Ledger Recover

Technical White Paper

This page is intentionally left blank

Table of contents

Introduction

Welcome to this technical white paper, a deep dive into the system design, architectural features, and operational flows of Ledger Recover. As an advanced solution for the backup and restoration of your device seed, Ledger Recover encapsulates a series of robust cryptographic protocols, purpose-built to deliver an unparalleled combination of security, resiliency, and user-centricity.

In this document, we'll be revealing the foundational design and key security targets of the underlying cryptographic protocol employed by Ledger Recover. This will involve insights into the system's structural layout, as well as a broader view of its primary design and security objectives. Our aim is to provide you with a clear understanding of the intricate mechanisms behind Ledger Recover, showcasing the unique measures incorporated to protect and uphold the integrity of your data.

We'll also take a close look at the three main operational flows of Ledger Recover – backing up your seed, restoring it on a new device, and securely deleting your backups. These procedures form the backbone of Ledger Recover's functionality, each one thoughtfully designed to ensure an efficient, user-friendly experience without compromising on the system's high level of security.

By the end of this white paper, we anticipate providing you with an enriched understanding of Ledger Recover's system design, cryptographic protocol, and its three primary functionalities. Whether you're a seasoned professional in the field or a newcomer, we trust that the insights provided will help you better understand Ledger Recover and highlight its value in secure data backup and restoration in the ever-evolving cryptocurrency world.

System Design Overview

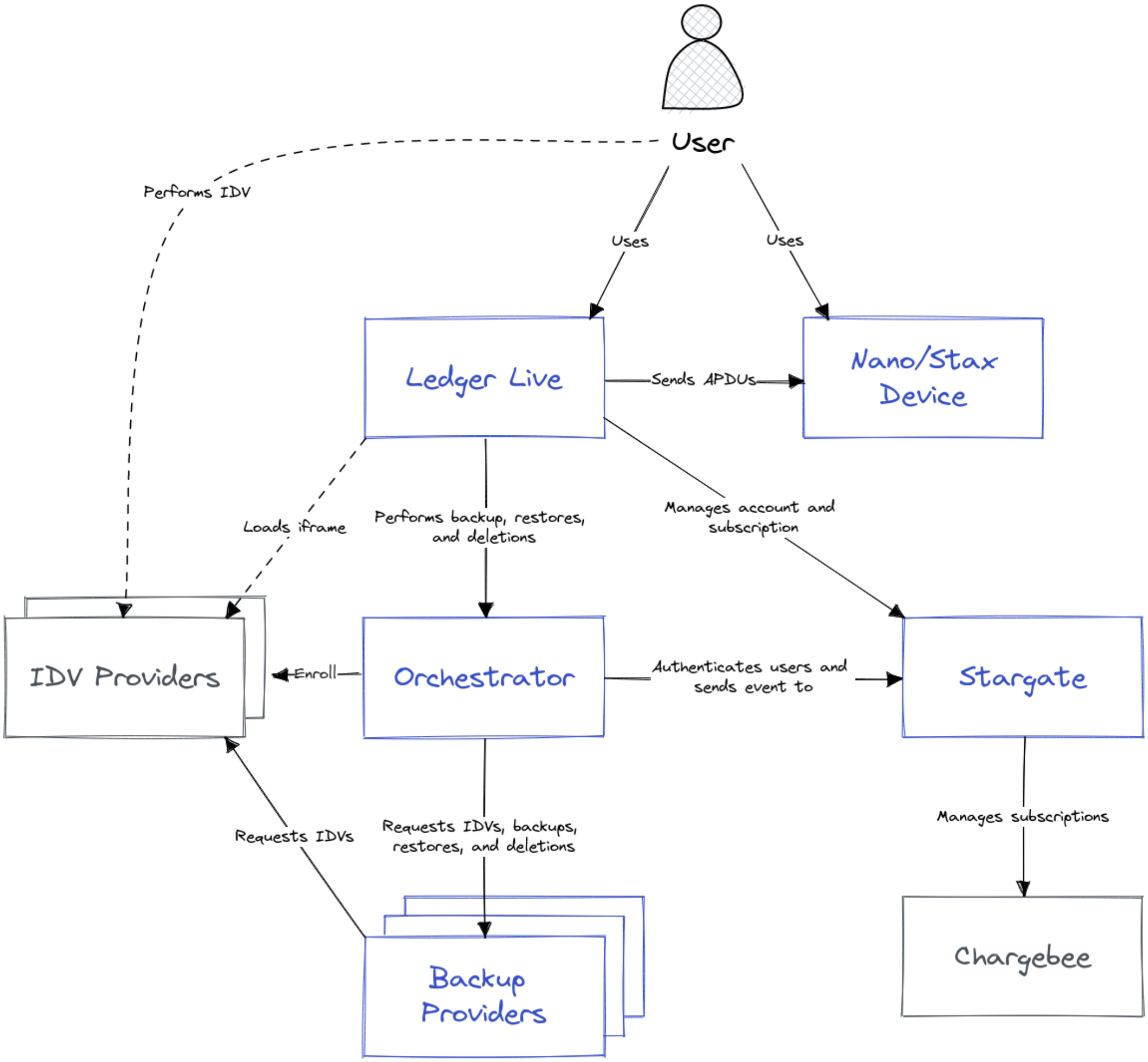
The overall design of Ledger Recover is quickly presented in the following diagram.

The two most important components are on one hand, the devices, hardware wallets containing a *Secure Element* and a trusted display allowing safe manipulation of the seed; and on the other hand the backup providers, especially their *Hardware Security Module*, securing all the cryptographic exchanges introduced in this paper.

Secure Elements (SEs) and Hardware Security Modules (HSMs) are integral components in the field of information security that provide secure storage and operations for sensitive data. A SE is a tamper-resistant platform that can host applications and their confidential and cryptographic data securely. It's commonly found in small form-factor devices such as smart cards, mobile phones, and IoT devices. On the other hand, an HSM is a physical device that safeguards and manages digital keys for strong authentication and provides crypto processing. It prevents unauthorized access to sensitive material and is often used in a server-type environment for critical operations such as managing SSL/TLS digital keys for website security, transaction processing, identity management, and database encryption. Both SEs and HSMs deliver enhanced security by ensuring that sensitive data is stored and processed within a secure and trusted environment.

Other notable components in the system are:

- The *Orchestrator* makes sure all messages transit correctly and in the right order between all other components: it is not critical for the security of the system but important for getting the best UX possible
- *Ledger Live* is the main user interface for interacting with Recover, on both mobile and desktop
- *IDV Providers* are our partners providing the Identity Verification service required by the current design for strongly authenticating users when trying to backup and restore
- *Stargate and Chargebee* are our account and subscription services



Ledger Recover System Design

Design Goals

The protocol is intended to be secured end-to-end between the devices and the HSMs of the backup providers, so no intermediate systems or actors - Ledger Live, orchestrator or an attacker - can read sensitive information shared between the backup providers HSMs and the devices.

The protocol is designed to support the following set of properties.

Self-Custody

- The seed is split into shares using a variant of Shamir Secret Sharing so that no backup provider touches, controls or sees the full seed.
- The seed is additionally encrypted before being split by a key common to all devices, to limit collusion risks from backup providers.

Resilience

- Splitting uses a 2-out-of-3 scheme so that seed can still be restored even if a single backup provider loses a share.
- Strong cryptographic commitments on share generation allows providing safe backup guarantees without leaking data on shares.

Share transport security

- End-to-end encryption to a known recipient with strong forward secrecy.
- Shares are encrypted based on an elliptic curve Diffie-Hellman key exchange - ephemeral with an authenticated party.
- Assuming the ephemeral private keys are secure, and the recipient is not being actively impersonated by an attacker that has stolen its static private key, the share cannot be decrypted by an intermediary.

