Guardian Recovery

Status	In progress
Owner	Ⓔ Eyob Melaku

Abstract

This document describes Payy's guardian recovery system, which replaces traditional iCloud and Google Drive based key backups. Instead of requiring users to store or remember seed phrases, the system allows users to register multiple authentication methods ("guardians") and recover their wallet by re-authenticating with a configurable subset of them.

The system is non-custodial by design: no single party or guardian can recover a user's private key on its own. By supporting multiple guardians and a configurable recovery threshold, the system remains secure even if a single guardian is compromised, while enabling seamless recovery in the event of device loss.

Motivation

When using Payy, numerous users have failed to backup their seed phrases leading to the loss of their funds. Although solutions like saving their private keys in storage layers like iCloud and Google Drive are available, most users either ignore or are unaware of how the backup is done. In addition, Cloud-based backups are often opaque to users, inconsistently implemented across platforms, and prone to silent failure, resulting in unrecoverable key loss.

The guardian recovery system aims to remove the problem of having to save seed phrases or using a single storage layer to store the private key. It utilizes already existing authentication methods (will be referred to as guardians from here on) such as google, passkey, wallets, sms, email and kyc to verify and retrieve the user's private key should the unfortunate event of losing their device happens.

There will not be a single party that can recover the private key, thus making it secure against any custodial risks or attacks. Should the user only have access

to certain guardians, they could still recover their private key given they meet a certain threshold configured during registration.

Specification

Terms

- **Guardian**: An authentication method which will be used to verify/authenticate a user. It's composed of:
  - `ipfs_cid`: the `ipfs_cid` for the authentication logic that will be executed in a litAction used to verify the user.
  - `auth_value`: the value that is returned by the guardian once the user successfully authenticates. This value can only be returned by the same user when they successfully authenticate (eg. password hash, oauth_id for google, sumsub id for kyc, ...). On-chain, `auth_value` is never stored directly for reverse lookup. Instead, the system stores `keccak256(auth_value)` when populating `authValueToAddress`.
- **Ciphertext**: The encrypted output produced from the user's private key. It is stored alongside `data_to_encrypt_hash`, a hash derived from the private key and the access-control conditions under which decryption is permitted. During the recovery phase, both values are used to reconstruct and decrypt the user's private key.

Components

Database

The database which is used to store guardians, ciphertexts and their relations. Tables in the database will be:

- `guardians`: this table stores the available guardians that can be used. They include:
  - ID: unique identifier
  - ipfs_cid: the `ipfs_cid` representing the authentication logic
  - name: their unique name (e.g. `PASSKEY`)

    - display_name: name used for displaying to the user (eg. `Passkey`)
    - is_active: boolean value that shows whether the guardian is enabled or not
- `ciphertexts`: this table stores the ciphertexts of the encrypted private keys:
    - ID: unique identifier
    - address: respective address of the user the ciphertext (of the private key)
    - ciphertext: the ciphertext string retrieved after encrypting the user's private key
    - data_to_encrypt_hash: string retrieved after encrypting the user's private key, needed alongside the ciphertext to retrieve the user's private key
- `guardian_to_ciphertext`: this table stores the many-many relationship between the guardians and ciphertexts. It shows which guardians have been registered for which user:
    - guardian_id: the respective guardian ID
    - ciphertext_id: the respective ciphertext ID
    - guardian_auth_value: the `auth_value` that was returned by the guardian when the user authenticates

Smart Contract

The smart contract, deployed on Polygon, stores a GuardianConfig for each user address. Each user address maps to a single on-chain `GuardianConfig` (`address → GuardianConfig`). Each `GuardianConfig` additionally maintains an enumerable list of active guardians to allow wallets and off-chain systems to inspect the user's registered guardians directly from the smart contract.

Payy uses two smart contracts:

1. **GuardianConfig Contract (Polygon):** stores all per-user guardian recovery state (thresholds, guardian lists, and verification mappings).

2. **LitAction Registry Contract (Ethereum):** stores and serves the **child** `litAction` `ipfs_cid` used by the recovery flow.

