

TAPAS 2.0

PROJECT REPORT

Ahmed Qarmout
Darshan Mehta
Raj Pandya

CSC490

University of Toronto - Mississauga

April 8th, 2019

Code source: <https://mcscsm.utm.utoronto.ca/qarmouta/tapas-2.0>

You can find our Keynote presentation here: <https://youtu.be/pqC3LXqfPJU>

ABSTRACT

It is inevitable that in this digital era, we register for online services, subscriptions, and accounts for various purposes. Our accounts contain sensitive information and the most we can do to protect it is make sure that it has a strong password. For the rest of the story, we place our trust in the company to use proper encryption tactics and security methodologies. Despite this, companies of all sizes face hackers and our credentials are stolen, sometimes unencrypted versions of it. Therefore, it is crucial that people take a second and invest some time in properly securing their information themselves. There has been a massive push for two-factor authentication recently and that is a great step in verifying logins. Unfortunately, with tools such as LastPass and 1Password, users passwords are stored offline on all of their devices and online in the cloud. With access to one device, it is easy for an attacker to login to various accounts and obtain sensitive information. With Tapas 2.0 and it's dual possession authentication structure, we bring the same idea of 2FA to storing and retrieving your passwords. When a user wants to login at a specific form, they can click on “Get Site Credentials” on Tapas 2.0 Google Chrome extension and then confirm on their phone if they'd like to send over credentials and for which account. The passwords will auto fill into the form and the user can proceed to log in. When a user wants to register for a new website, Tapas 2.0 will suggest a strong, 14-character alphanumeric password with special symbols and the login information will be encrypted and stored onto the phone. Currently, Tapas 2.0 is only for browser-based logins; however, the development team has set goals to bring dual-possession authentication for mobile logins soon.

To build this tool, we've built a Google Chrome extension, a rendezvous server, and a phone application built on React Native. Once the user registers for an account, Tapas will generate multiple encryption keys based on the password of the users account. These keys will be used to encrypt all communication through the server and Firebase, to encrypt the passwords, and to encrypt all backups of the passwords in case of lost or stolen devices. And to make sure Tapas works best for you, it will recognize when you've created a new account and when you've updated a password. It also comes with context menu options and Device Management page that are all built to help the user stay in control of their accounts security.

TABLE OF CONTENTS

ABSTRACT	1
INTRODUCTION	3
BACKGROUND AND RELATED WORK	6
METHODOLOGY AND DESIGN GOALS	9
Web Browser Extension:	9
Rendezvous Server:	11
Phone Application:	11
TOOLS AND LIBRARIES USED	12
Browser Extension:	12
Rendezvous Server:	14
Phone Application:	16
PROJECT OVERVIEW	18
Section 1: Design in Detail	18
Section 1.1: Creating an account and key generation:	18
Section 1.2: Pairing devices	21
Section 1.3: Saving credentials	23
Section 1.4: Retrieving Credentials	26
Section 1.5: Importing passwords to Tapas 2.0	28
Section 1.6: Backup and Recovery of Wallet	28
Section 1.7: Additional Notes	29
Section 2: Developer Guidelines	30
Section 2.1: Chrome Extension	30
Section 2.2: Server	31
Section 2.3: Phone Application	31
Section 3: Source File Breakdown	35
Section 4: End User Tutorial	37
EVALUATION AND LESSONS LEARNED	38
Technical and Security Evaluation	38
Timeline and Division of Responsibilities	41
Lessons Learned	41
Recommendations to Future Students	42
CONCLUDING REMARKS	43
FUTURE WORK	45
APPENDIX	47

INTRODUCTION

Currently, there are many password managers and two factor authentication methods that offer security or user-friendliness, but not both. With that, users are vulnerable at various points of attack and even the possibility of compromised data if a thief gets their hands on one device. We plan to focus on the security and ease of use aspect for laptop-based browser logins using dual-possession authentication with the user's primary smartphone. In our initial proposal, there was a plan to remove any master passwords, removing a vector of attack and not requiring a user to memorize a long password. Unfortunately, due to our implementation of recovery and backing up data, the users are required to register an account.

Nowadays, as many cyber-security experts summarize our online safety, it is not if we'll get attacked and our passwords will be stolen but when. Taking a look at SplashData's Top 100 Passwords list, we can quickly realize that people are either careless or unaware of the risks of weak passwords. With that in mind mixed with our passion for security and desire to learn how to develop extensions and apps, Tapas 2.0 came to be. Our goal was to provide users the ability to register for websites and login without ever having to know what their password is. There's no need for a sticky note on the desk with passwords written in pen, no need to memorize complicated passwords such as 0Bi\$-AzL\$-9@dE and absolutely no need to worry about security of the passwords.

Tapas 2.0 was very close to not being implemented because of the lack of interest from other classmates. In the end, we ended up reaching the minimum group size requirement of four and proceeded with the project. Personally, none of us use third-party password managers such as LastPass or 1Password because the sense of security they provide is weak. That's where our motivation to create a better tool came to be. From the start, thanks to consistent project planning meetings, we knew everyone who was part of this group was dedicated to making a powerful tool and that definitely helped us reach our goals. Unfortunately, one of our teammates had to leave the group but the remaining members were quick to pick up the unassigned tasks.

Other than Ahmed and his experience of creating and hosting servers, all three of us were entering a new area of development. This includes Chrome extensions, Firebase integration, and React app development for Android and iOS. Along with lots of online forums and tutorials, our experience with JavaScript helped us with development in all three areas. In the end, we've built the following three components and features:

Chrome Extension

1. Password Generator that allows the user to generate strong passwords with special characters, numbers, etc. during registration.
2. Device Management page that allows users to monitor all connected browsers with an account.
 - a. The browser details are based on the registration device model and Chrome version (i.e. *Macintosh; Intel Mac OS X 10_14_4 using Chrome 73*).
 - b. The user can delete other connected browsers, blocking all access on that browser.
3. Account Management page that allows the user to securely link a phone with a browser.
4. Extension popup that notices new logins and prompts user if they would like to save the credentials.
 - a. Extension popup also notices password updates and prompts user if they would like to save the new password.
 - b. The popup also allows users to request passwords for websites, which then are injected into the correct login fields on the page (or the user can do this manually by right-clicking and accessing Tapas 2.0 in the context menu).

Rendezvous Server

1. User Management: The server maintains a database of all user accounts and their keys.
 - a. User authentication: The server allows only authenticated users to be able to communicate between their devices.
 - b. User authorization: The server only allows authorized devices to communicate between each other.

2. Backup storage and retrieval. The server allows the user to store an encrypted version of their credential database in case the user loses their device.
3. Communication with Google services: The server organizes the communication with Google's Firebase cloud messaging API. This allows the server to send messages between devices.

Phone Application

1. Account creation: Users can create new accounts to have full access to TAPAS 2.0 services.
2. Saving passwords: Users can save new account information into their wallets.
3. Passwords retrieval: Users can retrieve account information from their wallets.
4. Importing passwords: Users can import an external list of account information.
5. Save backup of phone database: Users can save an encrypted version of their phone database.
6. Retrieve backup of phone database: Users can retrieve their backup and share the list of account across multiple phones.
7. Activate phones: Users can log in into multiple phones, and choose which phone is their primary device. The primary device is the only device that will be receiving notification from the browsers.
8. Change TAPAS 2.0 account password: Users can change their TAPAS 2.0 password anytime. Changing the password will also update all user keys saved on the server.

As for our security component, the weakest point in Tapas 2.0 is the user's master password. An attacker can attempt to guess a user's account password and once access is gained into the account, all the user's credentials are compromised. In future releases, we plan to bring two-factor authentication when registering a new phone on an account to deter such password-guessing attacks. Some threats we have accounted for include device theft, network packet sniffing, man in the middle attacks, server compromise, compromise of third part services, SQL injections, database compromise. To test, we've simulated scenarios where one or both of the devices, phone and laptop, are stolen and what implications that has on the user's

data. Other forms of attacks that we've considered but did not have time to implement include replay attacks by server and Firebase, malicious input validation, weakness in the cryptographics libraries used to secure the communication, malware, and phishing attacks. In the end, Tapas 2.0 can ensure that whatever data an attacker is able to get, decrypting will not be an easy task.

BACKGROUND AND RELATED WORK

Most of our motivation for this project is attributed to the original [Tapas Paper](#). This paper talks about a new type of password manager, which follows an approach of storing encrypted passwords and making it theft-resistant without the use of a master password. In this implementation, there are two application components, a Manager and a Wallet. The Manager is on the computer side (browser), which stores a symmetric key to encrypt and decrypt passwords that are placed in the Wallet (stores the password cipher-text on the smartphone). Both these devices would have an end-to-end secure channel using TLS alongside Diffie-Hellman key exchange or also known as a method of digital certificates to verify the authenticity of both devices. Lastly, for both devices to communicate it relies on a network connection and a Rendezvous Server to retrieve and store passwords. We use this paper to build upon the password manager idea suggested.

Additional resources include:

[The Emperor's New Password Manager](#): Security Analysis of Web-based Password Managers

This paper has a detailed section on pros and cons of current web-based password managers. This will help guide us from making the same mistakes that existing applications make (JavaScript and UI based vulnerabilities).

[Myki App](#)

Myki is an existing app in the market that “eliminates the risks associated with cloud storage by safeguarding your passwords and sensitive data offline”. We plan to use this resource to learn about various features we could implement for our own password

manager. Some features include auto fill 2FA tokens, login across devices, and share access of credentials without revealing passwords.

[Keeweb](#)

Keeweb is an open-source desktop/browser password manager. We don't plan on building upon this existing code. This will be used as a motivational resource to see what ideas spawn regarding design and usability.

[Secret Sharing](#)

This tool, 'SSSS', was initially to be used to help one recover their account. 'SSSS' generates X unique keys that will be shared with X friends. If in case the phone is lost/broken or the laptop is stolen, the user's data is recoverable if they are able to get Y of the X keys (where $Y \leq X$). The Y keys put together will decrypt a password that can be used to reconstruct the user's encrypted passwords (idea/implementation under review).

Demo: <http://point-at-infinity.org/ssss/demo.html>

[Password Managers: Comparative Evaluation, Design, Implementation and Empirical Analysis](#)

We wholeheartedly wish we found this thesis paper much sooner. In this, students from Carleton examine current password managers and their faults. More importantly, they evaluate Tapas and implement their own version of it. They are able to offer the 'no master password' feature we initially intended to implement but were not able to due to back up & recovery and multiple device usage. The paper also offers an alternative design where instead of the phone being the password manager, a USB key would hold the credentials — still following the idea of dual-possession authentication.

[UniPass: Design and Evaluation of a Smart Device-Based Password Manager for Visually Impaired Users](#)

Primarily used this paper for their *Usability Evaluation on Password Managers* section.

[Stop Using Password Manager Browser Extension](#)

This article highlights vulnerabilities in password managers operating in the browser extension domain. It's a weak article with only one argument against browser extensions, which is “When you use a browser extension password manager, you give attackers an

API to interact with your password manager via JavaScript or the DOM.” Nonetheless, it provides a view on why desktop-based and built-in browser password managers are better.

[Password Managers Exposing Passwords Everywhere \(auto-download\)](#)

This research paper looks into LastPass, OneLastPass, 1Password, MaskMe and compares them with built-in browser password managers. It highlights various ways attackers deceive extensions. One popular way is to have hidden username and password form fields, which the auto fill feature of these password managers would fill in (See Appx. E for an example and detailed description taken from this research paper). We used this paper primarily for the *Recommendations for Users* and *Recommendations for Password Manager Software Developers* sections in the conclusion.