Share storage security

- End recipient of a share is a Hardware Security Module, setup by each backup provider.

- All shares and all Personally Identifiable Info in the protocol stored by backup providers are encrypted by an encryption key that never leaves the backup provider HSM.
- Identifiers of backups are diversified to avoid easy correlation between databases, with a diversification key that never leaves the backup provider HSM.
- All cryptographic operations are done within the confines of the HSM and are atomic.

Session bindings

- To avoid session swapping by an attacker, all sessions between device, orchestrator, providers and IDVs are bound together at the crypto protocol level.
- Identity data is bound by devices to backup and restore sessions.
- Restore Sessions are hard locked to a device.
- IDV session is bound to restore session through use of a One Time Security Code, derived from the shared key between device and backup provider.

Non-repudiation

- The backup provider proves that it was able to decrypt the seed-share.
- Orchestrator is able to cryptographically verify that each share has been transmitted securely, without leaking info on the shares.

Privacy preserving

- *Orchestrator:* Static public key is transmitted in clear.
- *Backup provider:* Static public key is encrypted with forward secrecy to an authenticated party. Spying exchanges won't leak the provider identity.
- *Device:* Static public key is encrypted with forward secrecy to an authenticated party. Spying exchanges won't leak the device identity.

Small number of exchanges

- Only three messages (backup) or four messages (restore) need to be exchanged between the device and a single provider.

- Can be reduced by one message if the proof of reception of the seed-share is not required.

Ledger independent

- An (expert) user can replace Ledger as a trusted root and run the protocol to generate encrypted shares completely independently from Ledger.

Notation

Notation	Description
$(\cdot)^e$	Ephemeral key
k_{name}	Symmetric key or secret for name
d_{name}	Asymmetric private key for name
P_{name}	Asymmetric public key for name
$AsymKeyGen()$	Key-pair generation function
$VSS(K, i, n)$	Pedersen's Verifiable Secret-Sharing of secret K (i-th share, n recovery threshold)
$Random(n)$	Generate n random bytes
$M1 \mid\mid M2$	Concatenation of messages $M1$ and $M2$
$HMAC(K, M)$	HMAC of message M using key K
$Sign(d, M)$	Signature of message M using private key d
$ECDH(d, P)$	Elliptic curve Diffie-Hellman key agreement between private key d and public key P
$BIP32(seed, path)$	Derive $seed$ using BIP32 $path$
$\{\cdot\}_K$	Encryption using key K
$\{\cdot\}_K^{-1}$	Decryption using key K
S_i	Share of index i
T_i^s	Commitment of share i
T^c	Commitment of coefficients

Keys

Notation	Description	Location
d_L	Ledger endorsement private key	Ledger Factory HSM
P_L	Ledger endorsement public key	n/a (public)
d_U	(Optional) User endorsement private key	n/a (user controlled)
P_U	(Optional) User endorsement public key	n/a (user controlled)
d_D	Device static private key	Device
P_D	Device static public key	n/a (public)
d_{S_i}	Seed ID private key for backup provider i	Device
P_{S_i}	Seed ID public key for backup provider i	n/a (public)
c_i	Challenge sent by backup provider i	n/a (random)
r_i	Random byte string sent by backup provider i	n/a (random)
C_D^l	Ledger-generated Device static certificate	n/a (public)
C_D^u	User-generated Device static certificate	n/a (public)
d_{B_i}	Backup provider i static private key	Backup provider HSM
P_{B_i}	Backup provider i static public key	n/a (public)
$C_{B_i}^l$	Backup provider i static certificate, signed by Ledger	n/a (public)
$C_{B_i}^u$	Backup provider i static certificate, signed by the User	n/a (public)
d_O	Orchestrator static private key	Orchestrator

P_O	Orchestrator static public key	n/a (public)
C_O^I	Orchestrator static certificate, signed by Ledger	n/a (public)
C_O^u	Orchestrator static certificate, signed by the User	n/a (public)
k_{seed}	Devices seed encryption key (common to all devices)	Device (common)
$k_{hkdf_{B_i}}$	Derivation keys for diversifying the Backup IDs	Backup provider HSM
$k_{hsm_{B_i}}$	Backup provider HSM storage encryption key	Backup provider HSM

Algorithms

Operation	Algorithm
Signature	NIST FIPS 186-5 Digital Signature Standard (DSS) ECDSA over secp256k1
Key agreement	NIST Special Publication (SP) 800-56A Rev. 3, Recommendation for Pair-Wise Key-Establishment Schemes Using Discrete Logarithm Cryptography ECDH over secp256k1
Hash	NIST FIPS 180-4 Secure Hash Standard (SHS) SHA-256
Symmetric Encryption	AEAD-AES-SIV-CMAC-256
Seed Encryption (k_{seed})	AES-256-CBC
HMAC	Federal Information Processing Standard (FIPS) 198-1, The Keyed-Hash Message Authentication Code (HMAC)
Key Generation	NIST Special Publication (SP) 800-56A Rev. 3, Recommendation for Pair-Wise Key-Establishment Schemes Using Discrete Logarithm Cryptography
Pedersen's Verifiable Secret-Sharing	https://link.springer.com/chapter/10.1007/3-540-46766-1_9 See section below
BIP32	bips/bip-0032.mediawiki at master · bitcoin/bips
Derive_OTSC	See section below

Seed generation

Ledger devices are generating the seed and the seed recovery phrase using BIP 39. A summary is provided below for reference.

An initial entropy is generated from the device True Random Number Generator. The entropy is either 128 bits for a 12-word seed recovery phrase or 256 bits for a 24-word one.

A checksum is then added to the entropy, by taking the first 32 bits of the SHA-256 hash of the entropy. The checksum is appended to the entropy, and the result is split into groups of 11 bits, which serves as an index into an array of 2048 words (part of the BIP39 specification). The list of words is collectively the seed recovery phrase, or the “mnemonic”.

To create a binary seed from the mnemonic, we use the PBKDF2 function with a mnemonic sentence (in UTF-8 NFKD) used as the password and the string "mnemonic" + passphrase (again in UTF-8 NFKD) used as the salt. The iteration count is set to 2048 and HMAC-SHA512 is used as the pseudo-random function. The length of the derived key is 512 bits (= 64 bytes).

This seed can be later used to generate deterministic wallets using BIP-0032 or similar methods.

In Ledger Recover, what gets encrypted then split using Pedersen Shamir Secret Sharing is the initial entropy itself. When restoring the seed, the BIP 39 algorithm will be applied to the recombined entropy to restore the seed of the user.

Pedersen Verifiable Secret Sharing

Instead of the basic Shamir Secret Sharing scheme, we picked a “verifiable” variant that would allow us to perform consistency checks in a natural, built-in way. We would like to give the ability to backup providers and the orchestrator to verify that they indeed possess shares belonging to the same given seed, without revealing those shares.

A similar check is desirable when a fresh device performs the *Restore* operation to retrieve the seed from our backups, so we can avoid restoring a different seed than expected.

Verifiable Secret Sharing (VSS) schemes come into play to help us solve this, and in our case, it is Pedersen’s “Non-Interactive and Information-Theoretic Secure Verifiable Secret Sharing” from CRYPTO ‘91.

This scheme adds *commitments* to the shares produced by the Shamir secret-sharing procedure.

$$Commit(s, t) = s.P + t.Q$$

With *P* and *Q* defined over some appropriate group (of points of the elliptic curve) *G*.

In our implementation, this group is the curve *secp384r1*. Since seeds can span 256 bits, using a curve like *secp256k1* whose order is less than 2^{256} would leave the possibility of collisions or incorrect seed backup, albeit very unlikely. Computing a commitment in this case naturally amounts to scalar multiplications on *sepc384r1*, but leaves an open question: how do we pick *P* and *Q* ?