GuardianConfig Contract (Polygon)

The Polygon contract stores a `GuardianConfig` for each user address `(address → GuardianConfig)` and exposes getter functions for wallets and off-chain systems to read:

- threshold
- `guardianCID`s (enumerable list of active guardian hashes)
- `guardianEntries` lookups for a given guardian hash
- `authValueToAddress` reverse lookup for a given keccak256(auth_value) (or salted hash if used)

Each `GuardianConfig` contains the following fields:

1. **threshold**

The minimum number of guardians required to decrypt the user's private key. Thresholds are calculated on addition and removal of guardians as follows:

- If there is only one guardian, the threshold is set to `1`
- If there are `n` guardians, the threshold is set to `n / 2`
- If the number of guardians is odd, the threshold is set to `(n + 1) / 2`

2. **Guardian CID hashing**

The guardian's `ipfs_cid` is hashed using `keccak256` and used as a `bytes32` key in the mapping below. This is done because ipfs cids are variable-length and inefficient to store directly on-chain. Hashing produces a fixed-size bytes32 value suitable for mappings and indexing.

3. **guardianEntries**

Is a mapping of the guardian's `ipfs_cid` hash and the `auth_value` (`keccak256(guardian's ipfs_cid) → auth_value`). A mapping is used instead of an array here to make existence checks and removals straightforward and efficient. With a mapping, the system can quickly determine whether a guardian is already registered and remove it directly, without having to iterate over a list, thus effectively making removals `O(1)` instead of `O(n)`.

4. guardianCIDs

guardianCIDs is an array of bytes32 values representing the hashed `ipfs_cids` of all active guardians registered for the user.

- Each entry corresponds to `keccak256(guardian_ipfs_cid)`
- It allows wallets and indexers to retrieve the user's active guardians directly from the smart contract

5. authValueToAddress

`authValueToAddress` is a mapping from the hash of a guardian `auth_value` to the user address it belongs to:

- key: `keccak256(auth_value)` (stored as bytes32)
- value: the user's wallet address (address)

This mapping enables reverse lookup during recovery flows where the system starts from an authenticated guardian `auth_value` and needs to identify which wallet address is associated with it.

Because `auth_value` may contain sensitive identifiers (e.g. OAuth subject IDs, KYC IDs), only its hash is stored on-chain.

LitAction Registry Contract (Ethereum)

The Ethereum contract stores the **child** `litAction` `ipfs_cid` and exposes a getter so the recovery flow can fetch the currently active child logic.

- It does **not** store user guardian state
- It does **not** store thresholds, guardian lists, or auth mappings
- Its sole responsibility is returning the child `litAction` CID

***NB*: Getter functions**

Getter functions (Polygon GuardianConfig contract)

- getter returning `guardianCIDs` for a user
- getter returning `auth_value` for a given guardian hash (lookup into `guardianEntries`)
- getter returning address for a given `auth_value` hash (lookup into `authValueToAddress`)
- getter returning threshold

Getter functions (Ethereum LitAction Registry contract)

- getter returning the active child `litAction` `ipfs_cid`

Lit

Lit is used as a distributed trusted execution environment (TEE) for encryption and decryption of the user's private key. Lit provides an sdk with two functions:

- `encrypt`: encrypts the passed in string (in this case the user's private key) and returns the `ciphertext` and the `data_to_encrypt_hash`
- `decrypt`: takes in the `ciphertext` and `data_to_encrypt_hash` and upon successful execution of the `litActions` (explained below) collects shards of the private key and decrypts it on the user's app

  In order to combine the shards and decrypt the private key, an object known as an access control condition is passed. This access control condition includes

  - **`contract_address`**: The `ipfs_cid` of the `litAction` whose execution result determines whether decryption is authorized
  - **`method`**: The function within the Lit Action to invoke
  - **`return_value_test`**: A rule that determines whether the Lit Action's return value satisfies the authorization criteria by applying a comparator to a target value

- `comparator`: A comparison operator (`>`, `<`, `=`, `>=`, `<=`) applied to the `litAction`'s return value
- `value`: The value against which the `litAction`'s return value is compared

For running pieces of code in the TEE, Lit has what are known as Lit Actions (`litAction`). `litActions` are Javascript code that run within lit protocol in an isolated Deno environment. In order for these `litActions` to be executed, they must be hosted via an IPFS pinning service.