[Password Managers: Attacks and Defences](#)

Similar to the last paper, this USENIX paper examines attack vectors on password managers on browsers and defence tactics that can help improve security.

[What to Consider When Choosing A Password Manager](#)

[5 Things You Should Know About Password Managers](#)

The two articles above helped us outline our goals and understand how we should go about developing our tool by giving us a user's perspective. They're pretty straight forward as they mention the popularity of password managers, what platforms are most secure, and a high-level overview of encryption tactics.

[How Yubikey Works](#)

Yubikey is a similar tool that offers dual-possession authentication in the form of a USB key. Because it is one of our competitors, we thought it would be a good idea to keep an eye on what they are developing.

Note: as you are aware, the number of links above do not meet the minimum count of 15 requested. We would like to note that, instead of populating the list with nonsense that we did not utilize, everything on the list played a crucial part in our decision-making while developing this tool.

METHODOLOGY AND DESIGN GOALS

As mentioned earlier, our goal from the beginning was to build a tool that is user-friendly and provides a sense of security and adheres to the best security practices. To tackle the design problem of building a dual-possession authentication system, we broke our project into three main components: web browser extension, rendezvous server, and phone application. In the following sections, let's explore how these were broken down even further.

Web Browser Extension:

For the browser, its main components were to, in simple terms:

Part 1: Logging into account and pairing user's phone.

Part 2: Get credentials from the user, encrypt them, and send them to the phone.

Part 3: Request credentials from the phone, decrypt them, and give them to the user.

Part 1: Logging into account and pairing user's phone.

Every user will have a Tapas account that is used for pairing their phone with multiple devices (Laptops / Desktops). The design goal for logging into the account and phone pairing is to ensure that any information passed through the network (server, Firebase, etc.) should be done securely. In order to keep the account credential secure and limit the server from reading any plain-text information, it is hashed on the client side before being sent over the HTTPS connection to the server. A design problem we faced early on was to build a way for the server and Firebase to know how to connect a browser and phone together. Specifically, an implementation that could not be accidentally and intentionally replicated for Man-in-the-Middle attacks. For this, we decided upon a single QR Code that would hold a communication key and a browser guid. The communication key would encrypt all communication happening over the network and the browser guid would help identify which browser to send the message to, since Tapas 2.0 supports multiple browsers under one account. At this point, the registration and setup is complete — and the user is set to use Tapas 2.0.

For Part 2: Get credentials from the user, encrypt them, and send them to the phone.

Tapas 2.0 seamlessly integrates with the user's Chrome browser. As the user registers for a new website, they do not have to worry about memorizing their passwords or letting a third party tool store it in the cloud. Tapas 2.0 will automatically notice that a registration page has been submitted and prompt the user asking if they would like to save the new credentials to their phone. This part of the implementation takes place in *injector.js* and uses the operating systems' notification system to prompt the user. The idea behind this is that they are already familiar with their browser notification system, then let's use it instead of making the user learn something new for no reason. Tapas 2.0 also brings a strong password generator to the table, leaving the user stress free from memorizing difficult 14 character passwords with special characters (i.e. @x\$#-SCb8-rDUh). This can be generated on the browser extension, copied and pasted into the password field, or they can right-click on the password field to view the context menu, select *Tapas 2.0*, and *Create Strong Password* that will auto-fill into the field.

A major component of the browser extension is that it is the only component that should know how to encrypt and decrypt the credentials. In simpler terms, neither the server nor user's phone should be able to decrypt the passwords. The passwords key is never shared, or stored on any other devices except for the browser. Moreover, once passwords are collected, they are encrypted with the passwords key and sent to the server, which will then notify the phone to store a new credential entry in their phone.

For Part 3: Request credentials from the phone, decrypt them, and give them to the user.

Currently, when users request for their passwords from built-in password managers such as Apple Keychain or Chrome Passwords, these credentials are stored on a server and on each of the user's devices. They may be encrypted; however, if an attacker were to gain access to any device, they could essentially log into the users with one click thanks to auto fill. With dual-possession authentication, the process of retrieving passwords takes a few more steps and in return provides significantly stronger password management. When attempting to retrieve account credentials on our tool, the user has to click on the Tapas 2.0 icon in the browser tab and

select *Get Site Credentials*. At this point, their phone would receive a notification with the following options:

- I. The credentials for the site requested does not exist.
- II. A list of credentials for the site requested. The user would then select the username they would like credentials for. The information would be sent to the browser where it auto fills into the login fields. Note: Due to a wide range of login forms, Tapas 2.0 tries its best to scrape the website and find the correct fields. At times, it is not able to do it correctly. In this case, the user is able to right click on the login fields, select *Tapas 2.0* in the context menu, and may *Paste Username* or *Paste Password*. Our inclusion of these features is to make the tool as convenient as possible for the user.

Rendezvous Server:

The main responsibility of the server is the communication between devices. The server holds a database of all user accounts, and other information about the devices linked to each account, including a list of device IDs. This list of IDs helps the server route messages between paired phones and browsers. Another main feature of the server is holding a collection of backups for users. Each user can choose to back up a secure copy of their TAPAS 2.0 database, and later retrieve it on any device running our phone application.

Phone Application:

The phone application acts a portable safe. Users can save an encrypted copy of their account information, and only the paired browsers can decrypt them. The phone application is always listening to commands from the browser, which follows the CRUD protocol:

1. Create and update entries for encrypted information. Once the phone receives a *create* command from the browser, it will do a quick lookup in the phone's database. If the user account details were saved on the phone prior to this message, then it will update the existing record, otherwise, create a record.

2. Read user information and deliver them back to the browser. Once the phone receives a retrieve command from the browser, it will do a quick lookup in the phone's database. If the

user account details were saved on the phone prior to this message, the phone will send a notification to the browser containing the account information found. Otherwise, the phone will display an appropriate message to the user.

3. Passwords deletion. The phone application allows the user to delete a specific username or site information from the application. Users also have the option to delete the entire phone database if they really want to.

TOOLS AND LIBRARIES USED

Plenty of existing tools and libraries were used to make this project come together. To break it down into our three main components:

Browser Extension:

[Chrome Extensions](#)

We chose to develop our browser extension for Chrome because of its popularity and support from Google and the community. With lots of tutorials, developer capabilities, and integration with external tools, it was the most logical choice as this was also our first time developing extensions. We had intentions of extending our application to Mozilla Firefox, however, due to a small team of only three people, priorities were shifted.

[CryptoJS](#)

CryptoJS is an open-source JavaScript implementation of standard and secure cryptographic algorithms built by Jeff Mott. Tapas 2.0 utilizes CryptoJS AES and PBKDF2 library. CryptoJS was used in this project because of its large community support and high downloads on the npmjs site. Note on why we should have used another module is in the Lessons Learned section.

AES Encryption
<p>We chose to use the AES algorithm to handle all of our encryption/decryption operations because it's the world's leading encryption algorithm alongside RSA. AES is the standard encryption algorithm in the US government and it is widely implemented in many internet protocols and communication systems. Due to its high rank in the industry, we were bound to use this algorithm.</p>
PBKDF2
<p>PBKDF2 is not the best hashing algorithm to be used in today's world. However, we were bound to utilize this algorithm since Argon2, Bcrypt, and Scrypt were not compatible with React-Native.</p>
Futoin HKDF
<p>This HKDF node module was the best and most frequently updated source code. This had the best performance comparison against other HKDF implementations. Also, this module is fully compliant with HKDF standards.</p>
qrCodeJS
<p>The purpose for using the qrCodeJS library is to help in sharing the communication symmetric key and the browser guid to the phone without having to share it over the network. This is a well-praised QR Code generator for web use and served our purpose effectively. During our research phase, we did look into sharing the key and guid through the use of barcodes, over Bluetooth but at the end, asking the user to scan a QR code worked best for all components of the tool.</p>

[Papa Parse](#)

The Papa Parse library is used to help parse the imported Chrome passwords CSV file. It converts the file into an array of JSON objects, which then are encrypted and passed onto the phone. A few reasons we jumped right away to using this library was:

- 1) it's the first result when searching for 'CSV parser JavaScript' on Google.
- 2) it has a great demo page along with clean and clear documentation.

And finally 3) it served our purpose in a clean manner without complicating the code base at all. With all these reasons combined, especially the demo page, we looked no further. A note on why we would have used another CSV parser in Lessons Learned section.

Rendezvous Server:

[Node.js](#)

Node.js is an open-source, cross-platform JavaScript run-time environment that executes JavaScript code outside of a browser. Our server runs on node.js, which ensures compatibility with multiple platforms which gives customers and developers the ability to easily deploy and use our services on all available platforms.

[Express](#)

Express is an open-source web application framework for Node.js. Our API endpoints run on top of Express allowing us to build RESTFUL API's easily. All devices use RESTFUL API's as their main communication point.

[https](#)

HTTPS is a separate module for the HTTP protocol over TLS/SSL in Node.js. This helps secure the communication between the devices and the main server.

[fs](#)

This module provides an API for interacting with the file system. We use the fs module to access the certificate files that are required to enable the HTTPS protocol.

[body-parser](#)

We use this module to parse the body of the RESTFUL requests sent from the devices to the server. All RESTFUL API endpoints expect the body of the requests to be structured as a JSON object; however, the fetch API only supports sending the message as a string. Therefore, the devices have to struct the body of the RESTFUL requests as a JSON object, and this module helps in transferring it back to JSON object on the server side.

[uuid](#)

We use this module to generate universally unique IDs. These IDs are used to identify devices when a user successfully logs into the device. The IDs are later used to identify source and target devices of messages between browsers and phone.

[better-sqlite3](#)

We organize our data in tables using better-sqlite3 module. It provides efficient and synchronous database operations that makes our server fast and consistent.

[fcm-node](#)

This module allows the server to communicate the Google's Firebase services.

[Bcrypt](#)

We use this module to hash and verify user's authentication key.

Phone Application:

[React-JS](#)

A JavaScript library for building user interfaces. We used react to create the different UI components used in our phone application.

[React Native](#)

React Native lets you build mobile apps using only JavaScript. We used React Native to port over the React JS design library into the phone's universe. React Native also gave us access to some phone features such as camera, notification system, etc. React Native helps us to build cross-platform phone application, which increases the usability of our application.

[React Native UI Kitten](#)

This is component library that contains a number of UI elements that are used in the creation of the phone application.

Other libraries that assisted with UI include:

[React Native Paper](#)

[React Native Vector Icons](#)

[React Native Elements](#)

[React Native Bottom Navigation](#)

[React Native Collapsible](#)

[React Native Modal](#)

[React Native QR Code Scanner](#)

This is component library that helped in reading the QR Code generated by the browser.

[React Native Firebase](#)

We use this library to allow the phone devices to communicate to Google's Firebase services. This module is mainly used to receive Firebase Cloud Messaging notifications.

[Crypto JS](#)

We use this is library to generate the different PBKDF2 keys, and decrypt and encrypt the messages between devices.

[Futoin HKDF](#)

We use this is library to generate HKDF keys.

PROJECT OVERVIEW

Section 1: Design in Detail

Tapas 2.0 has three main components: a browser extension (**Manager**, encrypts/decrypts passwords), a rendezvous server, and a phone application (**Wallet**, stores encrypted passwords). The browser extension can encrypt and decrypt passwords and makes the request to the phone for passwords when the user wants to log into a website. It also scrapes the web page to keep track if a password has been updated, a new account is being created, and auto fills the forms when passwords are retrieved from the phone. In the Tapas 2.0 options page, the user is able to register a new phone to the browser, manage current browsers connected on the account, and import passwords from external password managers. The server manages communication between the devices as well as helps users recover backups. The phone stores all the encrypted passwords and alerts the user when the browser is requesting for credentials. Users are also able to manage the database of credentials and backup & recover their passwords from the server.

