Akord

# A protocol for private, permanent and composable storage vaults

A TECHNICAL LITEPAPER | VERSION 0.1 | SEPT 2022

By Richard Caetano and Weronika Kolodziejak

# Contents

# Introduction

Since the early 2010s, cloud storage has been the predominant means of storing user data such as profiles, emails, files and messages.

It has proven to be inexpensive, efficient and easy to integrate across a broad range of applications running in software-as-a-service, enterprise and government infrastructure.

The cloud is able to quickly deliver data, real time, to a range of devices – desktop, mobile, even your watch. However, as we've seen, centralised cloud services pose significant risks to the individual:

- **Privacy**: most software-as-a-service providers, at a minimum, have a backdoor to customer data at rest, in transit or when processed.
- **Censorship**: SaaS providers can revoke access to your data, especially if forced to by a government or legal judgement.
- **Lacking consent**: your data is often bought and sold, analysed or traded, without your consent or compensation.
- **Surveillance**: nefarious actors, governments and cyber criminals can exploit customers by hacking data storage providers.

## Web1-3

To quickly review how we got to web3.

**Web1: read–only**
The web as we used it in the beginning, static and searchable. Users had the right to connect and browse. In other words, Web1 was 'read only'.

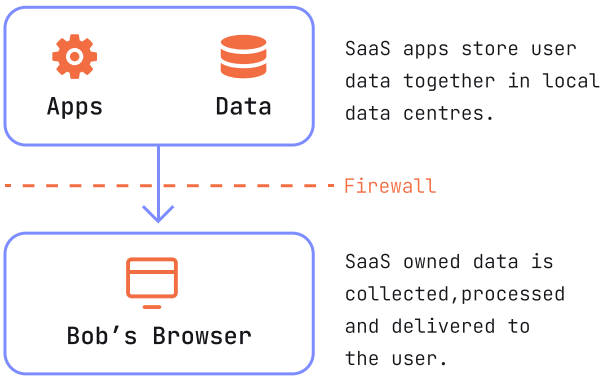
**Web2: read–write**
The web when social media scaled to billions of people. Users generated content with their right to post, publish, like and share. In other words, Web2 was 'read and write'.

**Web3: read–write–own**
The web as developed today where blockchains form the internet layer of value. Data, money and contracts are secured and integrated by decentralised apps (dApps), which are user owned. In other words, Web3 is 'read, write and own'.

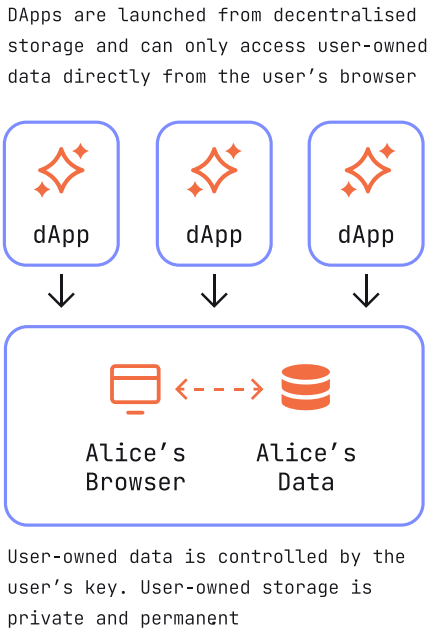
## Data ownership

Core to Web3 is the right for users to own their data and have complete control of who can access it. This fact alone makes us re-consider the architecture we use for building distributed applications, or dApps.

Web2 applications, running as a SaaS product, assume customer data is stored behind the firewall in order to process and deliver their application. This architecture requires the SaaS provider to have full access to the user's data.




However, dApps require an architecture where the application never stores user data, rather the dApp requests access to it from the client's browser.




By relying on end-to-end encryption and locally running dApps, the user can have much greater control over their data – unlocking user-controlled data ownership.

# Web3 storage vaults

In this paper we present a protocol that enables builders to deploy user-controlled, web3 storage vaults. Facilitating data ownership, they have the following functionality and properties.

- **Privacy** secured by end-to-end encryption.
- **Consent**: access controlled by the user's consent.
- **ZKP**: containers for verifiable claims and zero knowledge proofs.
- **Composable and extendable** protocol, enabling custom extensions.
- **On-chain permanent storage** for files and folders.
- **File versioning and auditing**.
- **Messaging and collaboration**.

