Distributed and Secure Password Manager

6.857 Computer and Network Security
Spring 2021

Christina Chen *chrix@mit.edu*
Marc Felix *felixm@mit.edu*
Stephen Otremba *sotremba@mit.edu*
Nyle Sykes *nsykes@mit.edu*

Abstract

Password managers are software applications used to store passwords for various applications; they allow users to use unique, complex, randomly generated passwords for each of their accounts, without having to memorize each password. An overwhelming majority of the password managers today operate under a centralized model, where a single company maintains a user's passwords in a central database. This requires a user to fully trust the security measures chosen by the organization of their password manager. Additionally, users must also trust that a password manager will reliably and securely store their data, and ensure it is easily retrievable during possible network failures. We propose a decentralized, secure password manager system that requires zero trust in any central organization and provides a stronger data availability guarantees, while maintaining a similar level of security.

1 Introduction

There exist a wide array of password managers that follow a centralized model. Password managers face two primary challenges: fault tolerance, or resiliency against loss of data, and security, or resiliency against adversaries looking to access the data. Open-source solutions rely on crowd-sourced efforts to eliminate security vulnerabilities, but the majority of options available do not fall under this category. Given the lack of transparency in code bases, we highlight the difficulty in distinguishing a reliable password manager from an unreliable one.

There has been research analyzing the security of existing password managers, none of which are open source and all of which are centralized. [1] With over a dozen popular services analyzed, four did not encrypt the master password or their users' data in any way. Multiple others misuse available protection mechanisms, leaving their data vulnerable to cryptographic attacks such as rainbow tables. This leaves few options that provide a reasonable layer of protection beyond what is offered by the operating system on which a password manager operates. This paper describes a system designed to address the unreliability of the centralized model.

We present the design of a distributed and secure password manager, where all infrastructure is stored in an open-source repository. User data is stored in a decentralized manner, encrypted and distributed among multiple nodes. With a decentralized password manager, there is no requirement of trust from users, both with regards to the cryptographic security of the system and with regards to the accessibility of their data. By committing to an open-source model, we, like Bitwarden, allow users to audit the security of our system and maintain full knowledge of how their data is secured throughout the system. This gives users full confidence that their data is secure without requiring them to place trust in us. We also ensure a stronger reliability guarantee by reducing dependency on any single storage facility. These two characteristics work to create a system that requires minimal trust from its users while providing the same level of information security as existing centralized password managers.

2 Prior Work

There exist few projects within the distributed password manager space. The projects that do exist, for various reasons, do not satisfy our design objectives.

- **You.** is a password manager leveraging decentralized storage and messaging. This solution uses a mobile application to serve as the password manager and a Chrome extension to serve as a bridge between the mobile application and the computer. A push notification oracle built with Ethereum is used to notify the mobile application when the computer needs to access a password. Unfortunately, in this solution, the passwords are not stored in a distributed manner, but on the phone itself.
- **Safeguard** is a blockchain-based, decentralized password manager. Passwords are protected via a decentralized Blockstack ID. Blockstack is an open network for decentralized applications and smart contracts on Bitcoin. Unfortunately, this service relies on the Blockstack platform, and the Blockstack platform is built on Bitcoin, which has limitations when used as a first layer platform compared to other protocols like Ethereum.
- **Vaulilo** is a secure password manager also based on Blockstack. It is truly distributed, but like Safeguard, it has inherent limitations native to the blockchain platform on which it is built.

3 System Overview

In the following sections, we describe our design goals with a focus on the system's security policy.

3.1 Design Goals

We have three primary design goals for this system.

- The system must be **trustless**. A key drawback of centralized password managers is the need to trust some central authority, their encryption standards, their data storage protocols, and their associated actors in order to use their service. A distributed password manager must not require trust in any central entity.
- The system must provide a **guarantee of data resiliency**. As we are distributing the storage of passwords across a network of nodes, we must ensure that password data will always be available to users and resilient to both attacks and normal network usage.
- The system must provide a **comparable level of security** to industry-standard, centralized password managers. It is important that we not compromise on security, an essential element of any password manager, as we pursue the decentralization of the service.