The first one is easy, we can use the base point of the curve defined by the NIST standard. However in Pedersen's original scheme, *Q* must be chosen so that its discrete logarithm in base *g* is unknown to the device performing the sharing. And for good reason: the main property of the scheme is that the committer (i.e. the device sharing its seed) cannot lie to the other participants and give them an incorrect share.

In order to pick this *Q* we deterministically generated a point on the used curve using the most basic hash-to-curve algorithm there is, a method called hash-then-increment. This method ensures that the discrete logarithm mentioned above is not known to anyone.

Python

```python
import hashlib
from ecpy.curves import Curve, Point

def hash_to_curve(s: str):
    curve = Curve.get_curve('secp384r1')
    x = int.from_bytes(hashlib.sha384(s).digest(), 'big')
    y_squared = (x*x*x + curve.a * x + curve.b) % curve.field
    y = pow(y_squared, (curve.field+1)//4, curve.field)
    p = Point(x, y, curve)
    assert p.is_on_curve
    print(p)

hash_to_curve(b"Ledger Recover Auxiliary VSS Point seed")
```

Using the above code with the starting string being "Ledger Recover Auxiliary VSS Point seed" we obtain, without even requiring the 'then-increment' step of the algorithm, the point:

```
Q                                                                             =
(0xfbfc7b98c353a92621854e13d1efbcf1385601a5ce14f5b64eb722ddc52b5690f302663f
904679847fc5114dcc7d94ed                                                      ,
0x3b59569d2791c3a8e1bc8c32b5c0e537d1a4a95f0c3b16c493e5d69fec29ead4c95f7ed6b
9c561a856a7b83d980f1cd8)
```

As an example of a verifiable sharing procedure of a seed in a 2-out-of-3 scheme with three participants that will act as backup providers (this part assumes familiarity with the mathematics behind Shamir Secret Sharing) :

We define *N* to be *secp384r1*'s order.

$$t = Random() \mod N$$
$$E_0 = Commit(seed, t)$$

The above value serves as a form of "reference commitment".

Define a random polynomial that shares the seed as:

$$F_1 = Random() \mod N$$
$$F(x) = seed + F_1 x$$

Set the shares to be sent as:

$$s_1 = F(1), s_2 = F(2), s_3 = F(3)$$

Define a random polynomial that shares *t* as:

$$G_1 = Random() \mod N$$
$$G(x) = t + G_1 x$$

Set the second part of the shares to be sent as:

$$t_1 = G(1), t_2 = G(2), t_3 = G(3)$$

Set:

$$E_1 = Commit(F_1, G_1)$$

Broadcast E_0 and E_1 to all backup providers.
Send $S_1 = (s_1, t_1)$ privately to the first backup provider.
Send $S_2 = (s_2, t_2)$ privately to the second backup provider.
Send $S_3 = (s_3, t_3)$ privately to the third backup provider.

Now when a backup provider receives such a share $S_i = (s_i, t_i)$, it can check the following relation:

$$Commit(s_i, t_i) == E_0 + i.E_1$$

Proof:

$$\begin{aligned} s_i.P + t_i.Q &== seed.P + t.Q + i(F_1.P + G_1.Q) \\ &== (seed + F_1.i)P + (t + G_1.i)Q \\ &== F(i).P + G(i).Q \end{aligned}$$

If this relation does not hold, then the share received is incorrect and has no relation to the expected seed. This is done without revealing any share other than the one received by the backup provider doing this verification.

The same goes when restoring the seed to a new device: the backup providers will (secretly) send the shares back to the device, along with their computed commitments and the broadcasted ones. That way, the device can also check that it is not restoring some arbitrary seed, as long as the backup providers do not collude in an attempt to cheat.

The scheme gives us powerful verifiability properties: the device cannot cheat - the backup providers can check the share they receive is generated correctly - and backup providers cannot send back a modified share without colluding.

In the following sequence diagrams, we are using the following notations:

$$VSS(seed, i) = (s_i, t_i) == S_i$$
$$VSSCommitCoeffs() = (E_0, E_1) == T_c$$
$$VSSCommitShare(S_i) = Commit(s_i, t_i) == T_i^s$$
$$\text{Verify commitments: check } Commit(s_i, t_i) == E_0 + i.E_1$$

One Time Security Code derivation

The One Time Security Code is a string computed by taking, as digits characters, the first 4 bytes modulo 10 of the Hash of a concatenation of a known prefix, the MAC and ENC keys generated by the ECDH exchange.

$$otsc_{bytes} = Hash(\texttt{0x06} \parallel k_{MAC} \parallel k_{ENC})$$

$$otsc = \{x_i \texttt{ as digit, } x_i = otsc_{bytes}[i] \mod 10; i \in [0..3]\}$$

This computation uses Hash and ECDH keys as specified in the algorithm table.

Initialization

The initialization stage is used to build trust between the various secure components in the system.

Ledger as the root of trust

Ledger signs a certificate for each backup provider HSM B_i:

$$C^l_{B_i} = Name_{B_i} \parallel P_{B_i} \parallel Sign(d_L, ROLE_{BP} \parallel Name_{B_i} \parallel P_{B_i})$$

Ledger also signs a certificate for the orchestrator HSM (role authorized to trigger a backup/restore):

$$C^l_O = P_O \parallel Sign(d_L, ROLE_O \parallel P_O)$$

Devices are issued with a certificate generated in production (already the case):

$$C^l_D = P_D \parallel Sign(d_L, ROLE_D \parallel P_D)$$

User as the root of trust

Alternatively, the user can replace Ledger as the root of trust by generating his own root CA key pair:

$$(d_U, P_U) = KeyGen()$$

and enrolling it in the operating system.

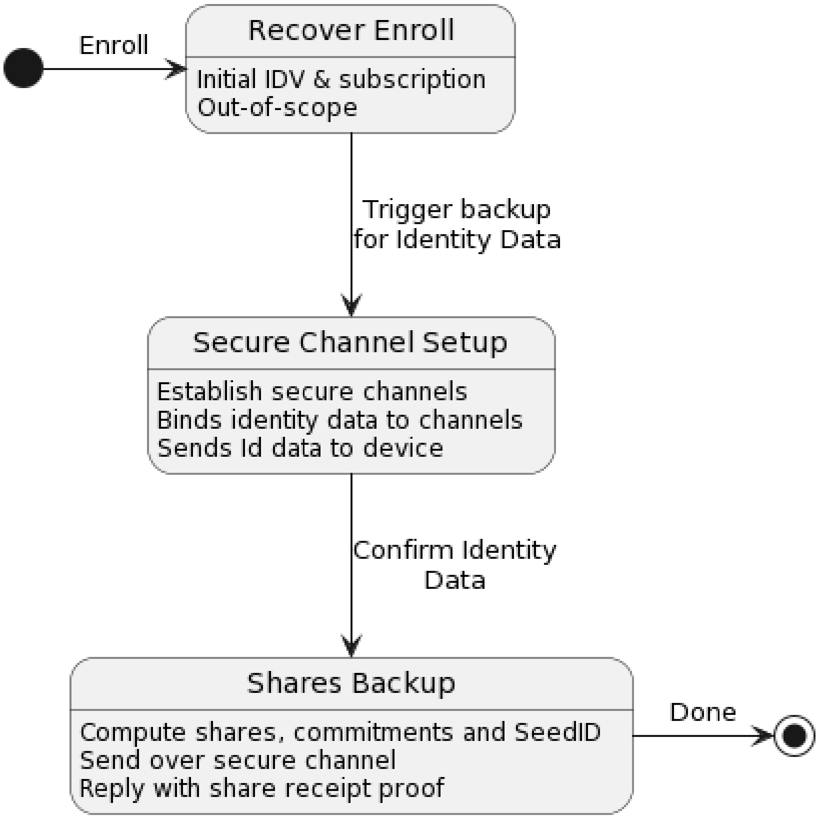
Then the user can proceed to generate certificates for their backup infrastructure:

$$C^u_{B_i} = Name_{B_i} \parallel P_{B_i} \parallel Sign(d_U, ROLE_{BP} \parallel Name_{B_i} \parallel P_{B_i})$$
$$C^u_O = P_O \parallel Sign(d_U, ROLE_O \parallel P_O)$$