As an abstract protocol, vaults require that all members are guaranteed access to consensus. Thus, the protocol can be best implemented over a blockchain network, such as Arweave or Filecoin.

## Design overview

**Vaults**. Private, permanent and composable storage units. Accessed directly from the browser, vaults require client-side encryption. Keys used to encrypt and sign are protected by the user's wallet.

**Members**. The vault includes one or more members who can read and/or write data in the vault.

**Nodes**. Data in the vault is organised by nodes. Nodes are abstract pointers to data and can be used to construct a hierarchy or graph of data.

## Keys

Associated with all vaults are a set of keys held by the members. Below is an overview of the keys used to build a vault.

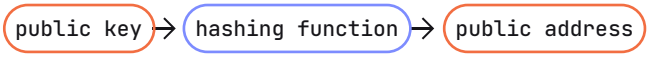
**User keys**. Two pairs of keys coming from the user's wallet:

- encryption private & public key
- signing private & public key.

**Vault keys**. A public & private key pair generated by the vault owner upon creation of the vault. This key can be rotated.

**Access key**. A symmetric key used to encrypt the data (files, memos, filenames, user's info).

**Public address**. Each member is identified by their public address, a special hash that is generated from the member's public key.




We cannot extract public keys from the public address. Therefore, all parties involved with the vault must share the public key between themselves to use the protocol.

## Transactions

Vaults are built as a series of transactions stored on-chain as a transaction log. Every state change to the vault is encoded as a transaction.

We'll define the required transaction set as follows.

- `vault:create`
- `vault:update`
- `member:accept`
- `member:revoke`
- `member:change-role`
- `node:create`
- `node:update`
- `node:delete`

With this base set of commands, we can construct a dynamic vault that can accept contributions over time. These commands can be extended or replaced by commands designed with specific requirements.

## Members

Vaults can serve one or more members. Members can be assigned one of three roles:

- **Owners** can control access, can archive and can transfer the ownership of the vault.
- **Contributors** can publish new documents, assets, etc to the vault.
- **Viewers** can access the vault, only to view content.

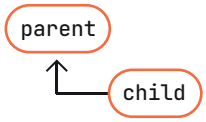
Vault keys are rotated when a member leaves or is removed, ensuring that access is revoked. Users only have permanent access to the data they could

see while a member of the vault, and not to any data added after they are no longer a member.

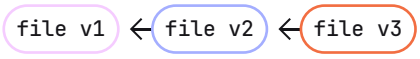
### Nodes

Within the vault, nodes are used as pointers to data, files, documents, etc.


Nodes can point to themselves to form hierarchies such as files and folders.




They can provide revision control for documents and/or other files.




## Building a vault

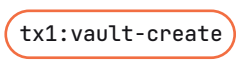
Next we will take a look at the steps to create and interact with a vault. Each step will correspond to an abstract command that is implemented as a transaction. The transaction log can then be used as provenance of the origin and state of the vault.

### Initialise the vault

All vaults begin with an initial transaction to encapsulate the signatures and/or proof of the following steps.

1. The vault owner generates a vault key pair (public & private key).
2. The private key is encrypted with the owner's encryption public key.
3. Transaction is submitted encapsulating the encrypted vault keys and public address of the owner.

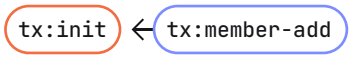
The resulting transaction log would contain:




### Adding/revoking membership

**Adding new members** to the vault requires a key exchange to validate the member.

1. Owner verifies the validity of the new member's identity and associated public key.
2. The vault owner decrypts the vault private key with their encryption private key.
3. The owner encrypts the vault private key with the invitee's encryption public key.




**Revoking access** to encrypted data is often not possible, especially when data is stored on-chain, because an encryption key cannot be unseen.

However, we can rotate the encryption key to restrict access to future data sets.

1. Owner generates a new vault key pair.
2. The new vault private key is encrypted for all active members.
3. All data exchanged in the vault are henceforth encrypted with the new public key.


### Contributing data

To respect the privacy rights of the vault members, data must be encrypted at rest, during transit and when processed. Therefore, encryption is performed by the client to ensure end-to-end encryption.

Each member in the vault has a public key, and subsequent public address, which they can use to exchange messages and validate signatures.