4 Security Policy

4.1 Objectives

In order to alleviate the technical and security deficiencies of traditional, centralized password managers, we propose a design for a distributed and secure manager capable of storing the desired assets in a decentralized and fault-tolerant manner while maintaining cryptographic security. To ensure that such a system is completely trustless, we choose to implement the system using a decentralized peer-to-peer network. Doing so would protect a user's passwords and notes from data loss by redundantly storing these values across multiple machines within such a network, freeing the user from relying on a centralized storage system which may be susceptible to outages, data corruption, or worse, data compromise. In this policy, we outline some of the desired core functionalities of such a distributed system, including potential security risks associated with each function and the relevant actors involved in each process.

4.2 Definitions

We define a set of terms that we will frequently make use of throughout our policy:

- **User**: An entity that uses our password manager to store information.
- **Password/Asset**: A password or other piece of sensitive information that a user wishes to store in our password manager.

- **Node**: A physical machine used for storage in our system's distributed network. In our peer-to-peer network, this is an individual computer owned by a network participant.
- **Node Owner/Manager**: An entity capable of viewing and potentially modifying the data stored in a node. In our system, this is another user of the network.
- **Password Management Client**: The software running on user devices and storage nodes that enables a user to interface with the storage network. The client executes processes that allow users to store and retrieve passwords from our distributed network and processes that allow each user's node to act as a storage node for other users.
- **Username**: A user's unique identifier, potentially an email address or phone number, used within the password management client.
- **Master Password**: A password used by a user to authenticate themselves when using the password management client.
- **Bystander**: Any entity that is not part of the previously specified groups.

4.3 Account Creation

To begin interfacing with the password management network, we must facilitate the creation of a new node and an associated user account whenever someone wishes to join the system. This account should be comprised of some uniquely identifying username for each user and a master password that authenticates a user's identity.

4.3.1 Relevant Actors

- Users
- Nodes/Node Owners
- Bystanders

4.3.2 User's Desired Functionalities

When enrolling as a node in the password management network, a user should be able to provide some uniquely identifying username (such as an email or phone number), along with a *strong* master password that will be used to validate the user's identity in the future. The user should also be able to enroll in some sort of Two-Factor Authentication system. The user's machine should also be able to easily identify other nodes in the network in order to participate in the password storage scheme.

Once enrolled as a node in the network, the user should be able to participate in all system functionalities that the password manager provides. We will describe some of those functionalities later in this policy and in the implementation component of this paper.

4.3.3 Node Owner's Desired Functionalities

Existing nodes of the network should be able to contact any newly enrolled node of the network in order to use them for potential password storage.

4.4 Asset Storage

The most basic function that our network must perform is the storage of sensitive user information for future retrieval. This is the core utility of a password manager.

4.4.1 Relevant Actors

- Users
- Nodes/Node Owners
- Bystanders

4.4.2 User's Desired Functionalities

To store an asset with our system, the user should be able to simply enter the desired data in its raw form into the password management client after authenticating with their master password. The user should have the option to store 3rd party account information such as usernames and passwords or other data such as notes or financial information.

In order to be able to trust the system, the user should be guaranteed information security when storing assets. For example, any data transferred to storage nodes must be encrypted/secure. Raw asset data must *never* be visible to other node owners.

4.4.3 Node Owner's Desired Functionalities

Adding another user's encrypted asset to a node's machine should be a completely passive process for that node's owner. The node's owner should not have to approve each new addition, and associated metadata should handle the process of recording who owns each stored asset.

All asset data given to a storage node and its owner should be encrypted with keys that are inaccessible to the node owner or shared using a secure secret sharing scheme. The node owner should never be able to view raw asset data or any master password associated with another user.

4.5 Asset Retrieval

After storing an asset in the network, users should be able to easily retrieve and decrypt that information on their personal devices whenever desired.

4.5.1 Relevant Actors

- Users
- Nodes/Node Owners
- Bystanders

4.5.2 User's Desired Functionalities

A user should be able to determine the location of any stored assets and securely access that asset data whenever they desire. This should be possible from any of the user's personal devices which have the password management client installed. Furthermore, they should be able to access this asset data after providing only their personal node information and/or their master password (and potentially a Two-Factor Authentication code).

A user's retrieved asset data should only be decrypted on the user's personal machine. This decryption must *never* occur on another user's node. We do this to ensure that only the user has the ability to view their raw asset data.