In our initial implementation of Tapas 2.0, we wanted to avoid any use of a master password. However, moving forward with no master password caused difficulties in coming up with a user-friendly solution for password recovery and utilization of multiple devices. By building upon the original Tapas idea, we incorporated the use of a master password, which has allowed us to add the extra features without sacrificing ease of usability.

Here is a detailed outline of all major components of our tool.

Section 1.1: Creating an account and key generation:

In our implementation of Tapas, a user will either download the Tapas 2.0 app on their Android or iOS device where they will register an account. Tapas 2.0 only requires two pieces of information from the user — a username and password. Once registration is complete, the key generation process begins. This process follows the same idea as Firefox Sync. Tapas 2.0 utilizes a password-based key derivation function (PBKDF) to generate a unique key for the user. The node module 'pbkdf2' is used as follows:

```
pbkdf2(pass='password', salt='username', rounds=1000, length=32, hash='sha256');
```

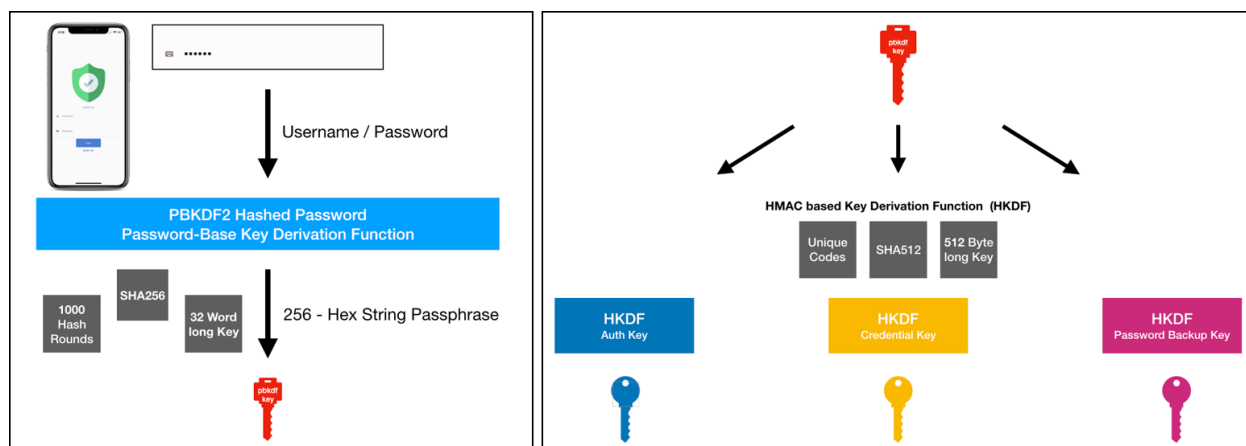


Figure 1: Visualization of how keys are generated on the phone for initial set up. Once the account is set up and verified by the server, all keys on the phone are erased since the phone does not need the keys for everyday application use.

The user's username and password are passed as the salt & password to the PBKDF function. The hashing algorithm is set to sha256 with a 1000 rounds of iteration and a final key length to be 32 Word long (256 bytes). This gives us a unique key for the user, we can call this **UserKey**.

Now that the **UserKey** has been generated, we can create different keys based on this to perform unique operations. The unique operations that our application has includes authenticating a user, storing encrypted passwords on the wallet, and performing backup & recovery of the wallet. We utilize a hash-based message authentication code (HMAC) based key derivation function (HKDF) to generate these 3 new keys. The goal of using HKDF is to use one strong hash key to derive one or more cryptographically strong secret keys ([HKDF 2010](#)). The node module 'futoin-hkdf' is used as follows:

```
hkdf(IKM=UserKey, length=512, {info='UniqueCode', hash='sha512'});
```

The argument IKM is short for initial keying material: the value that we wish to generate a new key based on. We configure the HMAC function to be sha512 with an output length of 512 bytes. Lastly, the info argument is used for application specific information. This allows Tapas 2.0 to distinguish the different keys that are generated from a single IKM. Moreover, "it prevents the derivation of the same keying material for different contexts." (HKDF 2010). We use this

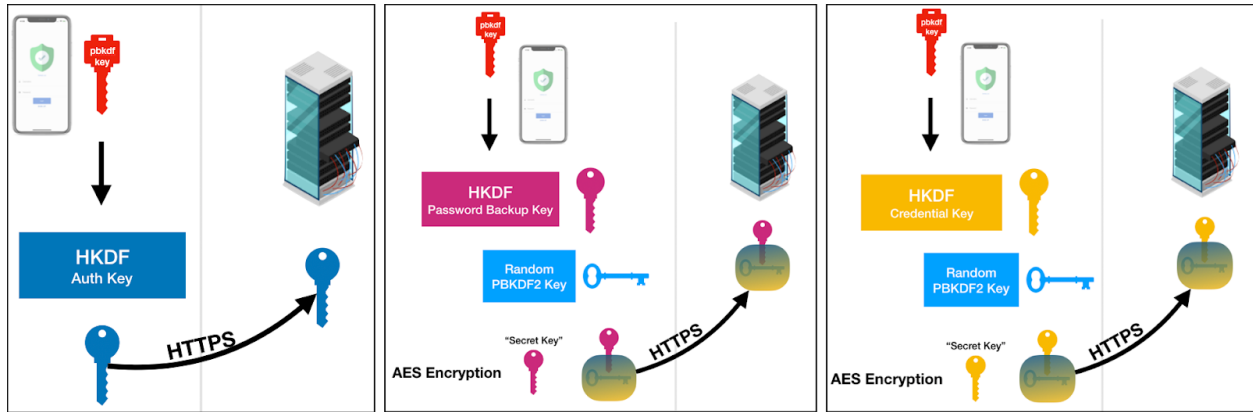


Figure 1.1: Shows how the three HKDF Keys are generated and used on account setup. Also, shows what key and encryption data is sent & stored on the server.

function three times with different 'info' texts to create an **Auth Key**, **Credential Key**, and a **Password Backup Key** (Wallet Backup).

Now let's talk about how each of the newly generated keys are used in Tapas 2.0.

Auth Key: When a user logs into their accounts in most applications, the passwords are sent in plain text via HTTPS to the server, which then performs a hash on the passwords to store it safely in the database. If the server is compromised, then an adversary can learn about the user's passwords and use it against them. To avoid this, we compute the Auth key either on the phone or browser extension before sending it over to the server. The server does not gain any knowledge about the user's password and thus, an adversary can only use a brute force attack to derive the password. Note: the username is used as a salt and never sent in plain text. On account creation, when the phone first sends the Auth key via HTTPS to the server (running Node.js), the server further hashes the Auth Key via the node module 'Bcrypt' with a community-recommended 11 rounds of hashing. Then the server generates a timestamp-based account ID via the 'uuid/v1' node module and saves it in the sql database with the new hash of the Auth Key. Once the server verifies this, it sends a message back to the phone letting it know the account creation was successful along with its account ID.

Credential Key: The credential key is used as a lock to safely store a **password key** on the server. The password key is used for encrypting and decrypting the passwords that are stored on the wallet. Note that only the browser extension does this operation (and not the phone). On the phone, we randomly generate another PBKDF2 key that is not based on the user's password,

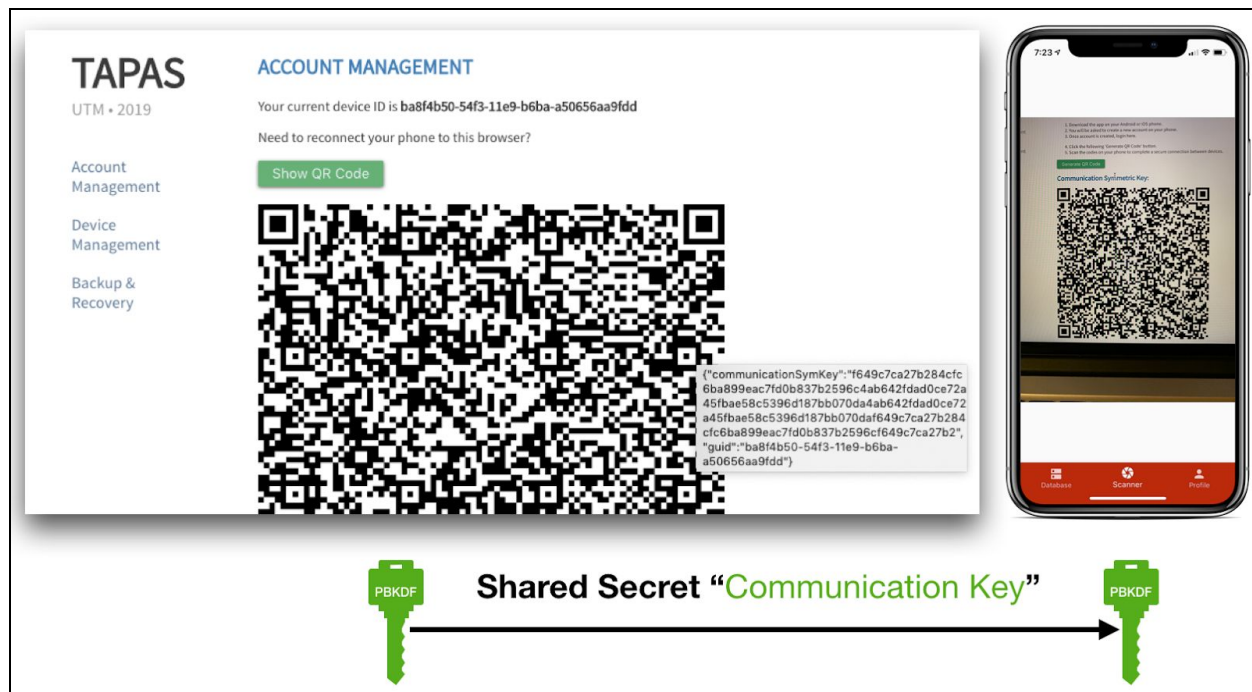
which will be used as a password key. The password key is then encrypted with the credential key using the node module 'crypto-js/AES'. This is sent along with the Auth key on account creation to be stored in the server's database. A diagram of this can be seen in Figure 1.1.

Password-Backup Key: Using the same idea from the Credential Key, the same is done for the Password-Backup key. This key is used as a lock to safely store a **recovery key** on the server. This recovery key is used to encrypt the encrypted wallet passwords to be stored on the server. On the phone, we randomly generate another PBKDF2 key, and we call this the recovery key. The recovery key is encrypted with the Password-Backup key using the same 'crypto-js/AES' node module. This is also sent along with the Auth Key on account creation to be stored in the server's database. A diagram of this encryption process can be seen Figure 1.1.

Once, the server verifies that the account was successfully created, all the keys generated on the phone are then erased. In case the phone is stolen, the confidential keys cannot be retrieved to decrypt the passwords. Once user is authenticated, they can choose to active a single device from a set of device. Once a phone has been activated, all browser messages will be directed to the active phone. This is controlled by the user and managed by the server. The user controls which phone is the active phone, and the server will manage the communication between the different browsers and the active phone. Please see the Appendix to view this feature (Appx. A).