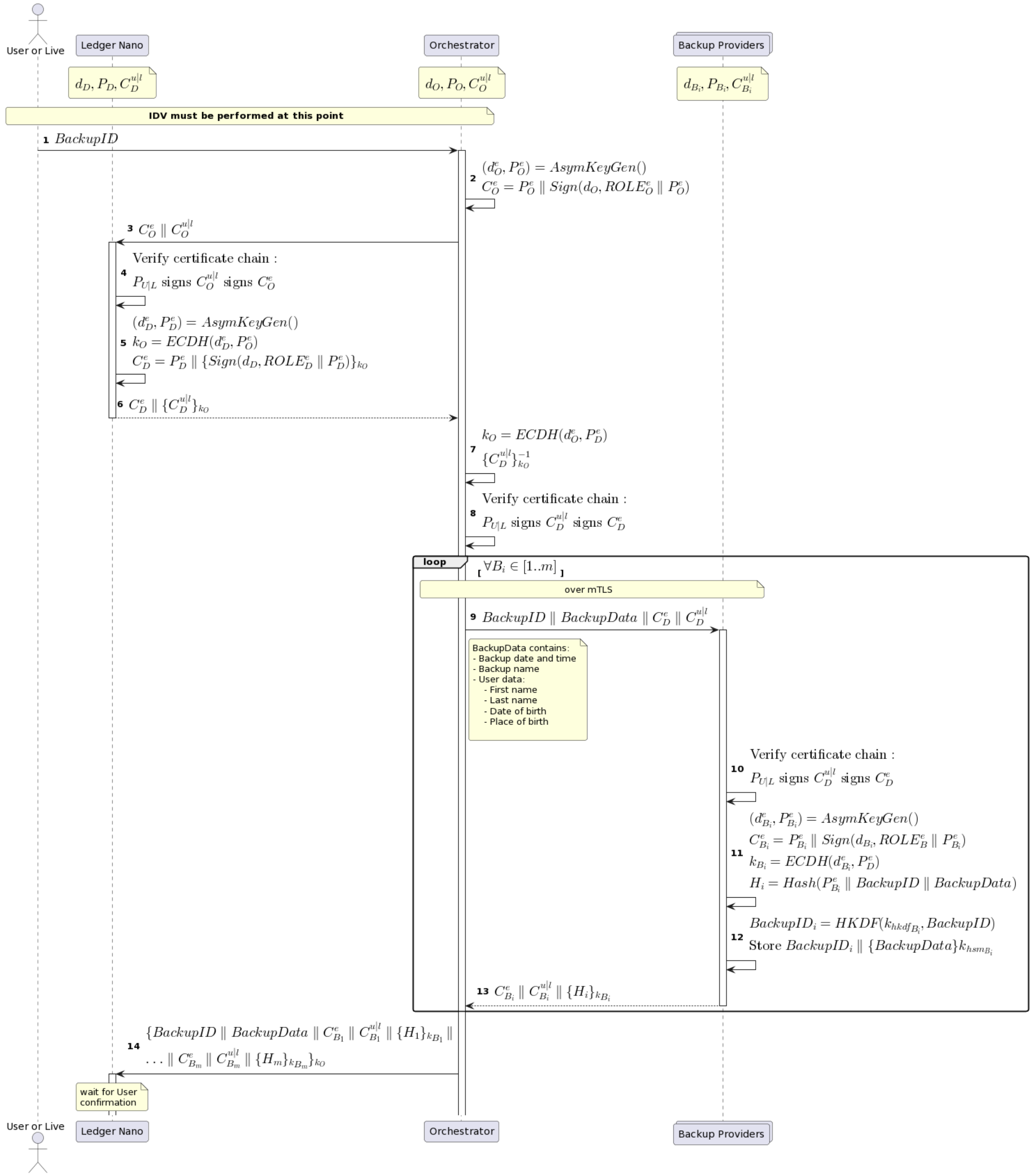
Backup

The backup flow is divided in three main stages:

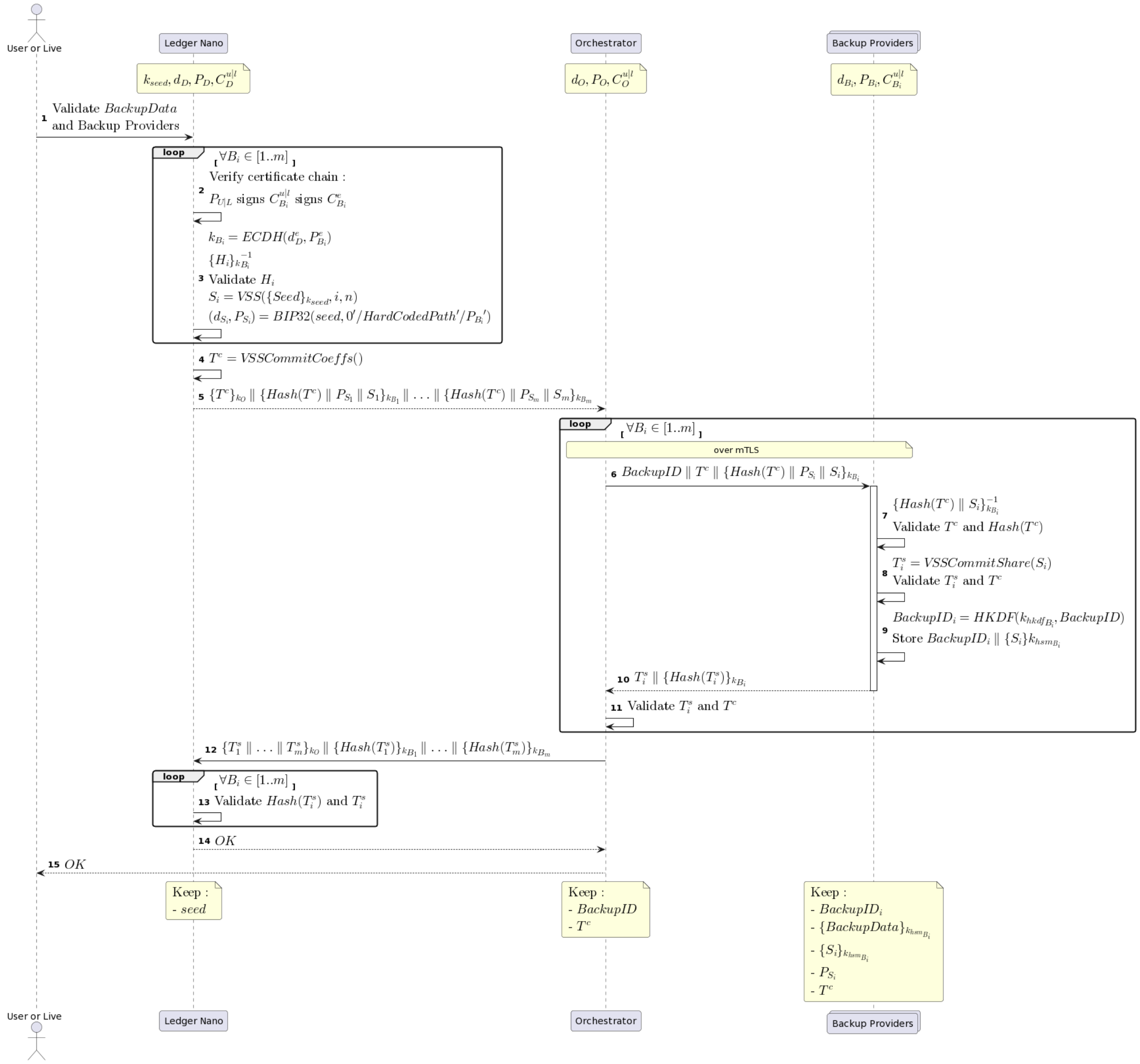
- Enroll stage: the user will enroll to recover, subscribe and perform a first IDV, this stage is out of scope of the current white paper, it is assumed the orchestrator has a trusted identity out of it.
- Secure Channel Setup: the secure channels are established between device and backup providers (and orchestrator), bound together and to the identity data.
- Shares backup: once the user approves the identity on device, shares are computed, commitments exchanged in both directions (proving shares are generated, consistent along the way and stored correctly) and ids computed for the delete phase.