4.5.3 Node Owner's Desired Functionalities

When receiving a request for retrieval of an encrypted asset, a node and its installed client should be able to find that asset in its memory/drive and produce to the querying user. This process should be passive and should not require a node owner to manually approve a retrieval request.

The node owner should only pass the encrypted asset back to the password management system and should never decrypt or otherwise alter the data prior to transmitting it to the requesting user.

5 System Design

5.1 Node Onboarding

A new node is added to our network when a new user creates an account. Similar to the design of private-IPFS, all nodes on our network store a network key. When a new node joins, it is given the network key.

The new node is also assigned a public key and a private key. We use RSA to generate this pair of keys. A node then generates its node ID, which is a cryptographic hash of its public key. The creation of a node ID allows different nodes on our network to authenticate each other. Node IDs on our network are generated through a SHA-256 multihash of a base64 encoded public key, a cryptographic standard that is also followed by IPFS currently [4].

If two nodes attempt to connect, they perform multiple checks to authenticate the other node. If one has a network key, the presence of the correct key is checked in the other. If both nodes verify that their network keys match, they then exchange public keys. Both nodes then verify that the node IDs match the hash of the exchanged public keys. Assuming all checks are passed, all communication between the two nodes is then encrypted using the keys they just exchanged.

Once the node is initialized, the master password is derived, as described in 5.3.1 and used to encrypt passwords. This process is detailed in section 5.3.2.

Additionally, when a new node is added to the network,

a hash of the node's ID to the node's network address is added to a distributed hash table stored across the network of nodes. This use of this distributed hash table is described in 5.5.

5.2 Password Generation

One of the key benefits of password managers is the ability for users to use lengthy, unique, pseudo-random passwords that would otherwise be difficult to remember without the aid of a password manager.

To accomplish this, our system uses a cryptographically secure pseudo-random number generator (CSPRNG), provided by the operating system on which the system is running, to generate a pseudo-random ASCII-string that can be used as a password. Since password generation is pseudo-random, there is a possibility of generating trivial passwords (e.g. "aaaaaaa"). To remedy this issue, we additionally employ open-source password-strength estimator *zxcvbn* [6] to ensure that generated passwords achieve a maximal security rating, regenerating the password when necessary.

5.3 Password Encryption

Once a secure password has been generated, the system can work towards securing it. The first step in this process involves local, symmetric-key encryption. At a high level, this is done in two stages:

1. Users enter their master password, and a master key is generated using PBKDF2.
2. Encrypt the password using AES-256 with the master key as an AES key.

5.3.1 Master Key Derivation

Since a user's master password can be variable length, and AES-256 requires a 256-bit key, we need a secure way to transform master passwords to 256-bits. We choose to do this using *Password-Based Key Derivation Function 2* (PBKDF2), which has five parameters: Psuedo-Random Function (PRF), Password, Salt, Iterations, and Key Length. We use SHA-256 as the PRF, the user's master password as the password, 100,000 as the number of iterations (industry-standard for open-source password managers [7]), and 256 bits as the key length. Using PBKDF2 with these parameters ensures that regardless of a user's master password, their master key will always be both the proper length for AES-256 and unique from all other master keys despite possibly being derived from a non-unique master password.

5.3.2 Encryption using Master Key

With a properly formatted master key generated, encryption of generated passwords is a relatively straightforward task. We can employ AES with a block size of 256-bits, which will output the ciphertext of the generated password. The use of this ciphertext is described in 5.4. As an additional security measure, we also flush both the master password and master key from memory once this process is complete, ensuring that this data is never stored for longer than necessary.

5.4 Secret Sharing

With a secure password encryption scheme designed, we can now discuss the manner in which this data will be stored in our distributed network of nodes. Since we want our system to provide an added level of security and fault tolerance relative to traditional password managers, our storage scheme must fulfill some key design goals:

- Our storage scheme should be **redundant**, meaning that multiple copies of the data must exist to ensure that the data is recoverable in the event that some copies are lost or compromised.
- The stored data should be **distributed** across multiple nodes of our network, thus ensuring that our data is safe in the event that any individual storage node is taken offline or otherwise made unavailable.
- The scheme should be **secure**, meaning that individual pieces of stored data should never reveal *any* information about the raw password that it encodes.