Lit requires a session signature to execute Lit Actions, the session signature determines the Lit fee payer and can be used to add additional access controls (however this is not required for this use case).

In order to prevent abuse of the Payy gas balance, the session signatures will be generated on Guild and returned when requested to the client. The session signature will be generated with a one time limit, expiry and will be scoped to only call the parent Lit Action, in order to prevent abuse. Any reuse or invocation outside the permitted scope is rejected by Lit, preventing replay or abuse.

The following `litActions` work together in layers, with each layer handling a specific part of the recovery flow. There are 3 layers of `litActions` used:

Parent litAction

The parent `litAction` acts as the coordinator. For the provided user address, it retrieves the user's `GuardianConfig` from the **Polygon GuardianConfig contract**, and retrieves the child `litAction` `ipfs_cid` from the **Ethereum LitAction Registry contract**, then executes the child `litAction`. Upon successful execution, it returns a truthy response indicating the user's legitimacy and authorizing the decryption of the user's private key.

This setup allows the recovery logic to remain flexible. By storing the child `litAction` `ipfs_cid` in the **Ethereum LitAction Registry contract**, the system can update or replace the child logic without changing the parent `litAction` (which is immutable).

Child litAction

The child `litAction` manages guardian verification and enforces the recovery threshold. It passes the relevant guardian `ipfs_cid`s and user `auth_value` to their corresponding guardian `litActions` and then collects the authentication responses returned by each guardian to check whether the number of successful authentications meets or exceeds the configured threshold for that specific address. Each guardian authentication is evaluated independently, and no ordering or dependency between guardians is assumed.

If the threshold is met, the child `litAction` will return a truthy response to the parent `litAction` indicating that the user is legit and can recover their private key.

Guardian litAction

Each guardian `litAction` focuses on a single task: verifying the user for a specific guardian. It:

- validates the user's `auth_value`
- performs authentication logic to confirm that the user is who they claim to be

If the verification succeeds, the guardian `litAction` returns a truthy response to the child `litAction`. Otherwise, it returns a falsy response.

Guild

Guild is Payy's backend service and has three primary responsibilities:

1. Reading from and writing to Payy's database, including tables used to track ciphertexts and guardian mappings.
2. Generating and returning a Lit session signature (`sessionSig`) that authorizes the wallet/app to execute the required `litAction`.
3. Relaying user requests on behalf of the wallet/app, so gas costs are paid by Payy rather than the user.

Session signatures are generated by Guild (instead of inside the user's wallet/app) for two reasons:

- **Payy pays for Lit execution, not the user.**

  Since Payy is covering the cost of running the `litAction`, the session signature must be issued in a way that aligns with Payy's billing and operational

control.

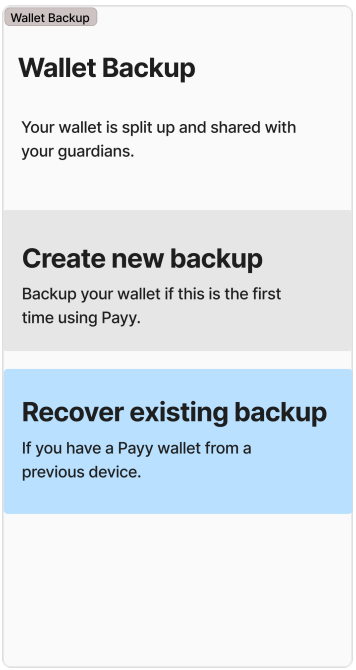
- **We avoid exposing the payer wallet in the client.**