Section 1.2: Pairing devices

First, the Tapas 2.0 extension must be downloaded on a Chrome browser. On Chrome launch, the extension connects to Firebase Cloud Messaging (FCM) service. The server uses Firebase to communicate to the Chrome extension since the extension cannot have a web socket server running independently. When the user logs onto the browser extension with their account, the browser recomputes the same keys as done on the phone because they are based off of the user's username and password. Steps are as follows:



*Figure 2: Shows how a computer and phone are paired.
Communication key is shared between the computer and phone.*

1. We use the same PBKDF2 and HKDF node modules used on the phone to generate the keys.
2. Browser sends over the AuthKey via HTTPS POST message for a login request along with its Firebase token ID (FCM ID), browser and OS details, and time of login.
3. Server verifies the AuthKey stored in the database.
 - a. Server creates a Browser ID (using 'uuid' node module) for the extension and stores it in the SQL database along with the FCM ID under the user's account.

4. Server replies with a Browser ID, which is unique for every computer the user logs into.

Now that the browser has a unique ID created by the server, a secure channel for communication needs to be created between the user's computer and phone. This is accomplished by having a shared secret between the two. The browser generates a random PBKDF key called **Communication Key**. The purpose of this key is to securely establish a trust between the phone and browser, so that any messages sent by the browser can only be read by the phone and vice versa. This communication key is stored locally on Chrome for persistence. Note: other extensions or scripts cannot access Tapas 2.0 Chrome database because of Chrome's extension

structure. Next, qrCodeJS library generates a QR code that holds the communication key and the Browser ID. We assume that an adversary does not have access to this screen. As you can see in Figure 2, the user scans the QR code on the phone and establishes the shared secret between the devices. The user can do this on multiple computers and each of the computers will have their own shared communication key with the phone.

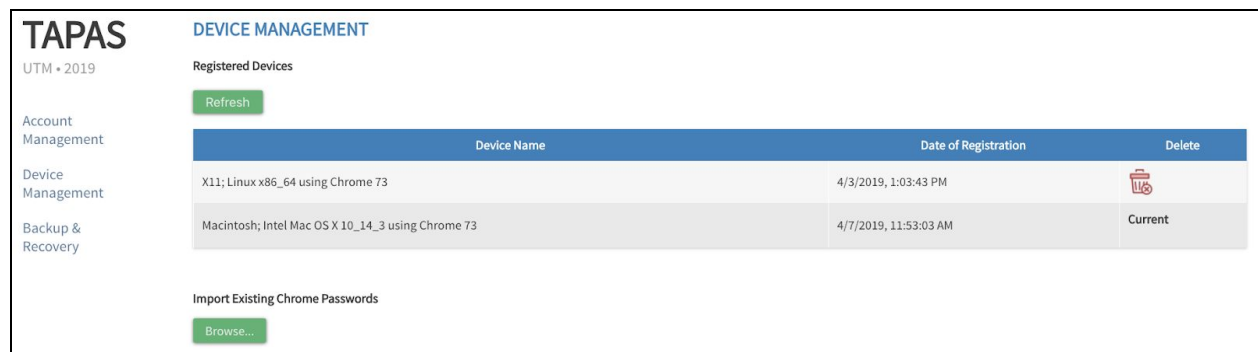
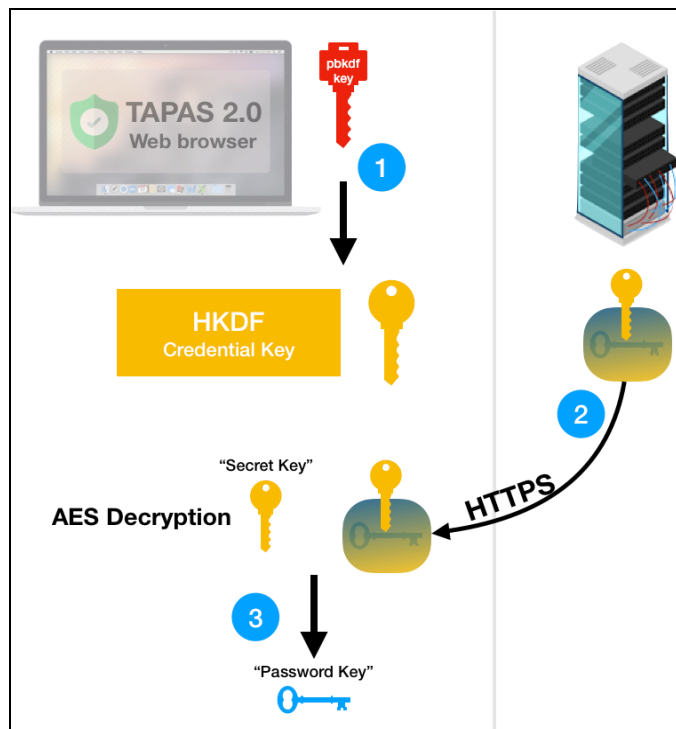


Figure 2.1: Device Management; User can view all computers connected to their account and have ability to deactivate them.

Along with pairing multiple computers with their phone, the user also has the ability to manage which computers should have access to their phone wallet. This gives the user more control over their account. The refresh button on the Device Management page (Figure 2.1) initiates a GET request to the server for a list of connected devices on this current account. The server fetches that information and responds back with all the details stored for all devices related to the account. The table on the page displays of all connected devices along with a functionality to revoke access for a particular browser. If the user wishes to delete access for one of the accounts, then a DELETE request is initiated to the server and the server removes the communication access of that Browser ID from the database.

Section 1.3: Saving credentials

At this point, the user has paired their devices and is ready to use Tapas 2.0 password manager. To store passwords onto their wallet, the extension requires the “password key” that was created on account creation. When the user logs in on the browser, the server sends back the encrypted password key via HTTPS response request. The browser then generates the 'Credential Key' to



decrypt the encrypted password key via the same 'crypto-js/AES' node module.

Finally, the browser extension has a hold of the “password key” and saves it to Chrome’s local database.

Figure 3: Shows how the browser extension gets hold of the password key from the server.

To meet our ease-of-use design, the goal for storing passwords needed to be a one click operation. When a user visits a website, the browser extension injects a

script (*injector.js*) once all DOMs have been loaded. This script parses for any form fields that have a password input in them. Once detected, it attaches a DOM listener to the submit button on that form. When the listener is triggered, Tapas 2.0 will be notified of the form submission and the values it contains. There can be two scenarios when the user is submitting password forms:

1. Existing account login
2. New account registration

If there are only two input fields in the form submission, it is assumed that the user is logging into an existing account and we can actively prompt them with an OS notification asking if they would like to save the password on their wallet (Figure 3.1). If there are more than two input fields detected, then there is a two step process to store these passwords.

After the user agrees to store the passwords in the wallet, they will be prompted to re-enter their username to finish the user end of the process. Behind the scenes, Tapas 2.0 stores the first five characters of a SHA512 hash of encrypted password, username, website information on the browser extension. This allows the tool to check if the user already has a password stored on the wallet or not and prevents multiple prompts asking the user for the same credentials. This also allows Tapas to detect if the user updated their password and ask them to consequently

update their password on the wallet as well (Figure 3.1). The user also has the ability to manually enter a password that they wish to store on their wallet, by accessing the main extension pop page. Please see the Appendix to view this feature (Appx. B).

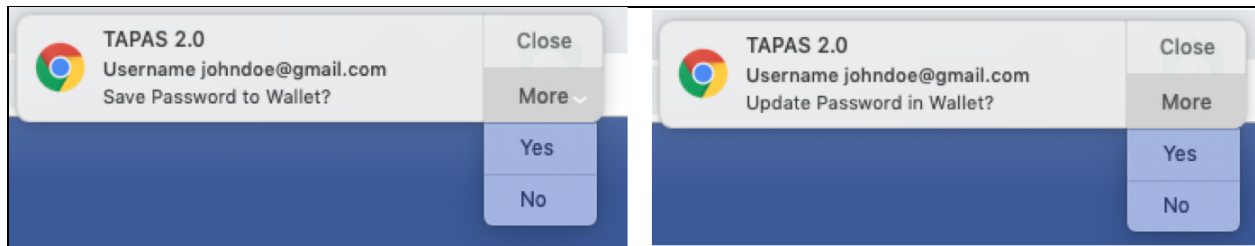


Figure 3.1: Mac OS operating system notification prompts the user to save or update their password.

At this point, the extension is ready to send the credentials to the phone. The extension creates an object that includes the website, username, and the encrypted password. Once this is created, we encrypt this message with the shared communication key. Next, the extension sends this message in a HTTPS POST request to the server and lets the server handle which phone to forward this too. The server formulates a Firebase message to the phone. The server responds back to the browser that the message was successfully sent. Firebase will send a push notification to the phone with the encrypted message. Note: only Android devices will receive notifications. For iOS notifications, a paid developer account is required. The phone then decrypts the message with the appropriate communication key that was shared between the browser and saves the result into its local database. Figure 3.2, illustrates a diagram on how this occurs.

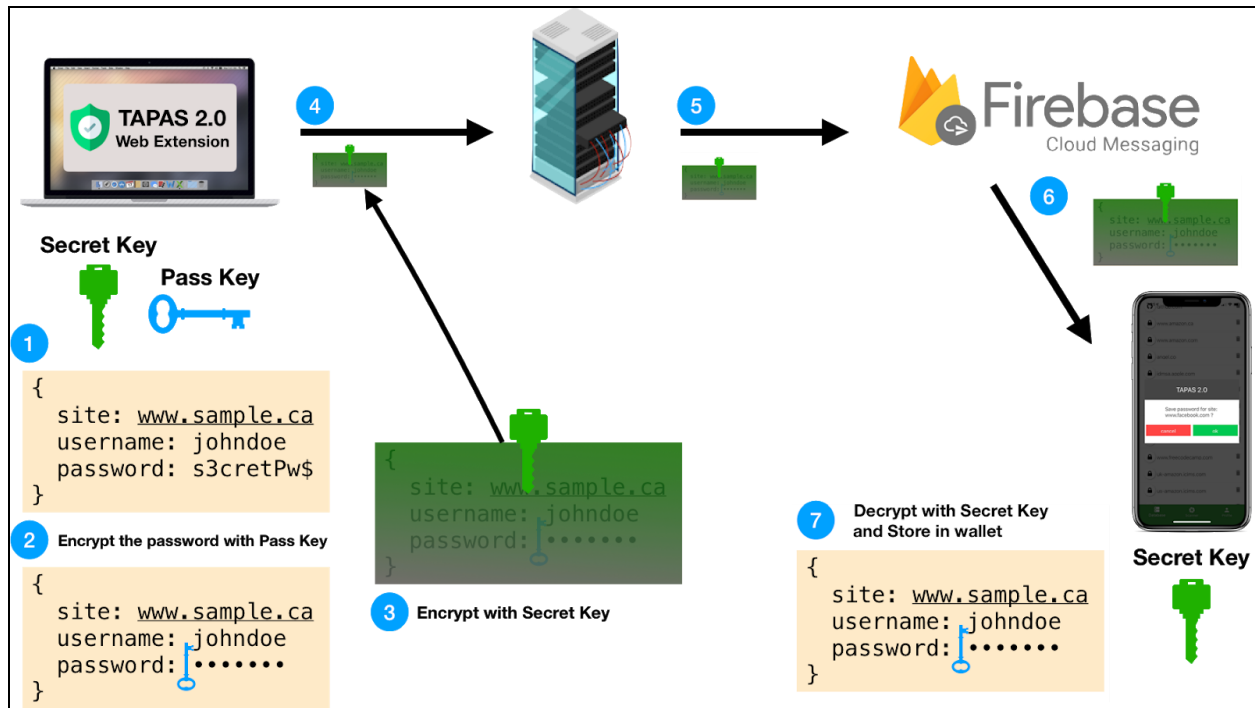
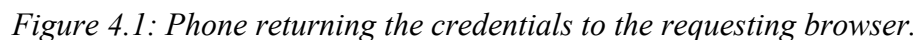
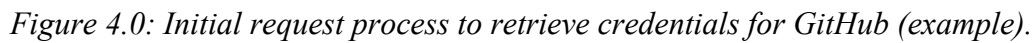


Figure 3.2: Process to store the passwords on the wallet.