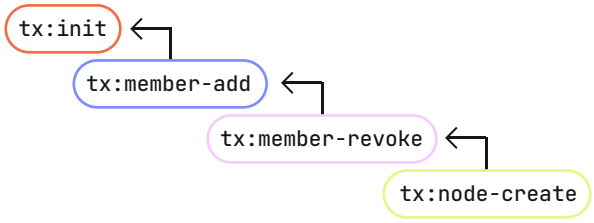
**Vault key**

A vault key is used to encrypt/decrypt the data, values, keys within a vault. When added to a vault, members share an encrypted copy of a vault key.

Members can then use the vault key to exchange data with other members in the vault.

1. Alice and Bob are both members of a vault.
2. Alice wants to share a file within the vault.
3. Alice generates a unique access key.

4. Alice encrypts file with the newly generated access key.
5. Alice encrypts the access key with the vault public key.
6. the encrypted file and encrypted access key are stored.



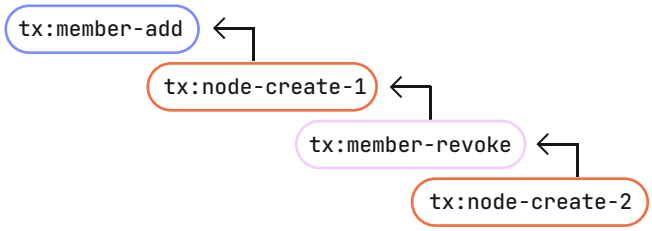

Later, Bob can access the file.

7. Bob decrypts the vault private key with their encryption private key
8. Bob decrypts the access key with the vault private key
9. Bob decrypts the file with the access key

## Dynamic access control

If a vault contributor attempts to include an invalid transaction, the clients can simply ignore the transaction and the vault owner can revoke access.

With the client accepting/rejecting transactions based on the vault's rule set, a decentralised consensus can be formed with other clients. This consensus forms the 'single source of truth' for the vault.




In this set of transactions, the member added and removed would only have access to the first node.

## Custom commands with nodes

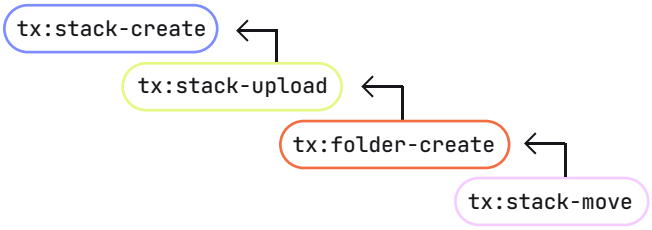
The vault can be constructed with an abstract pointer called a node.

Nodes function as pointers because they can reference:

- an external, public or private, data set, image or file;
- another node to form a hierarchy, such as a folder system;
- a node that is a previous version of itself, enabling file versioning;
- zero knowledge proofs, enabling privacy preserving verifiable claims;
- contracts, tokens and NFTs, enabling collections, ownership, and more.

To implement a file system, you can extend the protocol with the following commands to construct a 'file stack'.

- **`stack-create`** – storing the file hash, timestamp, version.
- **`stack-upload`** – storing a new file hash, timestamp, and incremented version.
- **`folder-create`** – create a new node to function as a folder.
- **`stack-move`** – update the file to list under the existing folder.




In this chain we show a folder was created and a file uploaded to it. Finally, a second revision of the file was uploaded, resulting in a stack with two versions.

## Quantum resistance

With a hypothetical quantum computer, it is possible to break elliptic curve encryption using the Shor's Algorithm. Thus the threat is a quantum computer advanced enough to execute this attack.

To mitigate this issue when working with permanent blockchains, it is recommended to mask public keys using a hash function to generate a public address when using elliptic curve encryption. It is extremely difficult to extract a public key from a public address and without the public key, it is not possible for a quantum computer to break the encryption.

Another approach is to use a more quantum secure algorithm, such as Lattice-based cryptography.

Web3 will bring a long tail of uses and applications to web3 vaults. Delivering the proper infrastructure and cryptography will be key to any implementation.

# Conclusion

Decentralised and permanent storage is central to Web3. It unlocks dApps, user ownership of data, censorship resistant protocols and more.

However, for dApps, as well as SaaS and enterprise, to fully adopt a paradigm where user ownership of data is central, application developers will need to rethink how they access user-owned data.

In our paper, we presented the concept of private web3 storage vaults, storage units that can exist on permanent storage blockchains like Arweave.

Vaults are composable and can be customised for any application's data needs. Vaults can share data between applications, contain user account and profile data, or function as a data escrow device.