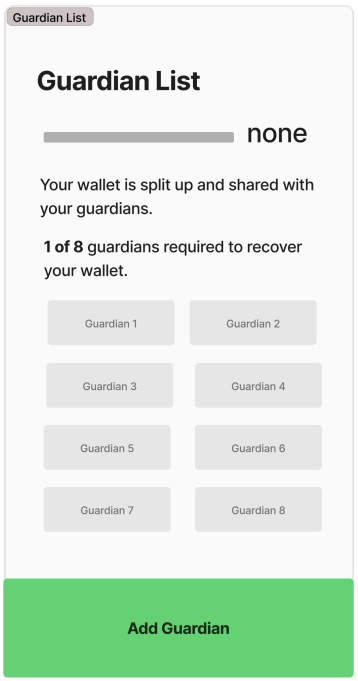
  If the wallet responsible for paying Lit execution fees were embedded or hard-coded in the user's app, it would be much easier to extract and abuse. Keeping session signature generation on the backend reduces the risk of the payer wallet being compromised.

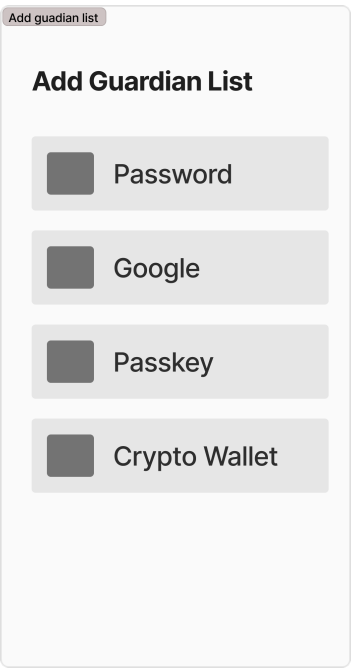
UX

The following UX mockup screens will be implemented in the wallet:

- **Wallet Backup** - lists the options for backup for a user without existing recovery registered
- **Guardian List** - shows the list of currently registered guardians, allows addition/removal of guardians
- **Add Guardian List** - shows the list of guardians the user can choose to authenticate and thus effectively make them a "Guardian" of their private key







Status Identification

Determine whether a `ciphertext` already exists for the user's address. If ciphertext exists for the address, this means the user has already backed up and has registered guardians for their private key, then the user will be routed to the Guardian List screen will be shown with all the registered guardians to the user for modification (addition/removal of guardians).

If there's no existing address, the user will be given the options to:

- **Create new backup**: if this is a new wallet and they want to backup their private key to prevent loss of funds
- **Recover existing backup**: if they've lost access to their device and want to recover their funds

This step will have a boolean variable - `existing` which will be passed to subsequent steps to indicate whether the user has backed up their private key before or they're creating a new backup. If it's a new backup `existing` will be false, if not it'll be true.

Create new backup

Guardian authentication

After the user selects a guardian, they're redirected to the guardian's authentication screen where they'll be prompted to authenticate with their chosen guardian. Once authentication succeeds, the guardian returns an `auth_value` and its `ipfs_cid`. Guardian `auth_values` are assumed to be stable identifiers for the duration of recovery.

Private key encryption

A private key is encrypted exactly once. Subsequent guardian additions or removals never trigger re-encryption. This will be detected in Status Identification step (where `existing` is false).

Lit's `encrypt` function takes the user's private key and returns a `ciphertext` along with a `data_to_encrypt_hash`. Both values are then stored in Payy's database in the `ciphertext` table.