Section 1.4: Retrieving Credentials

Retrieving passwords from the wallet is simply the reverse process of storing the credential onto the wallet. First, the user visits a login page. The user then selects on the 'Get site credential' button on the extension pop-up, which constructs a message with the website URL to retrieve the username and password from the phone. The browser encrypts this message with its communication key and initiates an HTTPS POST request to the server. The server then finds the paired phone from its database and sends a Firebase notification to the phone. Once the phone receives the request, it decrypts the message body with the shared communication key. Then the phone looks up the website in its own local database and fetches all usernames and cipher-text passwords under the same website. The user selects the credentials they want to use (if they have multiple accounts for same website). Next, the phone constructs an encrypted message using the same shared communication key the browser used and sends an HTTPS POST request message to the server. Then the server finds the paired browser from the database, and sends a Firebase

The browser extension now has the username and password for the current site. Tapas 2.0 attempts to automatically fill the users credential in the correct login form by using the injector



script (injector.js). In some websites, such as Instagram and Twitter, client console cannot edit the form values. If the auto fill fails, the user still has the option to paste their credentials via the context menu. A user would have to right click on the form fields and select paste username & password separately. Please see the Appendix to view this feature (Appx. D).

Section 1.5: Importing passwords to Tapas 2.0

Importing passwords into Tapas 2.0 allows users to transition from their existing password manager. As you can see in Figure 2.1, users have the option to import a CSV file of their credentials. Once the file is selected, Papa Parse will provide Tapas with a list of all the credentials in a JSON object. Next, the browser will send an `action:"init_bulk_update"` to notify the phone of an incoming stream of credentials. Once the phone responds with an acknowledgement, the credentials will be bundled in groups of 10 and sent over. The free tier of Firebase has a limit on the size of data being transferred in one call, thus the limit of 10. All imported passwords are encrypted in the same manner as reported in Part 3. Once the phone receives these bundles, it combines the bundles into a list of accounts. Then it pulls the existing list of accounts that are saved on the phone and merges the incoming list from the browser with the existing one on the phone. If there are any overlapping entries, the phone will update the old entries with the information given in the browser bundles.

Section 1.6: Backup and Recovery of Wallet

When a user wants to perform a backup of their credentials, the Profile page on the app has an option to 'save backup'. This prompts the user to enter their password in order to initiate the process. Once the user enters the correct password, we generate the Password Backup Key (because keys are never stored on the phone). Then the encrypted recovery key is requested from the server via HTTPS POST request, which is then decrypted using the Password Backup Key to get the Recovery Key. Next, the app encrypts the entire password wallet list using the Recovery Key and initiates an HTTPS POST request to save encrypted password file onto the server (Note Recovery Key is erased from the phone after its use). The server saves the encrypted password file in the SQL database under the user's account. Now, when the user ever

wants to recover their passwords, they simply request that encrypted file from the server and decrypt the result on the phone using the same Recovery Key (Figure 6, left box).

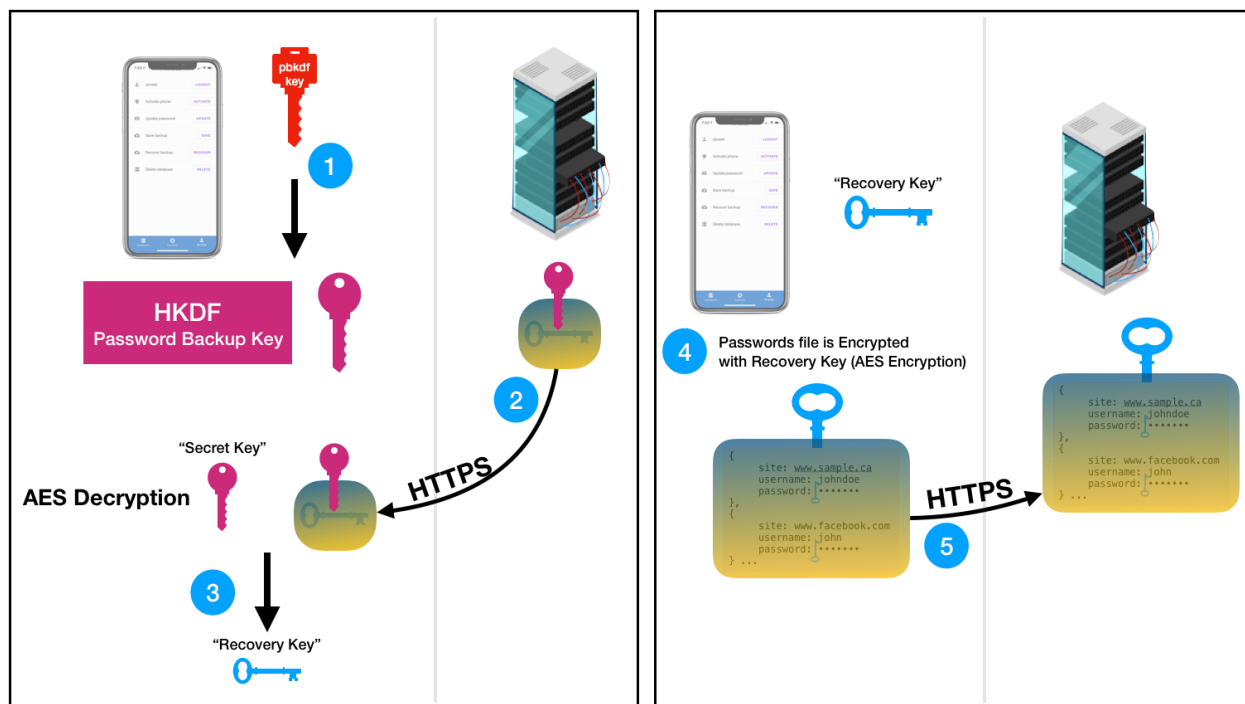


Figure 6: Process flow through to retrieve recovery key and backup passwords onto server.

Section 1.7: Additional Notes

The Chrome web extension was written primarily in HTML, CSS, and JavaScript. We built upon a sample extension from developer.chrome.com, adding the UI, various options pages and external resources such as Firebase and encryption dependencies where needed. Another major portion of the development time can be directed to integrating the extension with the QR code generator for pairing browser with phone, Firebase messaging for sending messages and receiving notifications, CSV file parser for importing passwords, password encryption and decryption with Crypto's AES, and Notie for user-friendly notifications. The server is written entirely in JavaScript allowing users to run the server on all major platforms that support running nodejs. The server is responsible for accounts and backups management.

As for the React-App, it is written in JavaScript and JSX with the help of React and React Native libraries for cross platform support. It also uses React Native API to access, modify or delete storage elements that are used by the application to maintain a local copy of the device

custom ID, Firebase token and the passwords' database. Last but not least, the app also uses the Google's Firebase API to get notification from the server. There are 3 types of states the phone application can be in while receiving a notification from Firebase services. Running the foreground, and that receives a direct notification of the application. Running in the background, whether the application is turned off or on, there will be a notification in the notification tray. Once a user clicks on the notification, it will launch the phone application, and display the notification modal. Once a user logs out of the phone application, the Firebase token will be revoked and the device will not receive any notification from the user until the user logs back into the application with the proper account.

Section 2: Developer Guidelines

The following sections define all the parts a future developer will need to set up before they can begin testing and extending the capabilities of our tool.

Section 2.1: Chrome Extension

Firebase configuration:

Follow this [Firebase tutorial](#) to learn how you can generate your own messaging key pair. Once you've got that information, go to `src > background > config.js` and replace the current `apiKey`, `authDomain`, `databaseURL`, etc. variables with the ones you generated in the tutorial. Lastly, go into `src > background > background.js` and modify the line containing `messaging.usePublicVapidKey` with your own key created during the tutorial.

Installing on Chrome:

On the latest version of [Google Chrome](#), visit `chrome://extensions/` to view your extensions. At the top-right, turn on Developer Mode. In the top bar, select Load Unpacked and navigate to the `src` folder in the `tapas-2.0` project folder. Once selected, the

extension will show up alongside your other extensions. If you make changes in the `src` folder, press the reload icon on the extensions page to view the changes.

Section 2.2: Server

Server setup:

Environment set-up for android:

Binaries:

Node: v11.12.0

npm: 6.7.0

1. Install dependencies

- a. Run `'npm install'`

2. Run the server

- a. Run `'nodejs index.js'`

Section 2.3: Phone Application

React setup for Android:

Environment set-up for android:

Binaries:

Node: v11.12.0

npm: 6.7.0

npm Packages:

react: 16.6.3

react-native: 0.58.0-rc.2

npm Global Package:

react-native-cli: 2.0.1

Note: root directory is `phone_app`

1. Install dependencies

- a. Run the setup script, *root directory/setup.bash*.

2. Add Firebase configuration file

- a. In your firebase account (console.firebase.google.com), navigate to your android app and download the 'google-services.json' file associated with your app. Replace this file with the current one under 'android/app/' directory.

3. Add android sdk path

- a. Create a new file under 'android' directory. The file should contain the following line: `sdk.dir = <PATH TO YOUR ANDROID SDK>`

4. Link your react native packages

- a. `react-native link react-native-firebase`
- b. `react-native link react-native`
- c. `react-native link react-native-vector-icons`
- d. `react-native link react-native-ui-kitten`
- e. `react-native link react-native-paper`
- f. `react-native link react-native-camera`
- g. `react-native link react-native-qrcode-scanner`
- h. `react-native link react-native-permissions`
- i. `react-native link react-native-material-bottom-navigation`
- j. `react-native link react-native-randombytes`
- k. `react-native link futoin-hkdf`

5. Building the application

- a. Run 'yarn start' in one terminal
- b. Run 'yarn run android' in a second terminal. Make sure you connect your android device first, and enable developer mode on your phone.

React setup for iOS:

Note: root directory is phone_app

Environment set-up for iOS:

Binaries:

Node: v11.12.0

npm: 6.7.0

npm Packages:

react: 16.6.3

react-native: 0.58.0-rc.2

npm Global Package:

react-native-cli: 2.0.1

iOS SDK: iOS 12.2 or higher

Xcode: v10.2 or higher

1. Install dependencies

- a. Inside the root directory run ``yarn`` or ``npm install``
 - i. Sometimes the iOS related react-native files will not be downloaded. To manually fix this issue you must run the following in the root directory (This is an important step!):
 - i. Run ``cd node_modules/react-native/scripts && ./ios-install-third-party.sh && cd ../../../../``
 - ii. Now navigate to ``node_modules/react-native/`` and check if there is a folder called ``third-party``, this is the folder that should have been created in this directory. If not navigate to ``node_modules/react-native/scripts`` and copy the newly created folder ``third-party`` to the parent directory.
 - iii. In the root directory run ``cd node_modules/react-native/third-party/glog-0.3.5/ && ../../scripts/ios-configure-glog.sh && cd ../../../../``
 - iv. All node modules should be successfully installed.
- b. Inside the root directory ``cd ios`` and run ``pod install``
 - i. This will install the related firebase libraries
 - ii. If you do not have CocoaPods, follow these steps located here: <https://guides.cocoapods.org/using/getting-started.html#getting-started>
 - iii. After complete installation a new file will be created in the iOS folder called `[projectName].xcworkspace`, in our case it is called `tapastwophone.xcworkspace`. This is the project file that Xcode will use to build the app.