In order to fulfill these objectives, we leverage secret sharing to split our encrypted passwords into a number of shares, each of which are then sent to unique, randomly chosen nodes in the network. To ensure the security of such a scheme, we choose to use Shamir's Secret Sharing Scheme (SSSS), with our secret s being our encrypted password. [2] In order to easily verify

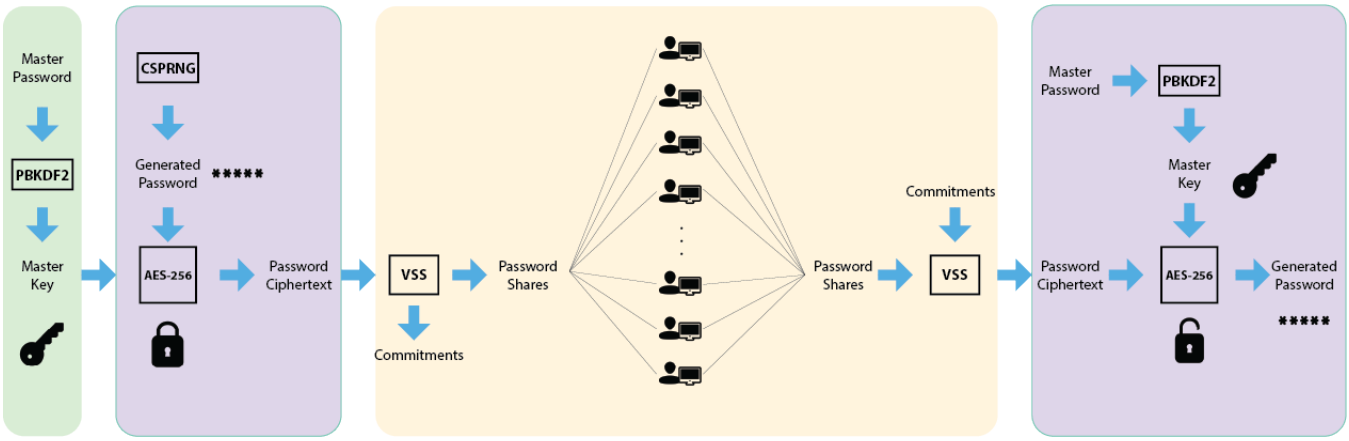

Figure 1: Password Life Cycle

that our shares are valid when reconstructing our secret, we extend Shamir's scheme using Feldman's Verifiable Secret Sharing (VSS). [3] Following these schemes ensures that each share of data reveals no information about the underlying password that it encodes, allowing us to store the shares in the network in a trustless manner.

5.4.1 Sharing Process

Following Feldman's VSS in this way would be done with the following procedure:

1. First, we choose a cyclic group G with prime order p for which the discrete log assumption holds, along with a generator of G. For this scheme, we choose p such that p is a Sophie-Germain prime and that $q = 2p+1$. We also choose G such that $G = \mathbb{Q}_q$, which has order p. We take g to be the generator for this group.
2. We then take our secret s and compute a random polynomial P of degree t with coefficients in $\mathbb{Z}_p$ without revealing it. We consider the computed polynomial to be $P(x) = s + a_1x + a_2x^2 + ... + a_tx^t$.
3. For some predetermined n and all $j \in \{1,2,...,n\}$, we pick a unique value x_j to generate a share that will be sent to an individual storage node. Our shares become $v_j = P(x_j) \pmod p$ for all j.
4. Finally, we compute commitments to the coefficients of the polynomial P in modulo q. These commitments are $c_0 = g^s \pmod q$ and $c_i = g^{a_i} \pmod q$ for $i \in \{1,2,...,t\}$. This gives us commitments c_0, c_1, ..., c_t.

Normally these commitments would be sent to shareholders along with the computed shares, but if we store these commitments on our user's machine, we can instead use them to verify that our returned shares are valid before reconstructing them to retrieve our encrypted password. We can now send these computed shares to nodes in our network to securely, redundantly store our passwords.

In our implementation, we choose values $t = 4$ and $n = 30$ via a probabilistic analysis described in section 6.3. This means that we send 30 shares of the secret and require at least 5 of them to successfully reconstruct it.