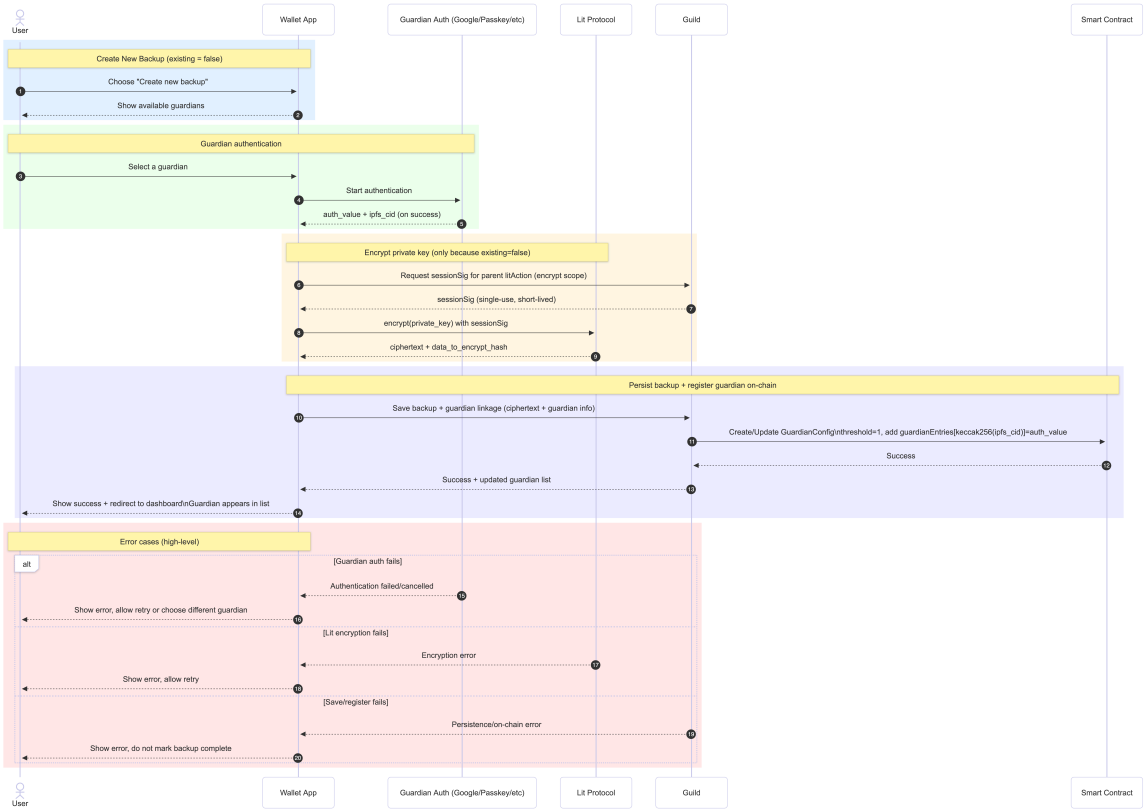
Then the guardian's `ipfs_cid` and `auth_value` are registered on the smart contract as follows:

1. The guardian's `ipfs_cid` is hashed using `keccak256`
2. The resulting hash is appended to the user's `guardianCIDs` array
3. The hash is used as the key in `guardianEntries`, with the `auth_value` stored as the value

By then a new `GuardianConfig` with the configured threshold (in this case will be 1, since only one guardian will exist during new backup creation) and `guardianEntries` mapped to the address will be stored in the smart contract.

Then the `ciphertext` is linked to the guardian in the database via the `guardian_to_ciphertext` table. This mapping shows what/which guardian is associated with a `ciphertext` and will be later used as a reverse lookup during the recovery phase.

Finally, the user is shown a success message and redirected to their dashboard, where the newly added guardian appears in their list of registered guardians.



Recover existing backup

User Identification

After the user selects a guardian, they're redirected to the guardian's authentication screen where they'll be prompted to authenticate with their chosen guardian. Once authentication succeeds, the guardian returns an `auth_value` and its `ipfs_cid`. The `auth_value` will then be used to perform a reverse lookup in the database and identify which private key the user is trying to recover by searching the `guardian_to_ciphertexts` table with the matching `auth_value` and joining with the `ciphertexts` table to retrieve the address.