Enroll
Recover Enroll
Initial IDV & subscription
Out-of-scope
Trigger backup for Identity Data
Secure Channel Setup
Establish secure channels
Binds identity data to channels
Sends Id data to device
Confirm Identity Data
Shares Backup
Compute shares, commitments and SeedID
Send over secure channel
Reply with share receipt proof
Done

Secure Channels Setup

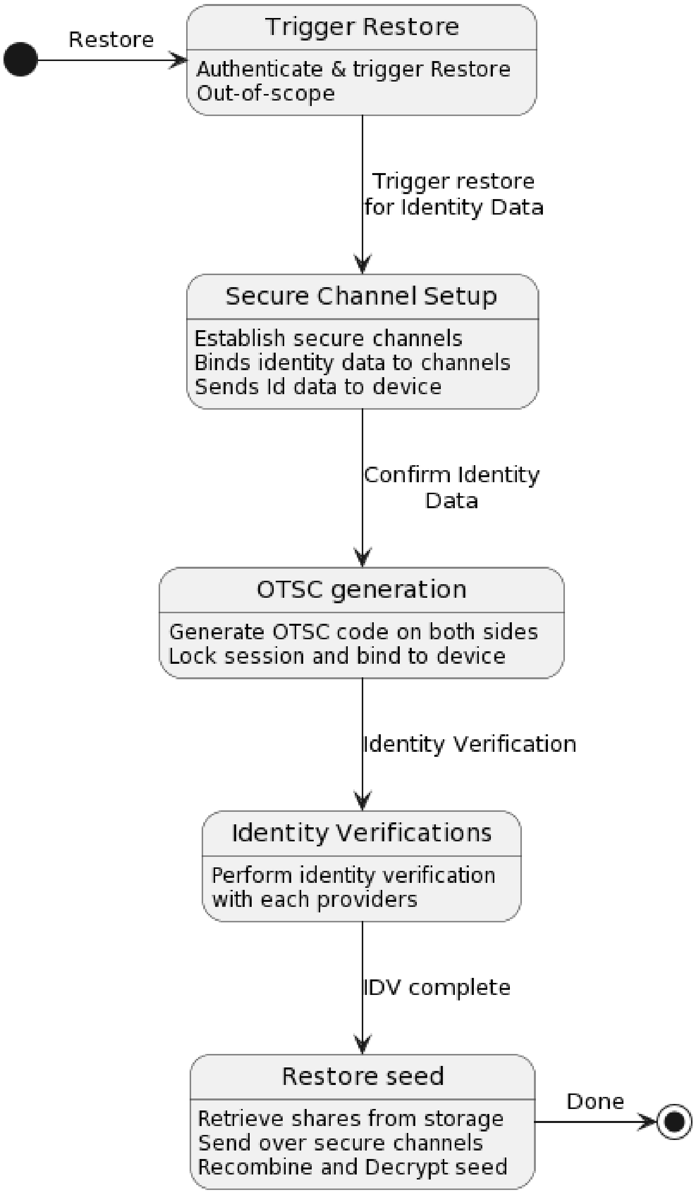

User or Live
Ledger Nano
Orchestrator
Backup Providers
$d_D, P_D, C_D^{u|l}$
$d_O, P_O, C_O^{u|l}$
$d_{B_i}, P_{B_i}, C_{B_i}^{u|l}$
IDV must be performed at this point
1 $BackupID$
2 $(d_O^e, P_O^e) = AsymKeyGen()$
$C_O^e = P_O^e \parallel Sign(d_O, ROLE_O^e \parallel P_O^e)$
3 $C_O^e \parallel C_O^{u|l}$
4 Verify certificate chain :
$P_{U|L}$ signs $C_O^{u|l}$ signs C_O^e
5 $(d_D^e, P_D^e) = AsymKeyGen()$
$k_O = ECDH(d_D^e, P_O^e)$
$C_D^e = P_D^e \parallel \{Sign(d_D, ROLE_D^e \parallel P_D^e)\}_{k_O}$
6 $C_D^e \parallel \{C_D^{u|l}\}_{k_O}$
7 $k_O = ECDH(d_O^e, P_D^e)$
$\{C_D^{u|l}\}_{k_O}^{-1}$
8 Verify certificate chain :
$P_{U|L}$ signs $C_D^{u|l}$ signs C_D^e
loop
[$\forall B_i \in [1..m]$]
over mTLS
9 $BackupID \parallel BackupData \parallel C_D^e \parallel C_D^{u|l}$
BackupData contains:
- Backup date and time
- Backup name
- User data:
- First name
- Last name
- Date of birth
- Place of birth
10 Verify certificate chain :
$P_{U|L}$ signs $C_D^{u|l}$ signs C_D^e
11 $(d_{B_i}^e, P_{B_i}^e) = AsymKeyGen()$
$C_{B_i}^e = P_{B_i}^e \parallel Sign(d_{B_i}, ROLE_B^e \parallel P_{B_i}^e)$
$k_{B_i} = ECDH(d_{B_i}^e, P_D^e)$
$H_i = Hash(P_{B_i}^e \parallel BackupID \parallel BackupData)$
12 $BackupID_i = HKDF(k_{hkdf_{B_i}}, BackupID)$
Store $BackupID_i \parallel \{BackupData\}k_{hsm_{B_i}}$
13 $C_{B_i}^e \parallel C_{B_i}^{u|l} \parallel \{H_i\}_{k_{B_i}}$
14 $\{BackupID \parallel BackupData \parallel C_{B_1}^e \parallel C_{B_1}^{u|l} \parallel \{H_1\}_{k_{B_1}} \parallel$
$\ldots \parallel C_{B_m}^e \parallel C_{B_m}^{u|l} \parallel \{H_m\}_{k_{B_m}}\}_{k_O}$
wait for User confirmation
User or Live
Ledger Nano
Orchestrator
Backup Providers

