also ensure that even temporary session data and hashes are erased post-login, enhancing privacy in a way that the competitors do not. By streamlining workflows and enhancing security in real-time, our password manager addresses the common frustrations users face with vault-heavy tools while filling critical gaps in the market.

A notable competitor which is not vault based is the Master Password algorithm, developed by Maarten Billemont called Spectre App. Unlike traditional password managers that store passwords in a vault, Master Password generates unique, site-specific passwords on-the-fly using a combination of the user's master password, their name, and the site's domain name. This approach eliminates the need for password storage, as each password is deterministically recreated when needed. The algorithm employs cryptographic techniques to ensure that the generated passwords are both secure and unique to each site. By not storing passwords, Master Password reduces the risk associated with data breaches and offers a stateless solution to password management. While our solution also does not store passwords on the fly, the difference comes from the uniqueness in how our random passwords are generated. Additionally, the incorporation into the browser as an extension along with the simple to use set up process, our system has an edge over the competition.

Project Scope and Viability

According to the 2009 paper by Benjamin Strahs from the University of William and Mary, the steps in our approach have scientific viability. In the paper he discusses the need for randomness in the salts for password hashing. He discusses that many password agents use salts, however, they lack randomness and can easily be traced. In his solution to provide uniqueness, he rotates between a few salt repositories randomly to choose salts. In our solution, we plan to use user information, domain information, and true random numbers. This information allows us to create our own unique "salt repository".

Regarding encryption standards to protect user data, the data will be encrypted in transit using the standard encryption policies which exist within the https secure service. Any stored data for the simple "plain-text" password will be stored in a hashed format using the SHA-256 encryption protocol. In addition, we plan to follow all of Chrome's compliance standards. One of the main rules is to only use APIs which are made available by Google.

This project is a substantial engineering effort as it involves creating a browser-integrated password manager with advanced features, including popup-based login handling, real-time password hashing, and adaptive multi-factor authentication (MFA). The challenge lies in implementing secure client-server communication, integrating with AWS components for data management, and developing encryption protocols to protect user data. Additionally, the project requires designing a responsive interface and efficient backend infrastructure, making it a complex and appropriate endeavor for a senior design course.

The proposal is viable given the team's expertise in web development, cryptography, and cloud infrastructure. AWS components will simplify the management of user data, ensuring fast retrieval and scalability. The modular design of the system allows for incremental development and testing, reducing the risk of integration issues. With a clear timeline that includes milestones such as frontend design, browser extension development, and cloud deployment, the scope of work is manageable within a semester while offering significant technical depth.

Use Cases

a. Users will need to sign up for an account. The user will provide an email address, create a personal key for future sign-ins, and set up two factor authentication if the key is forgotten.
b. Users will navigate to a website they do not have an account to. The user will be asked if they want to create an account for the website and if so, the program will create the account and hashed password for the domain.

c. Users will navigate to a website they do have an account to. The user will be prompted to enter their personal key. This key will be used to generate the hashed password and use the password to log the user in.
d. Users may not remember their personal key. The user will then be prompted to update the personal key using a two factor authentication mechanism.

Requirements

1. Each generated password must have a minimum of 60 bits of entropy for strength as outlined by NIST Guidelines.
2. Extension needs to determine anytime a password is needed on a webpage by scraping the web page's elements. Once a password input field element is detected, a popup will appear, prompting the user to use our password manager to generate a password for that site.
3. Data to be stored in AWS RDS must be encrypted using AES and must not be in plain text before transmitting.
4. A user can disable our password manager for any given website, such that our extension will not pop up on sign-in pages and will not autofill the password fields for the specific site.
5. The passwords must never be saved, they will always be computed in live time within a 15 second window any time a sign in is required. The algorithm decrypts the encrypted salts which are stored and uses those to compute the hash.

Design

The context diagram in figure 1 demonstrates the interactions between the different entities involved in our product. It starts with a user who is using Google Chrome with our extension installed. Prior to use, the user must create an account with our product. This also is shown in the connection between the user and our extension in figure 1. Once this is complete, the user will use the Google browser as normal, and when our extension detects a sign-in or sign-up page, it will prompt the user for the base password. The inputs will then be sent to AWS for storing and computation, and the final enhanced password will be returned to the extension which will automatically populate the field in the browser for the user.

In figure 2, it shows the component diagram for our product. The user starts off by creating their account through our extension account handler which is displayed on a dashboard for our extension. The extension account handler will collect simple user information such as name, email, and it will also ask security questions. These security questions are a part of the

user authentication mechanism in figure 2, along with other methods of authentication for the user such as MFA should it be implemented. All of this information is sent to AWS for it to be inserted into the password data database by our Lambda functions. From here, when the user enters a website, our extension will check if the user has an account with that site. This process is encapsulated by the user account handler as seen in figure 2. If they do not have an account, then our extension will have a pop-up for the password creation. The user will enter a base password into the dialogue box. This will be sent to our AWS infrastructure which will use the user's account information to create a more secure password through hashing and salts. This enhanced password is returned to the password store manager which communicates with our AWS infrastructure when a password needs to be fetched and enters the password into the website's password input. If the user already has an account, they simply need to enter the password into the pop up for our extension. The password store manager in figure 2 will send the base password to our AWS infrastructure which will reconstruct the enhanced password for the site. The password will be returned to the password store manager in the extension which will enter the password into the site for the user.