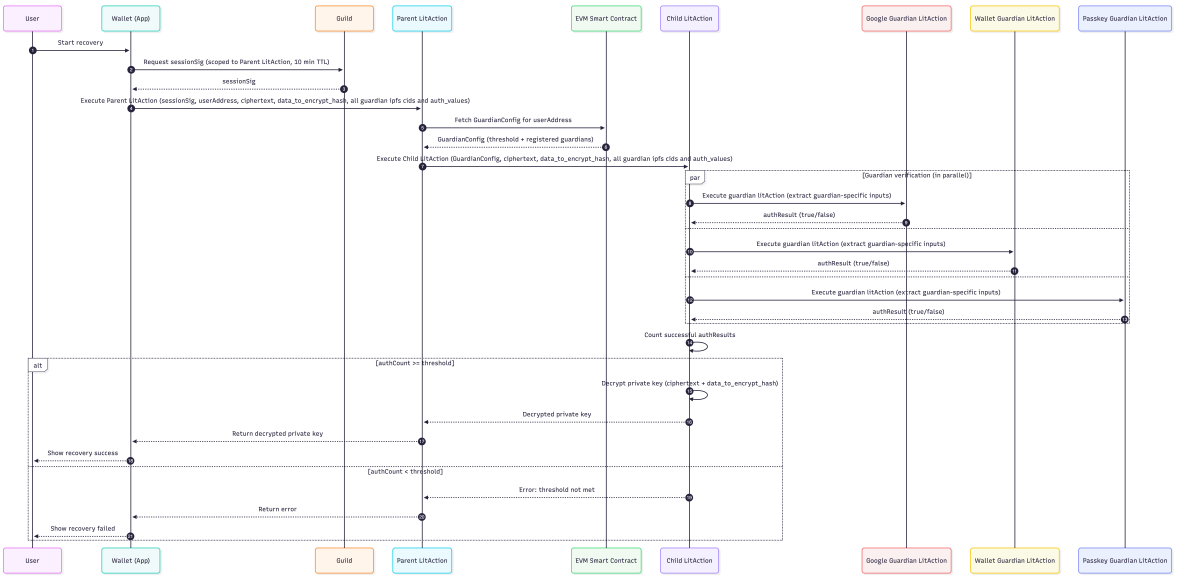
Guardian Configuration Retrieval

Once the user's address has been identified, the system uses it to fetch the corresponding `GuardianConfig` from the smart contract. This configuration defines the required threshold, the enumerable list of registered guardians (`guardianCIDs`), and the authentication mapping used for verification.

The retrieved configuration is then shown to the user on the guardian screen as **"X of Y guardians required"**, where:

- **X** is the minimum number of guardians that must successfully authenticate (the threshold), and
- **Y** is the total number of guardians registered for the user and is derived from the length of the on-chain `guardianCIDs` array.

As each guardian successfully authenticates, the progress toward the threshold is updated. Finally when the number of successfully authenticated guardians reaches the threshold, the recovery process is considered ready, and the user will be prompted to proceed with decryption and allowing to execute the parent `litAction`.



LitAction Execution

To execute any Lit Action, a session signature (`sessionSig`) is required. The session signature acts like a short-lived access token and is included in the request when invoking the parent `litAction`.

The session signature is generated by Guild with a single use and 10-minute expiration. It is scoped so that it can only be used to execute the parent `litAction`. The user's wallet hits guild's provided `sessionSig` endpoint to generate and return a `sessionSig`.

Using the session signature, the wallet invokes the parent `litAction`, providing the following inputs:

- `address` - the user's address, which is used to fetch the user's configuration from the smart contract
- `auth_values`: A map of the `ipfs_cid` to the respective `auth_value` of all the authenticated guardians. (eg. `Qc....` → `google_id_12324`)

After receiving the request, the parent `litAction` retrieves the user's guardian configuration from the **Polygon GuardianConfig contract** using the provided address, and retrieves the child `litAction` `ipfs_cid` from the **Ethereum LitAction Registry contract**. Once the configuration is loaded, the parent `litAction` invokes the child `litAction`.

The child `litAction` then invokes each guardian's `litAction` from the `auth_values` variable passed from the parent `litAction`, passing all the relevant parameters. Each guardian `litAction` is responsible for extracting the information it needs from

the shared input, as the input contains data for all guardians rather than being guardian-specific.