Shares Backup


User or Live
Ledger Nano
Orchestrator
Backup Providers
$k_{seed}, d_D, P_D, C_D^{u|l}$
$d_O, P_O, C_O^{u|l}$
$d_{B_i}, P_{B_i}, C_{B_i}^{u|l}$
1 Validate $BackupData$ and Backup Providers
loop
$[\forall B_i \in [1..m]]$
2 Verify certificate chain : $P_{U|L}$ signs $C_{B_i}^{u|l}$ signs $C_{B_i}^e$
3 $k_{B_i} = ECDH(d_D^e, P_{B_i}^e)$ $\{H_i\}_{k_{B_i}}^{-1}$ Validate H_i $S_i = VSS(\{Seed\}_{k_{seed}}, i, n)$ $(d_{S_i}, P_{S_i}) = BIP32(seed, 0'/HardCodedPath'/P_{B_i}')$
4 $T^c = VSSCommitCoeffs()$
5 $\{T^c\}_{k_O} \parallel \{Hash(T^c) \parallel P_{S_1} \parallel S_1\}_{k_{B_1}} \parallel \ldots \parallel \{Hash(T^c) \parallel P_{S_m} \parallel S_m\}_{k_{B_m}}$
loop
$[\forall B_i \in [1..m]]$
over mTLS
6 $BackupID \parallel T^c \parallel \{Hash(T^c) \parallel P_{S_i} \parallel S_i\}_{k_{B_i}}$
7 $\{Hash(T^c) \parallel S_i\}_{k_{B_i}}^{-1}$ Validate T^c and $Hash(T^c)$
8 $T_i^s = VSSCommitShare(S_i)$ Validate T_i^s and T^c
9 $BackupID_i = HKDF(k_{hkdf_{B_i}}, BackupID)$ Store $BackupID_i \parallel \{S_i\}k_{hsm_{B_i}}$
10 $T_i^s \parallel \{Hash(T_i^s)\}_{k_{B_i}}$
11 Validate T_i^s and T^c
12 $\{T_1^s \parallel \ldots \parallel T_m^s\}_{k_O} \parallel \{Hash(T_1^s)\}_{k_{B_1}} \parallel \ldots \parallel \{Hash(T_m^s)\}_{k_{B_m}}$
loop
$[\forall B_i \in [1..m]]$
13 Validate $Hash(T_i^s)$ and T_i^s
14 OK
15 OK
Keep : - $seed$
Keep : - $BackupID$ - T^c
Keep : - $BackupID_i$ - $\{BackupData\}_{k_{hsm_{B_i}}}$ - $\{S_i\}_{k_{hsm_{B_i}}}$ - P_{S_i} - T^c
User or Live
Ledger Nano
Orchestrator
Backup Providers

Restore

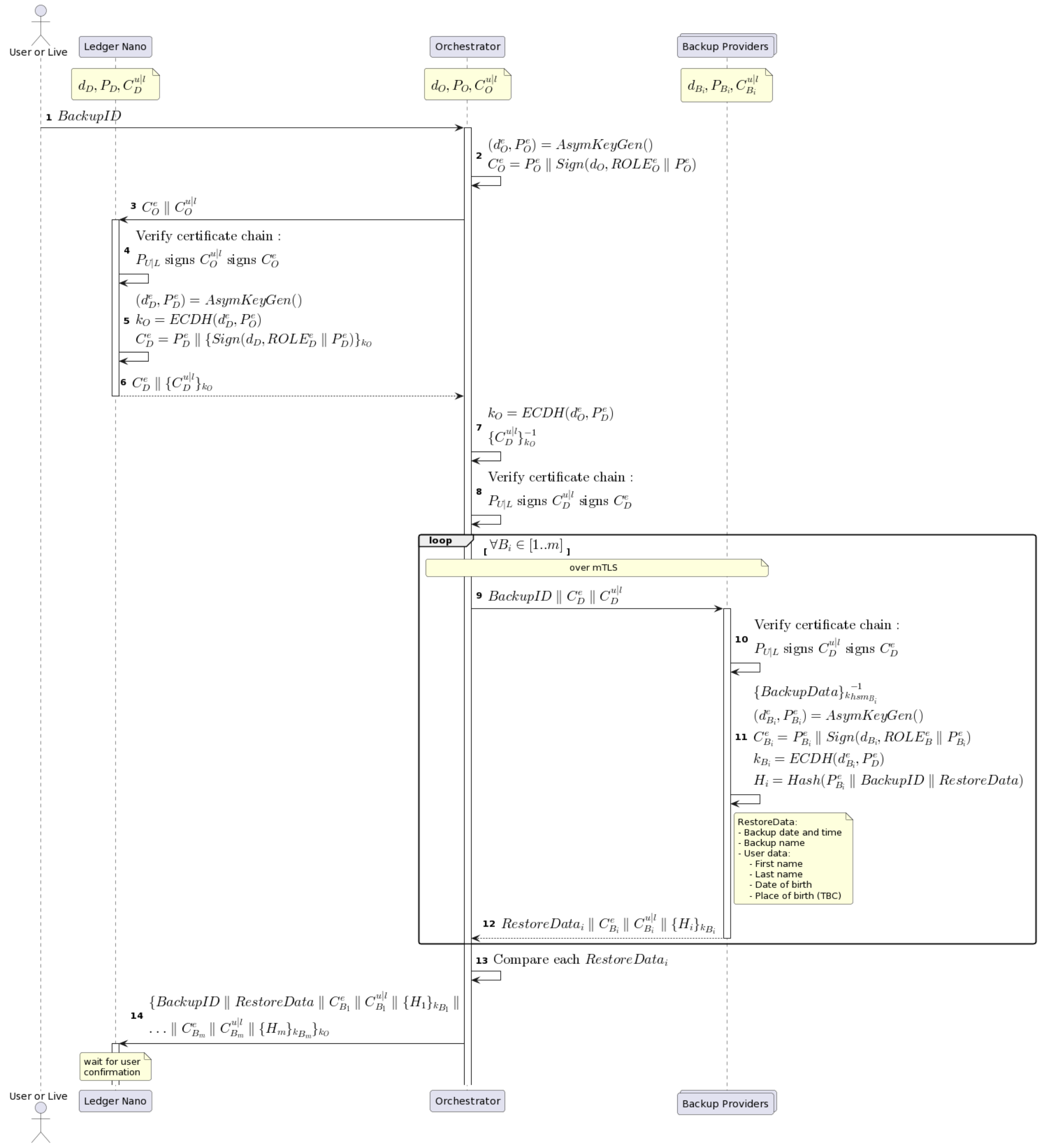
The Restore flow has five main stages:


Restore
Trigger Restore
Authenticate & trigger Restore
Out-of-scope
Trigger restore
for Identity Data
Secure Channel Setup
Establish secure channels
Binds identity data to channels
Sends Id data to device
Confirm Identity
Data
OTSC generation
Generate OTSC code on both sides
Lock session and bind to device
Identity Verification
Identity Verifications
Perform identity verification
with each providers
IDV complete
Restore seed
Retrieve shares from storage
Send over secure channels
Recombine and Decrypt seed
Done