- c. After downloading these dependencies you will need to link the react-native modules that we utilize for the app to iOS:
 - i. Run the following:
 - 1. `react-native link react-native-firebase`
 - 2. `react-native link react-native`
 - 3. `react-native link react-native-vector-icons`
 - 4. `react-native link react-native-ui-kitten`
 - 5. `react-native link react-native-paper`
 - 6. `react-native link react-native-camera`
 - 7. `react-native link react-native-qrcode-scanner`
 - 8. `react-native link react-native-permissions`
 - 9. `react-native link`
`react-native-material-bottom-navigation`
 - 10. `react-native link react-native-randombytes`
 - 11. `react-native link futoin-hkdf`
- d. Depending on user permissions on your Mac, downloading the dependencies on terminal will create the new folders and files with '**system**' permissions rather than your **account/username** permissions. Xcode will only be able to access those folders if it has your **account/username** permissions associated with it. Do the following:
 - i. In the root directory navigate to **node_modules** folder, right click on the folder name and select 'get info'. Navigate to 'Sharing & Permission'. Check whether or not it has your account/username set to 'read and write' permissions. If not, click on the **+** and add your username with 'read and write' permissions. After your username is added to the permission set, select your username and click on the settings icon at the bottom and select 'Apply to enclosed items', this will apply this permission to all the contents in this folder.
 - ii. Repeat step one for the **ios** folder in the root directory.

2. Add Firebase configuration file

- a. In your firebase account (console.firebase.google.com), navigate to your iOS app and download the 'GoogleService-info.plist' file associated with your app. Replace this file with the current one under 'ios/tapastwophone/' directory
- b. Open your Xcode Workspace file 'tapastwophone.xcworkspace':
 - i. Rename the Bundle ID in Xcode to match the same Bundle ID you created for your Firebase app.
 - ii. The 'GoogleService-info.plist' might not have been added in the Xcode project automatically, you will need to add it manually under the tapastwophone folder in Xcode.

3. Building the application

- a. Xcode may not automatically link all libraries that are required for the iOS app. Here is a solution to manually add libraries into Xcode Workspace (tapastwophone.xcworkspace): [module was not found \(Stack Overflow\)](#)
 - i. These are the mandatory libraries required:
 1. RNfirebase.xcodeproj
 2. React.xcodeproj
 3. RNRandomBytes.xcodeproj
 4. ReactNativePermissions.xcodeproj
 5. RNCamera.xcodeproj
 - ii. These files are located in the root directory under 'node_modules/[libraryName]/ios/'
- b. Before Xcode can build the iOS app, it needs a bundled version of the react-native code, which will be used to build the iOS app. Follow the steps in this link to set this up: [First time deploying with react native \(Medium\)](#)
- c. Finally, connect your iPhone and click build!

Section 3: Source File Breakdown

tapas-2.0

↳ phone_app

setup.bash

- setup script to download dependencies for android app

↳ **android**

- contains the required files to build the android application.

↳ **assets**

- contains external files, such as pictures and icons.

↳ **components**

- contains custom components created to deliver the UI features.

↳ **ios**

Podfile

- contains the firebase libraries and dependencies.

↳ **tapastwophone**

- contains iOS app related code, firebase configurations, and react-native source bundle to build the app.

↳ **tapastwophone.xcworkspace**

- contains Xcode project data for the IDE.

↳ **server**

Index.js

- connects the different node modules to perform user management tasks and communication with firebase API.

↳ **src** (Chrome extension)

↳ **background**

- runs when Chrome starts, sets up Firebase comm., and builds context menu options.
- Handles the encryption/decryption of communication messages between phone and browser, as well as encrypting and decrypting the credentials.

↳ **img_resources**

- all images part of the extension.

↳ **injector**

- parses the webpage for login/registration forms,
- populates the fields with username/password (autofill feature).

↳ **options**

- contains extension settings pages (Account Mgt., Device Mgt., etc.)

↳ **js_resources**

- contains code for papaparse(csv parser), qr code generator, aes

encryption, notice user-friendly notifications, pbkdf2 encryption, and Account/Device Mgt. background pages.

↳ **popup**

- contains the user-facing popup html/js code.

manifest.json

- identifies the extension's information such as name, description, icon, dependencies, requirements, and more.

Section 4: End User Tutorial

Ideally, the end user (whom we'll call Alice) will be downloading our extension from the Chrome Web Store and the app from either the Google Play Store or Apple App Store. In the beginning, Alice will create a new account on the Tapas phone app. Once that is done, Alice will log into the Tapas web extension. Once logged in, the extension will ask Alice to scan a QR code displayed on the browser using her phone. The middle tab on the phone app, labelled 'Scan', is meant for this. After scanning is complete, Alice will go to the Profile tab on the phone and click 'Activate' to complete setup. Now she can begin using Tapas 2.0.

To make Alice's life a little easier, she can export passwords from Chrome and import them into her database through Device Management in the extension options. These extension options can be reached by either right-clicking on the extension icon, then 'Options', or clicking on the icon, then the 'Tapas 2.0' header image. Once the CSV file is selected, Alice will confirm on the phone that yes, indeed, she does want to import passwords.

Alice loves using GitHub to save her hard work. Unfortunately, GitHub credentials were not saved in the CSV file she imported. To manually save passwords, Alice can click on the extension, then the small stacks icon beside the website name (which should say 'github.com' in this example) and will manually enter the username & password. Now when she visits the login page, she can click on the extension icon and select 'Get Site Credentials'. On her phone, she will get a popup asking if she'd like to send over the credentials and for which account (if she had multiple GitHub accounts saved). Once she taps on the arrow beside the username for the credentials that she'd like to transfer over, they will be received by the browser and will auto fill into the username / password fields in the form. She can proceed to click the login button and log

in. Some forms are a little more complicated and Tapas 2.0 will not be able to auto fill. In this case, Alice can right-click on the username / password fields and a context menu will pop up. In there, under Tapas 2.0, Alice can 'Paste Username' and 'Paste Password'. She also can 'Generate Strong Password' in case if she's registering for a new account.

Speaking of registering for new accounts, Tapas 2.0 will notice that she just pressed 'Sign up' and will prompt her if she'd like to save credentials to a website. The extension will also catch if she updates her password for an existing account. Alice can select 'Yes' or 'No' on the notification and if 'Yes' is selected, then she will confirm 'Yes' on her phone once again. Now while Alice is on the phone app, she can head over to the Profile tab and backup her passwords to the server. In case she ever loses her phone, all she'll need is another device with Tapas 2.0 installed, and she can recover her passwords (as long as she remembers her login information). Alice is now officially a trained Tapas user and will find that she can now use strong passwords during account registrations without ever having to worry about what the passwords are.

EVALUATION AND LESSONS LEARNED

Technical and Security Evaluation

For us, testing was a continuous process throughout the development period. Once our project proposal was complete, we began breaking down our design goals into subtasks, which would be consumable by one individual and could be completed within a couple days. As portions of the project were marked completed and pushed to our testing phase in Gitlab, another developer or friend would test various use cases. This may include application behaviour under different Wi-Fi connectivity statuses, with unexpected server responses, or malformed user inputs and unexpected ways of interaction with the UI. Thanks to the breakdown of tasks early on, the expected results of majority of our tasks were simple to identify, so we did not come across any major issues.

Additional tests included monitoring data leak using Wireshark, a packet sniffer. With this tool, we were able to check if any of the data being passed from the computer to the server

and back were leaking any confidential information. With the implementation of HTTPS on top of our multiple rounds of encryption, anyone sniffing our network traffic would not be able to reverse engineer the cipher text messages. We also ran simulations of theft where one or both of the devices would be 'stolen'. At this point, we asked questions such as *what data can I extract?* and *what do I require to get some useful information?*. At both times, the answer was usually 1) I need login credentials to the laptop or the phone, 2) I need login credentials to the Tapas 2.0 app. Tapas 2.0 app and browser stay logged in after the initial setup so assuming they passed the first barrier.

Case 1: Laptop Stolen

If a Tapas 2.0 user's laptop is stolen, the attacker can delete connected browsers on the account, ping the phone and request passwords or request to store new credentials. At no point in time will they be able to retrieve any credentials because the victim has to confirm on the phone whether they would like to send the data over. The Tapas 2.0 account login cannot be changed by the attacker so from the victim's standpoint, they are able to log into any other Chrome browser and revoke access to the attacker's browser. The victim can also change their Tapas 2.0 account password on their phone to ensure the attacker does not do any more damage.

Case 2: Phone Stolen

With the attacker having access to the victim's phone, they still are not able to retrieve passwords. For one, they are encrypted on the phone, and two, they cannot reset the password without knowing the current account password. They can however delete all credentials on the device. If the victim backed up their credentials, they cannot be deleted because that requires the account password.

Case 3: Server Compromised

All credentials and account info are double-encrypted on the server. The major issue with a compromised server would be a denial of service attack. Also, if the attacker wanted to listen to any conversations passing through, anything caught would be encrypted by SSL and then by the communicating devices. That brings us to our strongest weakness. If a Tapas 2.0 user's account credentials are stolen, the attacker can log in on their own browser and phone, recover backup

from server, and retrieve all credentials in plain-text, one by one (until we add an 'export passwords' feature). Moreover, if the user knows their account credentials are stolen, they still have a chance to change their Tapas 2.0 account credentials, which will update all keys before an adversary can take advantage of it.

Once the development of Tapas 2.0 was complete, we created a Google Form to collect feedback from CSC427 Computer Security students, classmates and professors with the following questions and responses (total: 13).

1. Do you currently use any password managers? If so, which one(s)?

46.2% — No

53.8% — Yes

57.2% — Google Chrome Built-in Password Manager

28.5% — Lasts

14.3% — Apple Keychain

2. What is your first impression of Tapas 2.0?

a. Seven responses with satisfaction level of 'Excellent' and six for 'Very Good'.

3. Do you notice any security flaws?

- a. Seven responses with responses along the lines of 'no'. Other comments included:
- You can Tapas □ Paste Password on to a username field and find out the password of your friends (after all the other stuff though so it's not THAT bad)
 - The input field for the username shows the password. Please search and mask data if the user try to put a password into the particular field.
 - People who use the app might not delete their passwords from Chrome.

4. How likely are you to use this tool?

69.2% — Likely or higher.

15.4% — Neutral towards it.

15.4% — Not likely or lower.

5. What is your most and least favourite feature?

- a. Comments about favourite feature include: the UI, password generator & auto fill feature, security and 2FA.

- b. Commands about the least favourite feature include: popup UI's and iOS stability.

With that, we will review our 'Paste Password' context menu feature, keep monitoring Google Chrome's built-in password manager, and continue to listen to feedback for further areas of improvement.