Each guardian's `litAction` independently verifies the user by comparing the `auth_value` passed from the user specified in the `auth_values` object from the parent `litAction`, and the matching `auth_value` stored in the smart contract by hashing the `ipfs_cid` of the guardian and retrieving it from the `guardianEntries`. Depending on whether the two `auth_value`s match it will return a truthy or falsy response.

The child `litAction` collects the responses from all guardians and checks whether the number of successful (truthy) authentications meets or exceeds the required threshold.

If the threshold is met, the user is considered successfully authenticated and the child `litAction` will return a truthy response to the parent `litAction`.

The parent `litAction` will then, upon receiving the truthy response from the child `litAction`, propagate the response back to lit's `decrypt` function, effectively authorizing the decryption of the user's private key.

Guardian List

When the user is redirected to this screen, it indicates that their private key has already been backed up and that **at least one guardian is registered** for the address.

The **Guardian List** screen is primarily used to **manage existing guardians**, allowing the user to add or remove guardians associated with the backed-up private key.

At this stage, the `existing` variable is set to true. This prevents:

- invoking Lit's `encrypt` function, and
- creating a new record in the `ciphertexts` table,

since a valid `ciphertext` already exists for the user's address.

Adding a Guardian

Adding a guardian to an existing backup closely mirrors the flow for creating a new backup, with the key difference that **no new encryption is performed**.

Because a `ciphertext` already exists:

- the private key is **not re-encrypted**, and
- no new entry is created in the `ciphertexts` table.

After the user selects a guardian and successfully completes its authentication flow:

1. The guardian is linked to the existing `ciphertext` by inserting a new entry into the `guardian_to_ciphertext` table.
2. A corresponding entry is added to the user's on-chain `GuardianConfig`'s `guardianEntries` mapping in the smart contract.
3. The recovery threshold is recalculated and updated on-chain, based on the new total number of guardians.

In practice, since adding a guardian implies that more than one guardian exists, the threshold being 1 is unlikely outside of the initial setup.

If the guardian has already been registered, an error - "Guardian already registered" will be presented to the user

Once these steps complete successfully, the newly added guardian appears in the Guardian List UI.