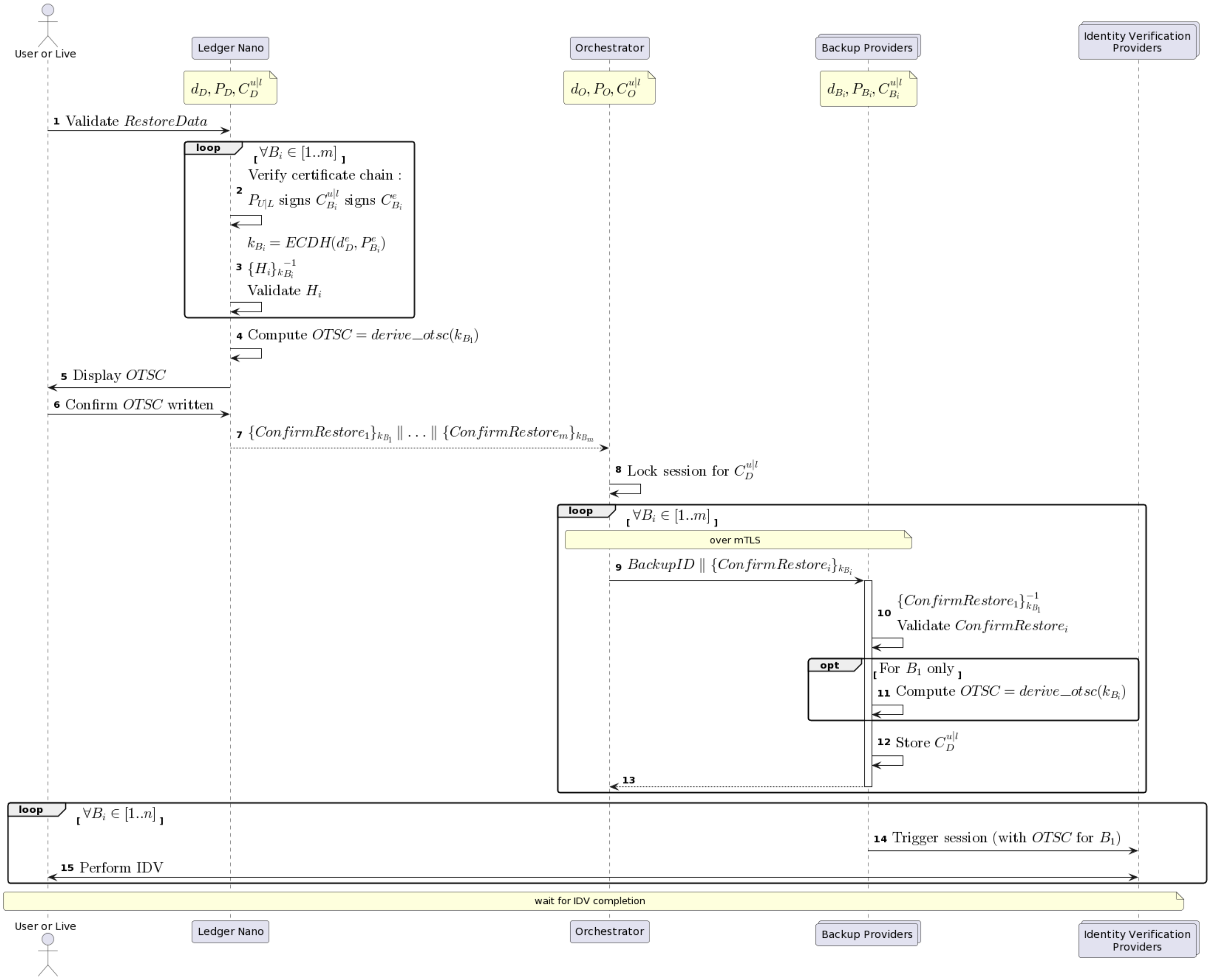
- Trigger stage: the user will authenticate and trigger a restore on a new device. This stage is out of scope of this document
- Secure Channels Setup: as for backup, the secure channels are established between device and backup providers (and orchestrator), bound together and to the identity data

- OTSC generation: a One Time Security Code is generated from the secure channel itself on both device and backup provider side, and will be used for binding IDV session
- Identity Verification: the user is redirected to and will perform two distinct Identity Verification with two different IDV providers (in current implementation, Tessi and OnFido)
- Restore seed stage: After verifying the IDV results, the backup providers will release the shares to the new device over secure channels. The device will recombine the shares into the encrypted seed and decrypt the seed to restore functions.