5.5 Password Storage and Retrieval

5.5.1 Password Storage

Given 30 shares, 5 of which can be combined to produce the ciphertext using our secret sharing scheme, we then store these shares randomly across the distributed network of nodes. The password storage protocol works as follows:

1. Randomly select 30 nodes from the network of nodes. This can be done similarly to how IPFS chooses nodes to store information (choosing a node with a hashed value closest to the hashed

value of a share). The exact mechanism is not important as long as it leads to proper load balancing across the system.

2. Send `passID = hash(username || entryID)` and `share i` to each node `i`. The `entryID` is unique to each entry, or password, for that user. It can be as simple as a time-stamp representing when that password was created, or it could be metadata describing what type of password is being stored. The first node will receive `passID` and the first share, the second node will receive `passID` and the second share, and so on.

3. Each node `i` will map `passID` to `share i` in their local storage.

4. Locally, the user will map `passID` to `[node 1 ID, node 2 ID, ..., node 30 ID]` and to [c_1, c_2, ..., c_{30}], where c_i is the commitment to the i^{th} node's share. This enables the user to know which nodes to query to retrieve the stored shares at a later point.

5.5.2 Password Retrieval

The password retrieval process is similar to the password storage process, but in reverse.

- The user computes `passID = hash(username || entry ID)` with the `entryID` corresponding to the password they wish to retrieve.
- The user references their internal storage and uses `passID` to retrieve the list of node IDs that are storing the shares, along with their relevant commitments.
- The user references the distributed hash table to retrieve the network addresses for each node ID.
- The user queries each network address and provides them with `passID`. Each node `i` will return to the user `share i`.
- The user verifies that each returned share is legitimate using the commitments c_i from our VSS procedure. After verification, we choose any 5 verified shares and simply reconstruct our secret to retrieve our stored encrypted password. The analysis in section 6.3 describes why this procedure should successfully reconstruct our secret with very high probability.

In the event that one of the shares returns as invalid, we consider the scheme weakened and immediately instruct our storage nodes to delete their stored shares of this password. We then redo our password storage procedure from section 5.5.1 by choosing a new sharing polynomial and new set of 30 storage nodes. We do this to minimize the impact of a potential adversary node that has returned an invalid share with malicious intent.

5.6 Password Decryption

Once the password ciphertext has been reconstructed from the shares, decryption can take place locally using the same symmetric-key structure as the initial encryption. Since neither the master key nor the master password are ever stored locally, the master key must be re-derived every time a user attempts to decrypt a password ciphertext. In order to generate the proper master key, PBKDF2 must once again be used with the same parameters described in section 5.3.1. With the master key derived, we can simply run AES-256 in decryption mode to retrieve the original password plaintext.

As an added security measure, once the password ciphertext has been decrypted, the plaintext is only allowed to live in memory until one of two conditions are met:

1. The user has indicated that they no longer require the password plaintext.

2. A timeout limit of 30 seconds has elapsed.

When either of these conditions are met, the password plaintext, master key, and master password are flushed from memory.

5.7 Node Removal

There are a few situations that lead to a node leaving the network. A node may become inactive for extended periods of time or engage in suspicious behaviour. When a node is removed from our network, it is treated as an adversary, and we ensure that the data stored in the node is rendered useless. As mentioned in section 5.4.1, our implementation of secret sharing allows ciphertexts to be reconstructed through 5 shares.

To eliminate any probabilistic likelihood that multiple removed nodes can reconstruct a cipher text together, for each share on a removed node, we recompute a new polynomial P of degree t with coefficients in $\mathbb{Z}_p$ for the related ciphertext. The newly generated shares are then sent to nodes within the network, and the old copy of shares are deleted. The node ID of the removed node is removed from local storage and removed from the distributed hash table mapping node IDs to network addresses. At this time, the node will no longer be accessible for retrieval.

6 Security Analysis

In the following sections, we analyze the security of various key components of our system.

6.1 Analysis of Network Composition

Our network is designed to be resilient against any malicious nodes, either outside or inside the network. One category of possible adversaries is foreign attack nodes, who may try to join the network or ask for information from the network. Our system's dependence on both network keys and a node ID ensures proper authentication before any two nodes connect, and remains secure against any such attacks.