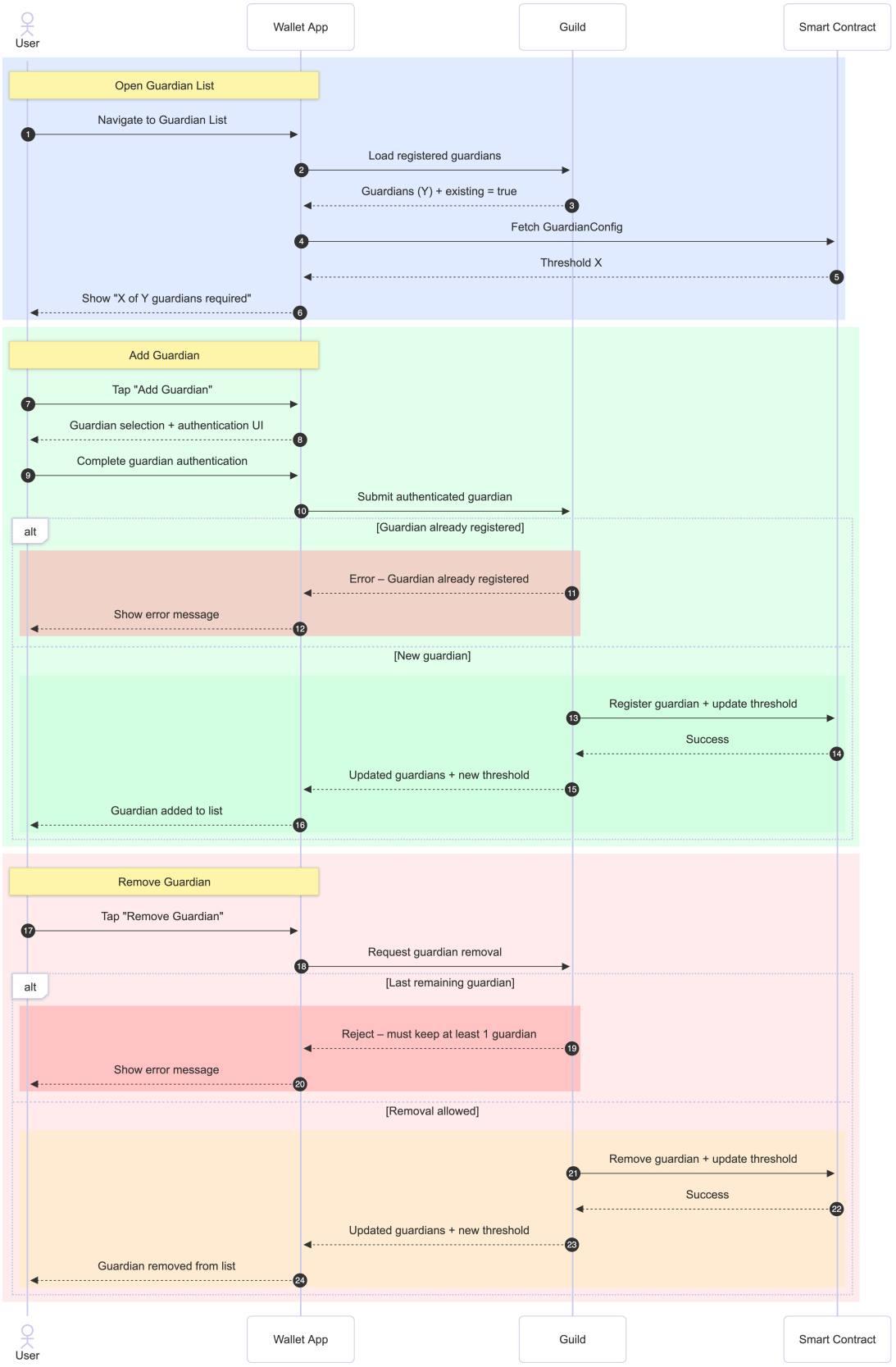
Removing a Guardian

Removing a guardian reverses the addition process:

1. The association between the guardian and the `ciphertext` is removed from the `guardian_to_ciphertext` table.
2. The corresponding guardian hash is removed from the user's `guardianCIDs` array, and the matching entry is deleted from the `guardianEntries` mapping in the smart contract.
3. The recovery threshold is recalculated and updated to reflect the reduced guardian count.

Removing the last remaining guardian constitutes an explicit opt-out from the guardian recovery system. Upon opt-out, all recovery-related state is cleared, including deletion of the `ciphertext` from the database and removal of the user's `GuardianConfig` from the smart contract. After opting out, the user may continue using Payy's traditional recovery mechanism (e.g. cloud-based backup or manual seed phrase storage).

Alternatively, if enabled, the system may perform reverse lookup on-chain by hashing the authenticated `auth_value` and querying `authValueToAddress` via its getter to retrieve the associated wallet address.


User
Wallet App
Guild
Smart Contract
Open Guardian List
1 Navigate to Guardian List
2 Load registered guardians
3 Guardians (Y) + existing = true
4 Fetch GuardianConfig
5 Threshold X
6 Show "X of Y guardians required"
Add Guardian
7 Tap "Add Guardian"
8 Guardian selection + authentication UI
9 Complete guardian authentication
10 Submit authenticated guardian
alt
[Guardian already registered]
11 Error – Guardian already registered
12 Show error message
[New guardian]
13 Register guardian + update threshold
14 Success
15 Updated guardians + new threshold
16 Guardian added to list
Remove Guardian
17 Tap "Remove Guardian"
18 Request guardian removal
alt
[Last remaining guardian]
19 Reject – must keep at least 1 guardian
20 Show error message
[Removal allowed]
21 Remove guardian + update threshold
22 Success
23 Updated guardians + new threshold
24 Guardian removed from list
User
Wallet App
Guild
Smart Contract