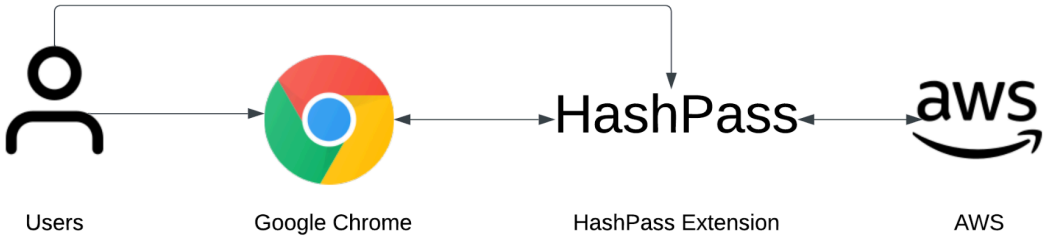

Figure 1. System context diagram showing the interactions between components.

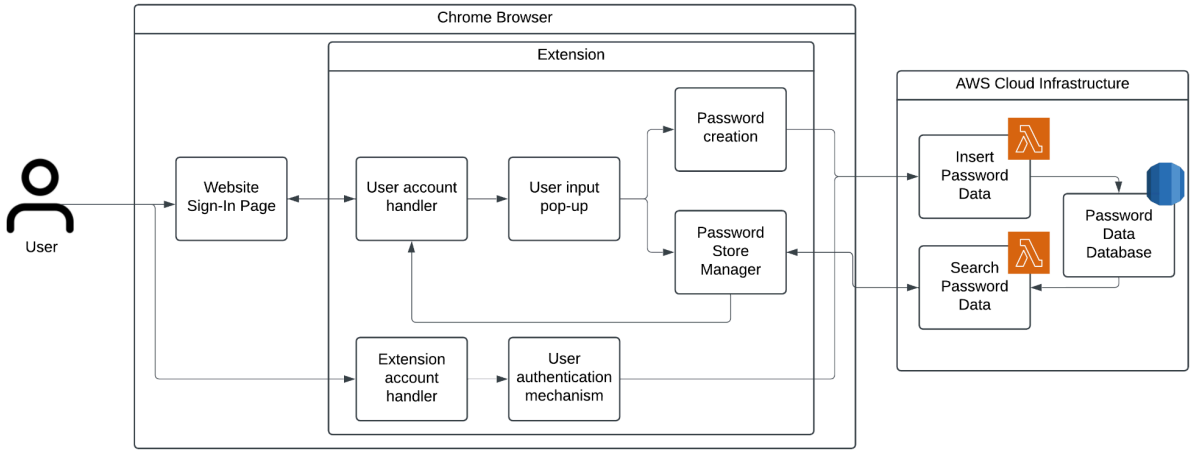

Figure 2. Component diagram depicting the components of HashPass.

Libraries, Tools, and Frameworks

Library/Tool/Framework	Purpose	Justification
Javascript	Language for development of the Chrome Extension	Javascript is one of the only languages that has built-in support for the Chrome Extension API. Thus, all development will need to be done in Javascript.
React	The library used to create the User Interface	React will provide a way to create a simple User Interface for the extension.
Argon2	Password hashing function	A secure hashing algorithm designed for passwords. Hashing passwords will increase the security of our product which is the overall goal.
AWS RDS	Relational database for user data and fields used for password generation	AWS RDS ensures each user's data is securely separated and easily managed through relational tables, enabling fast lookups.
AWS Lambda	Compute component used to handle API requests and logic for the password hashing and extension.	AWS has several different compute components available. However, Lambda is a stateless function which means it is only run when invoked by an event. This is suited towards API driven applications such as this.
AWS API Gateway	Routes and manages API requests to the AWS Lambda functions.	To invoke the Lambda from the Extension, API Gateway will allow for the program to access the computer and database easily. It also offers protections against malicious actors.
Axios	Javascript library for REST API requests	The Lambda function will need to use API calls. Axios is a popular Javascript library that handles API calls.
Jest	Javascript testing framework	We want to have unit and tests for our product to ensure that it works as intended.

References

https://blog.lastpass.com/posts/top-personal-and-business-features-in-password-manager

https://aws.amazon.com/rds/features/

https://aws.amazon.com/lambda/

https://aws.amazon.com/api-gateway/

https://developer.chrome.com/docs/extensions/develop

https://engineering.purdue.edu/kak/compsec/NewLectures/Lecture15.pdf

https://www.weather.gov/documentation/services-web-api

https://axios-http.com/docs/api_intro

https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=9e32f5b848bddcfba7d24fa4f0f6c28e2b0c0da0

https://agupubs.onlinelibrary.wiley.com/doi/full/10.1029/2023CN000228

https://spectre.app/

https://www.npmjs.com/package/argon2