Timeline and Division of Responsibilities

Reviewing our Project Proposal, we were successfully able to exceed all requirements. Some stretch goals that we accomplished include building an iOS phone app (building a stable version would require an apple developer account), wiping data off devices / revoking access of browsers, and minor features such as utilizing the context menu, OS-native notifications, and all with a team of three. Within the first month, one of our team members dropped the course forcing a reevaluation of sprint commitments. Thanks to Ahmed's experience with servers, he picked up all the server maintenance while also working on the Tapas 2.0 React app full-time. Darshan and Raj primarily worked on the browser extension for the first half of the sprint. This included Darshan setting up the Firebase connection and web page injector (to parse form data). Raj worked on all the user interface (popup and options pages) and other browser-side features. Half way through, Darshan jumped ships and dedicated his time in making the iOS app a working product. Throughout the project, Ahmed and Darshan took the lead with all security-based implementation, how data would transfer over the network, how passwords are encrypted, etc. Thanks to long hours and genuine interest in the development of this tool, Ahmed, Darshan, and Raj were able to meet all sprint commitments. Imagine the possibilities if, under this teams leadership and cohesion, what a Tapas 2.0 team of size five or seven could have accomplished!

Lessons Learned

Some obstacles we encountered include the lack of tutorials and training for external tools. This caused a major roadblock, especially for iOS development using React JS. It also caused some difficulties when trying to implement a serverless, Bluetooth version of Tapas 2.0.

In the end, Bluetooth connectivity was the one and only Learning Outcome (from Project Proposal) that we failed to accomplish due to Chrome extension limitations. On the side of team development, once the browser tasks were completed, Raj should have learned how to maintain the server, taking load off of Ahmed. At the time, Ahmed was managing both app development and server, so this change in responsibilities would have definitely helped lighten the load on one developer. Another major obstacle we ran into while improving our hashing algorithm, is the limited support for React Native. We tried switching to Argon2, bcrypt and scrypt, but those libraries do not have support to run in React Native environment. Which limited our options to crypto-js as it is the only cryptographic library that is supported and runs smoothly without any issues in React Native environment.

Recommendations to Future Students

First of all, strong communication between all team members is key. To help us out, we integrated Gitlab (repository management tool) notifications with Slack (team messaging tool). This sent a notification to all members when a ticket was pushed into code review or went for a merge request to master. Speaking of Gitlab, another recommendation is to know how to use git well. Making sure one knows how to work off branches, when to merge into master, how to stay on top of master are all helpful tips to making sure the project grows well when working in teams. Furthermore, we held weekly Google Hangouts calls where we screenshared our progress, asked and offered help wherever needed, and planned out steps for the next week. And finally, online meetings are great but in-person development sessions are key to syncing with your team members, merging parts of the tool together, and growing a positive company culture.

On the technical side, we suggest replacing Papa Parse with a lighter CSV parser. Papa parse offers many features not needed for our tool. A lighter CSV parser will help the tool perform strong on all devices.

On the phone side, we suggest using a different cross platform library than React Native. React Native is still new and not complete yet. It only supports the basic features, but some features we wanted to implement required advanced access to the phone features that React Native does not support yet. Also, it gave us a lot of issues when the building application for

iOS, which slowed our development cycle. For future students, maybe consider the advantages of building native application.

CONCLUDING REMARKS

The Tapas 2.0 team started off with the goal of building a secure and easy-to-use password manager tool. Specifically, our goal focused on laptop-based browser logins using dual-possession authentication with the user's primary smartphone. The purpose of this was to combat the current structure of password managers where if an attacker were to gain access to one device, they could potentially obtain all passwords. Thanks to dual-possession, this adds another crucial layer of security in retrieving passwords. On the other hand, it does take a few extra seconds to unlock phone and select which credentials to send. However, in this day and age of our digital life, it is critical that everyone use strong passwords. With Tapas 2.0, a user does not have to worry about what the password is and can mindfully use a different password on every account they own. Some limitations we encountered during development include:

- **Network:** Tapas 2.0 heavily relies on the network components of our application to be functioning. If the server or Google services go rogue, then all Tapas 2.0 users can experience a denial of service.
- **Smartphone:** At all times, the user needs to have their smartphone charged and with them to utilize the password manager, otherwise users won't have access to their passwords. However, if users have a backed up version of their passwords then they can download the app on a 'friends' phone and utilize the app from there.
- **Web-Logins:** Tapas 2.0 is limited to storing web-account credentials. This can be further expanded to add other confidential data that can be used on daily basis, such as but not limited to credit/debit cards, passport information, and SIN information.

Despite the list above, we truly believe that any inconvenience Tapas 2.0 causes will be well worth it in the long run.

Since the beginning, each step of this two-month project has taught us something new. Whether it be Chrome extension development, Firebase Cloud Messaging integration, implementation of PBKDF and HKDF encryption algorithms, or React app development, the list is long and valuable. Along with the technical aspects, we've learned how to effectively use Slack and Gitlab, and are confident that we can work well in teams. Right from Sprint one, we outlined our requirements, the main one being ask for help when needed and communicate as much as possible rather than work alone. We believe that, along with our passion for security and the various parts of this tool have helped us succeed with a team of three.

From Raj:

As mentioned earlier, I was very satisfied by the level of communication and openness within the group. Ideas could be tossed around and there was a good level of discussion for every idea put forth. Everyone had a voice and a chance to speak and everyone listened, gave constructive criticism and the members were always willing to help to make the project better. Two years ago, I thought of building a Chrome Extension but never got to it. So now, finally having the chance and a real reason to build one, it was quite fun. I did spend a lot of time on the UI side, more than I had wished. Nonetheless, my teammates were kind enough to walk me through what they did and how they did it, so I could learn their parts as well.

From Darshan:

This was a fun group of individuals to work with. We each have our unique strengths and together it made us a great team. Coming from a cyber-security background I was heavily interested in learning more about cryptography and how effectively it can be used in this project. I enjoyed researching the different algorithms and cryptography schemes that we could design into our application. Throughout the term, we came up with various security related architecture schemes and how each of the components would communicate, trying to get the best possible solution to work. Moreover, there was a lot of application development learned as I had the opportunity to get my hands wet in developing an iOS app (more of getting react-native to deploy onto iOS and learning about all the restrictions Apple has to offer without a paid dev license) and a Chrome extension.

From Ahmed:

It has been a pleasure working with Raj and Darshan. They are talented developers, and the best team I have ever worked with. I had to continuously push myself to the limit to match their expectations and hard work. I always wanted to learn about phone apps, how they are made and cross platform development, but never had the chance to due the constant overwhelming school work. But having done it now as part of my school work, I could tell from experience that iOS development sucks, and Apple needs to improve their services to help new developers get of the ground. It was a great experience, and I have learnt a lot in this course.

In conclusion, from where we started and what we expected the final product to be, we are very satisfied with the outcome. CSC490 has been a great course for us to explore any capstone project of our choosing as well as learn about other projects through the biweekly presentations.

Thank you Furkan & Serguei for your guidance.

- The Tapas 2.0 Team.

FUTURE WORK

One of the major features we wanted to implement but did not have a chance to is detecting phishing websites. Users might be prompted with links that take them to fake of phishing websites, which open the user to the vulnerability of sending their credentials to a wrong party. Our goal for this feature was to detect the difference between the legitimate websites that are stored and recognized by the devices and only allow password retrieval for secure and safe website. This would take the security of our application one step forward, and increases the trust in our application. Briefly looking into the possibility of this feature, Chrome has its own API that can help to detect the validity of website certificates, here is a [link](#).

Another feature that should be considered for Tapas 2.0 is a serverless communication between the phone and browser. In the current implementation, we rely on our server and Google services to be functional, without this the app will not work. By removing the rendezvous server

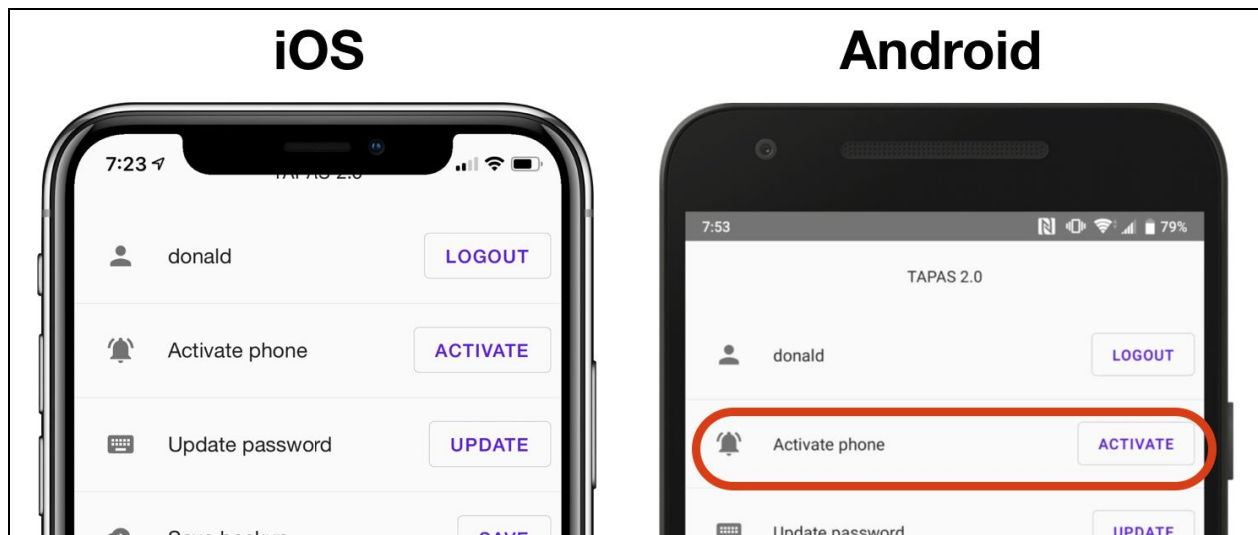
and Firebase service, we limit the attack vectors down to the point where an adversary must perform credential theft or steal both the laptop and phone to gain access to the passwords. Moreover, another advantage to this is that the user does not need to remember a master password in this case. The main reason for utilizing a master password was to allow users ability to manage multiple devices and backup & recover their passwords. To go about creating a serverless communication system, we would still need a 'server' like system for which the wallet and the manager (Chrome extension) can talk through. One solution is to host a python server on the same machine as the manger, Note this implementation can only work on personal devices. The python server would run on the local Wi-Fi network and capture the IP address of the machine. The manager always knows the location of the python server since it runs on localhost. When the user starts up Chrome, the manager would request the IP address and port information details for the phone. The manager would follow the same methodology of creating a shared secret (PBKDF Communication key) between the phone for communication trust. Along, with this the manager would generate its own one time "password key" that is used to encrypt/decrypt passwords onto the wallet. The pairing process would be the same except, the manager would generate a qr-code that encodes the local server's IP location of the computer's connected Wi-Fi network along with the communication key. Every time the user changes its local Wi-Fi network the user would have to regenerate the qr-code with the new IP location. This solution does of course decrease the ease-of-use in utilizing the application. However, looking at the trade-off between ease-of-use versus security, the security plays a greater role.

Enhance load time on the phone, currently as the phone database grows, the load time to display the entire list of accounts increases. We should partition the load into multiple steps instead of loading all sites at the same time. We can load the top 10 site, then show a button that says "show more". Additionally, we can implement search in the phone database instead of scrolling. This way if you look for a particular site, you can just look it up instead of scrolling until you find it. Also, for deleting sites from the phone, we should add the ability to select multiple sites by press and hold, then one delete button to delete them all selected entries.