A second category of adversaries is malicious nodes within the network. Our design requires a `passID` for the retrieval of each share, which ensures that only the node that generated the shares is able to later retrieve it. Even if the adversary becomes part of the network, they are unable to receive any new information they did not generate themselves.

Finally, we will add load-balancing across the network, to prevent any single node from launching a DDoS attack and overwhelming the network.

6.2 Analysis of Encryption

The security of AES with 256-bit blocks in a secure mode such as Galois/Counter Mode (GCM) is well-established and generally agreed upon, therefore we will not directly analyze the security of AES.

However, it is worth analyzing the added security of using PBKDF2 to generate the master key. Since AES is assumed to be secure, one of the next most feasible attacks is a rainbow table attack [5] where an adversary can precompute a table of "master" keys for common passwords then use those to check whether they result in a valid decryption. If we assume an adversary is able to obtain the necessary shares to reconstruct a password ciphertext at all, it is reasonably likely that the adversary would be able to determine the valid master key for a given ciphertext *if we used a naive key derivation function*.

PBKDF2 alleviates this concern in two ways. First, by including a salt in the key derivation process, we make a rainbow table attack exponentially more computationally intensive relative to the length of the salt (i.e. the user's username). Second, by setting the number of iterations used for key derivation to 100,000, we make the computation of a single key significantly more intensive, meaning each unique table will take longer to compute. [5]

These two factors effectively render any precomputation-style attack infeasible, further increasing the security of the decryption process.

6.3 Analysis of Secret Sharing

With a standard network of nodes storing shares in a secret sharing scheme, system designers must consider the possibility that any one node might be corrupted or malicious. In our system, we must consider these in addition to the possibility that any one node is offline and unable to respond to a query for information. Since nodes are users' personal devices, there is a nontrivial probability that a node may be a properly-functioning good actor that has a temporary loss of internet connection.

We assume that the probability that any one node is corrupted, malicious, or offline is 30%. We choose to store shares across 30 nodes and require 5 shares in order to decrypt the secret. We are thus able to compute the probability that any one query to the network for a password fails via the following expression:

$$\sum_{i=r}^{n} (1-p)^i \cdot p^{n-i} \cdot \binom{n}{i} = 99.99999998\%$$

$$\text{where } p = 30\%, n = 30, r = 5$$

This implies that any given user would need to query the network for a password 5.6 billion times in order to expect that one of those queries is unsuccessful. We believe that this provides the user with sufficient data resiliency, one of our key design goals.

7 Conclusion

Our implementation of a password manager is designed to provide at least the same level of information security as existing industry-standard, centralized managers, while storing the data in a distributed, fault-tolerant manner. We ensure password security through password encryption, RSA node authentication, and secret sharing. This security is maintained effectively through a network of nodes, eliminating the need for a central server or for trust in any one central entity. The redundancy of our system also creates a strong guarantee of data resiliency against both adversarial attacks and network failures.

8 Acknowledgements

The completion of this paper would not have been possible without the abundant support that we received from the 6.857 staff. We would specifically like to thank Ron Rivest and Yael Kalai for the enlightening lectures that provided the foundation of our work. We would also like to thank Andrés Fabrega and Kyle Hogan for their incredibly helpful project advice. Thank you all for an amazing semester!

References

[1] *"'Secure Password Managers' and 'Military-Grade Encryption' on Smartphones: Oh, Really?."* Belenko and Sklyarov, https://www.elcomsoft.com/WP/BH-EU-2012-WP.pdf.

[2] *"How to Share a Secret."* A. Shamir. Commun. ACM, 22(11):612–613, 1979.

[3] *"A Practical Scheme for Non-interactive Verifiable Secret Sharing."* P. Feldman. *In IEEE FOCS'87*, pages 427–437, 1987.

[4] *"IPFS And Privacy."* https://docs.ipfs.io/concepts/privacy/.

[5] *"Making a Faster Cryptanalytic Time-Memory Trade-Off."* Oechslin, P. Advances in Cryptology - CRYPTO 2003. LNCS. 2729. pp. 617–630.

[6] *"zxcvbn: Low-Budget Password Strength Estimation."* D. L. Wheeler, https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/wheeler.

[7] *"Bitwarden Security Whitepaper."* Bitwarden. https://bitwarden.com/images/resources/security-white-paper-download.pdf.