APPENDIX

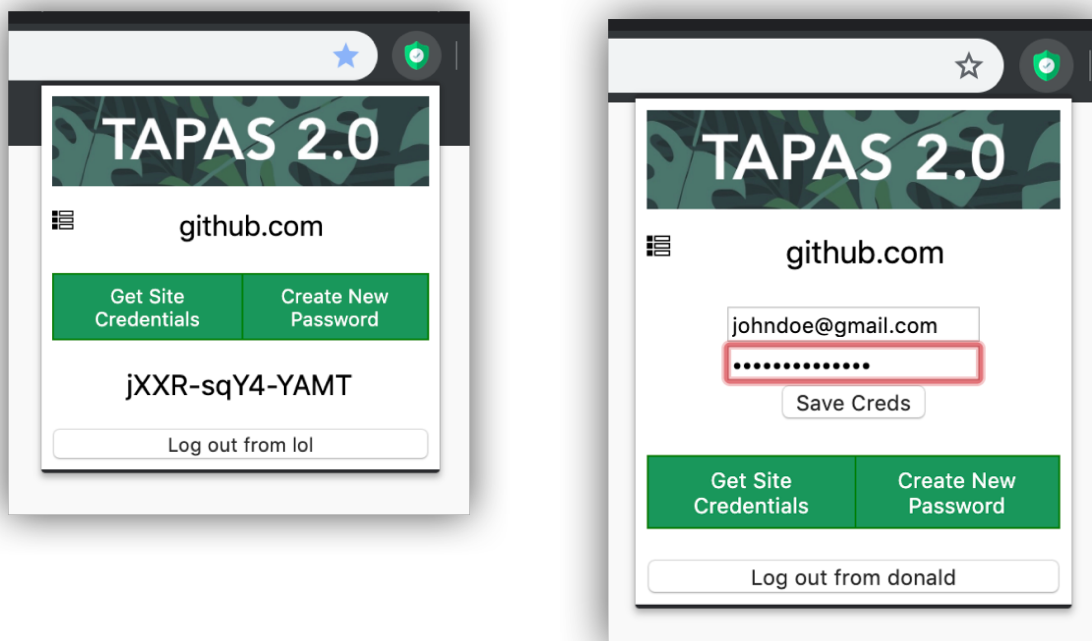
APPENDIX A

Activating a phone to users current account.



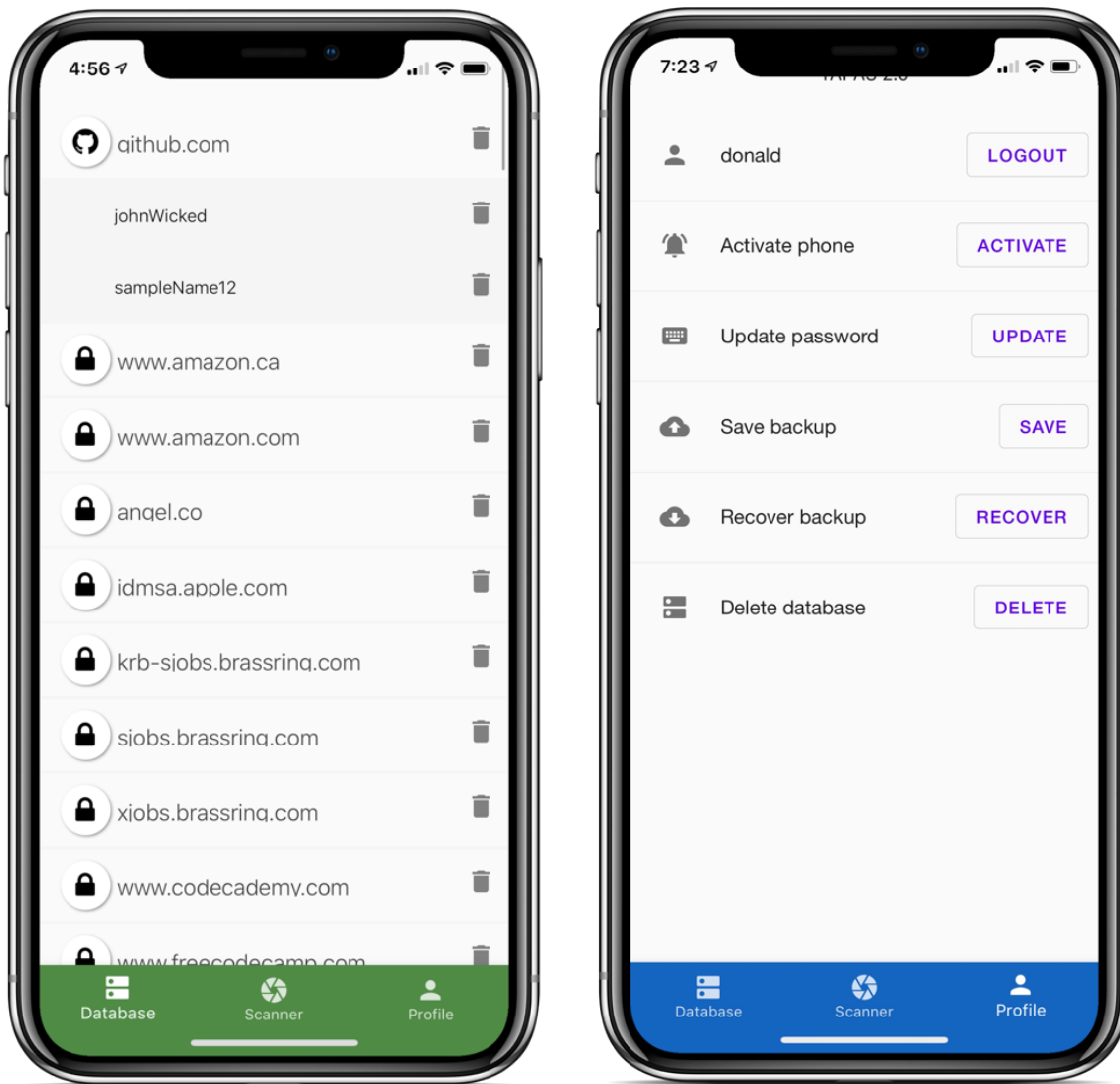
APPENDIX B

Manual password entry method to store password on wallet.



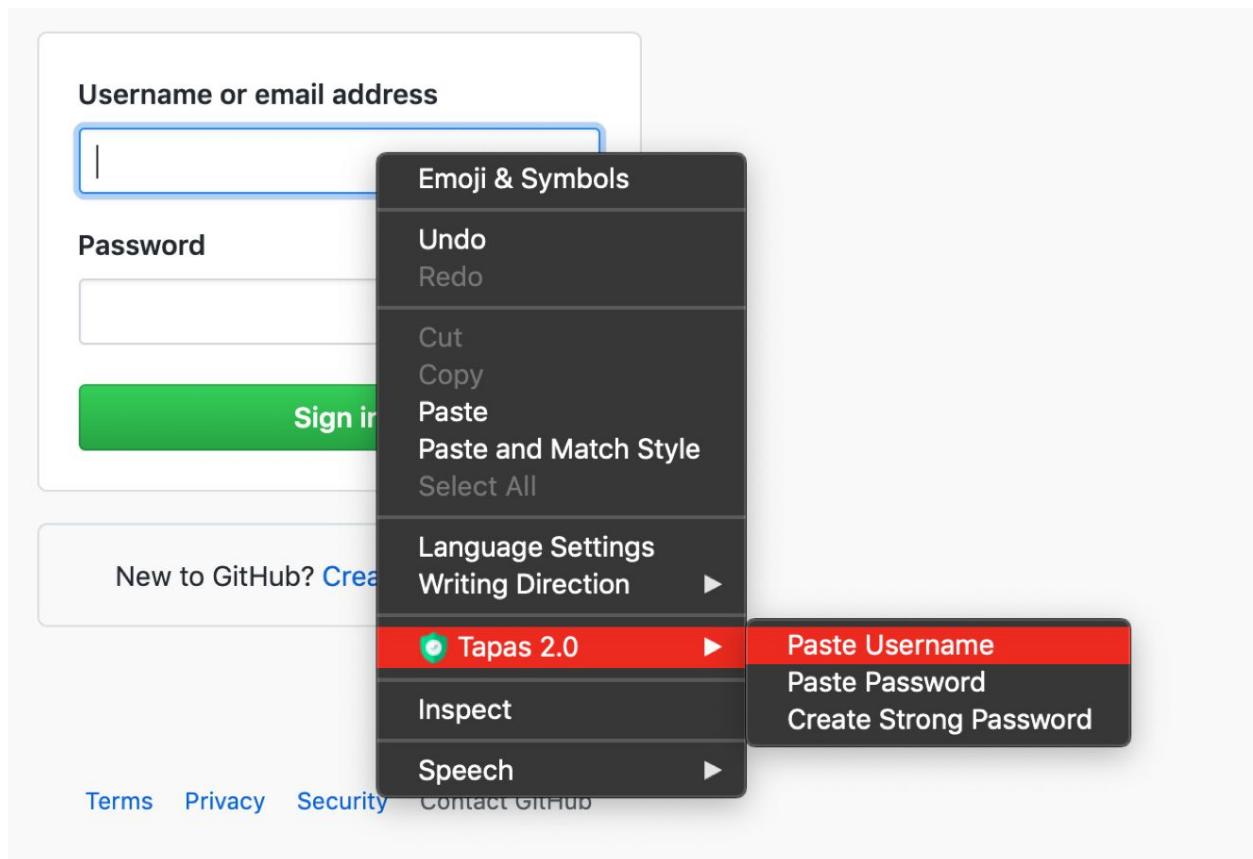
APPENDIX C

Tapas iOS app prototype view of wallet storage for different sites and view of user profile page



APPENDIX D

Alternate method for user to paste username and password into login field via context menus.



APPENDIX E

Attackers use hidden forms to fool password managers into autofilling usernames and passwords into the form fields.

```
<html><body>Thanks for taking our Survey!  
<form action="https://www.isecpartners.com" method="post" style="font-size:medium;  
margin:0px;font-family:Times;border:0px;padding:0px" target="_blank">  
  Do you like cats?: <input type="text" name="cats"><br>  
  Do you like dogs?: <input type="text" name="dogs"><br>  
<input type="email" name="Email" value="" style="max-height:0px;padding:0px;border  
-width:0px;width:0px">  
<input type="password" name="Passwd" style="max-height:0px;padding:0px;border-  
width:0px;width:0px">  
<input type="submit" name="signIn" value="Submit"></form></body></html>
```

Yahoo! Mail users running LastPass are the most vulnerable to credential theft. Any Yahoo! Mail user who has LastPass with auto-login enabled for the yahoo.com domain and views emails over HTTPS could have their username/password stolen just by opening the phishing email. When the email opens, LastPass will automatically “log in” and send the credentials to <https://www.isecpartners.com>. If a user only has “auto-fill” enabled, the credentials will still be stolen if the survey is submitted.

Gmail users are a bit better off, because Google will warn you that a form is about to be submitted before fulfilling the request, even if LastPass auto-login functionality is enabled. For Gmail users, a victim would still be vulnerable if they actually respond to the survey and have auto-fill enabled, or if they have auto-login enabled and click through the warning. Many victims will unwittingly submit their username and password to <https://www.isecpartners.com>. To give an idea of how successful a phishing campaign may be, compare the two screenshots of survey emails sent to a Gmail address¹⁰:

to me ▾


If you have trouble viewing or submitting this form, you can fill it out online:
<https://docs.google.com/forms/d/1LGwmRgxtwc4d5HNvEj3UPAYrFyD0BnpJGjlrC7r3cZY/viewform>

Better pet?

Do you prefer dogs or cats?

☐ Dogs
☐ Cats

Never submit passwords through Google Forms.

Powered by  Drive

This content is neither created nor endorsed by Google.
[Report Abuse](#) - [Terms of Service](#) - [Additional Terms](#)

Note: our references have been linked throughout the report.

Feel free to contact any one of the authors to receive a complete list of the references.