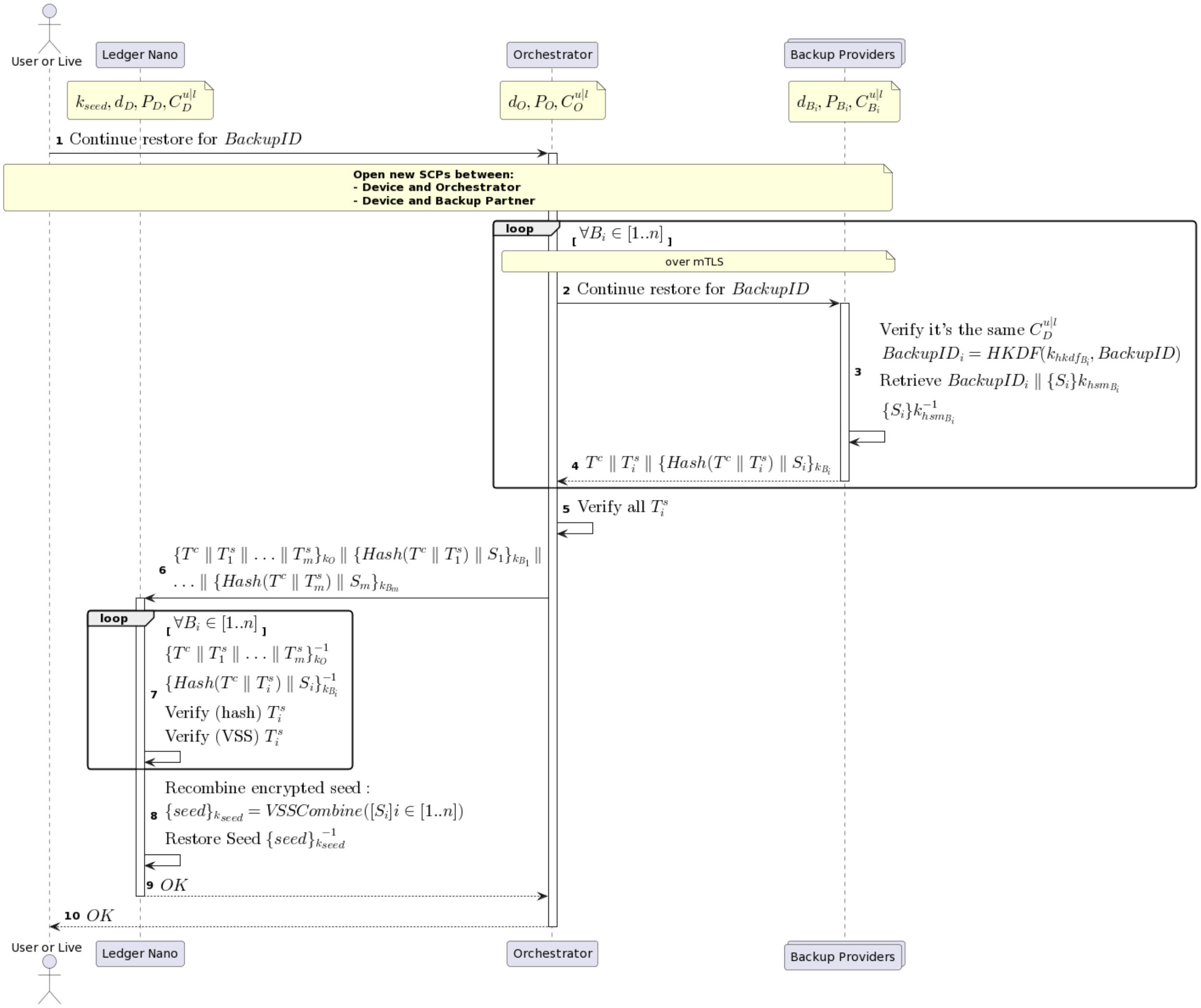
Secure Channels Setup


User or Live
Ledger Nano
Orchestrator
Backup Providers
$d_D, P_D, C_D^{u|l}$
$d_O, P_O, C_O^{u|l}$
$d_{B_i}, P_{B_i}, C_{B_i}^{u|l}$
1 $BackupID$
2 $(d_O^e, P_O^e) = AsymKeyGen()$
$C_O^e = P_O^e \parallel Sign(d_O, ROLE_O^e \parallel P_O^e)$
3 $C_O^e \parallel C_O^{u|l}$
4 Verify certificate chain :
$P_{U|L}$ signs $C_O^{u|l}$ signs C_O^e
5 $(d_D^e, P_D^e) = AsymKeyGen()$
$k_O = ECDH(d_D^e, P_O^e)$
$C_D^e = P_D^e \parallel \{Sign(d_D, ROLE_D^e \parallel P_D^e)\}_{k_O}$
6 $C_D^e \parallel \{C_D^{u|l}\}_{k_O}$
7 $k_O = ECDH(d_O^e, P_D^e)$
$\{C_D^{u|l}\}_{k_O}^{-1}$
8 Verify certificate chain :
$P_{U|L}$ signs $C_D^{u|l}$ signs C_D^e
loop
[$\forall B_i \in [1..m]$]
over mTLS
9 $BackupID \parallel C_D^e \parallel C_D^{u|l}$
10 Verify certificate chain :
$P_{U|L}$ signs $C_D^{u|l}$ signs C_D^e
11 $\{BackupData\}_{k_{hsm_{B_i}}}^{-1}$
$(d_{B_i}^e, P_{B_i}^e) = AsymKeyGen()$
$C_{B_i}^e = P_{B_i}^e \parallel Sign(d_{B_i}, ROLE_B^e \parallel P_{B_i}^e)$
$k_{B_i} = ECDH(d_{B_i}^e, P_D^e)$
$H_i = Hash(P_{B_i}^e \parallel BackupID \parallel RestoreData)$
RestoreData:
- Backup date and time
- Backup name
- User data:
- First name
- Last name
- Date of birth
- Place of birth (TBC)
12 $RestoreData_i \parallel C_{B_i}^e \parallel C_{B_i}^{u|l} \parallel \{H_i\}_{k_{B_i}}$
13 Compare each $RestoreData_i$
14 $\{BackupID \parallel RestoreData \parallel C_{B_1}^e \parallel C_{B_1}^{u|l} \parallel \{H_1\}_{k_{B_1}} \parallel$
$\ldots \parallel C_{B_m}^e \parallel C_{B_m}^{u|l} \parallel \{H_m\}_{k_{B_m}}\}_{k_O}$
wait for user confirmation
User or Live
Ledger Nano
Orchestrator
Backup Providers

OTSC and IDV stages


User or Live
Ledger Nano
Orchestrator
Backup Providers
Identity Verification Providers
$d_D, P_D, C_D^{u|l}$
$d_O, P_O, C_O^{u|l}$
$d_{B_i}, P_{B_i}, C_{B_i}^{u|l}$
1 Validate $RestoreData$
loop
[$\forall B_i \in [1..m]$]
Verify certificate chain :
2 $P_{U|L}$ signs $C_{B_i}^{u|l}$ signs $C_{B_i}^{e}$
$k_{B_i} = ECDH(d_D^e, P_{B_i}^e)$
3 $\{H_i\}_{k_{B_i}}^{-1}$
Validate H_i
4 Compute $OTSC = derive\_otsc(k_{B_1})$
5 Display $OTSC$
6 Confirm $OTSC$ written
7 $\{ConfirmRestore_1\}_{k_{B_1}} \parallel \ldots \parallel \{ConfirmRestore_m\}_{k_{B_m}}$
8 Lock session for $C_D^{u|l}$
loop
[$\forall B_i \in [1..m]$]
over mTLS
9 $BackupID \parallel \{ConfirmRestore_i\}_{k_{B_i}}$
10 $\{ConfirmRestore_1\}_{k_{B_1}}^{-1}$
Validate $ConfirmRestore_i$
opt
[For B_1 only]
11 Compute $OTSC = derive\_otsc(k_{B_i})$
12 Store $C_D^{u|l}$
13
loop
[$\forall B_i \in [1..n]$]
14 Trigger session (with $OTSC$ for B_1)
15 Perform IDV
wait for IDV completion
User or Live
Ledger Nano
Orchestrator
Backup Providers
Identity Verification Providers

Seed Restore stage



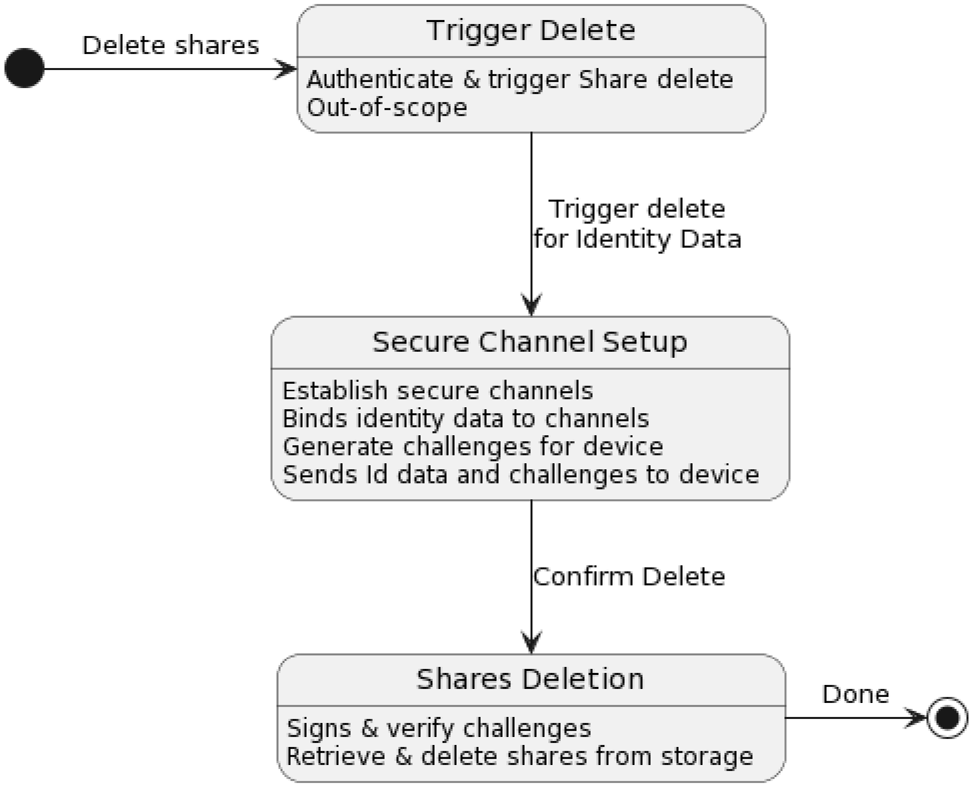
User or Live
Ledger Nano
Orchestrator
Backup Providers
$k_{seed}, d_D, P_D, C_D^{u|l}$
$d_O, P_O, C_O^{u|l}$
$d_{B_i}, P_{B_i}, C_{B_i}^{u|l}$
1 Continue restore for $BackupID$
Open new SCPs between:
- Device and Orchestrator
- Device and Backup Partner
loop
[$\forall B_i \in [1..n]$]
over mTLS
2 Continue restore for $BackupID$
3 Verify it's the same $C_D^{u|l}$
$BackupID_i = HKDF(k_{hkdf_{B_i}}, BackupID)$
Retrieve $BackupID_i \parallel \{S_i\}k_{hsm_{B_i}}$
$\{S_i\}k_{hsm_{B_i}}^{-1}$
4 $T^c \parallel T_i^s \parallel \{Hash(T^c \parallel T_i^s) \parallel S_i\}_{k_{B_i}}$
5 Verify all T_i^s
6 $\{T^c \parallel T_1^s \parallel \ldots \parallel T_m^s\}_{k_O} \parallel \{Hash(T^c \parallel T_1^s) \parallel S_1\}_{k_{B_1}} \parallel$
$\ldots \parallel \{Hash(T^c \parallel T_m^s) \parallel S_m\}_{k_{B_m}}$
loop
[$\forall B_i \in [1..n]$]
7 $\{T^c \parallel T_1^s \parallel \ldots \parallel T_m^s\}_{k_O}^{-1}$
$\{Hash(T^c \parallel T_i^s) \parallel S_i\}_{k_{B_i}}^{-1}$
Verify (hash) T_i^s
Verify (VSS) T_i^s
8 Recombine encrypted seed :
$\{seed\}_{k_{seed}} = VSSCombine([S_i]i \in [1..n])$
Restore Seed $\{seed\}_{k_{seed}}^{-1}$
9 OK
10 OK
User or Live
Ledger Nano
Orchestrator
Backup Providers

Delete

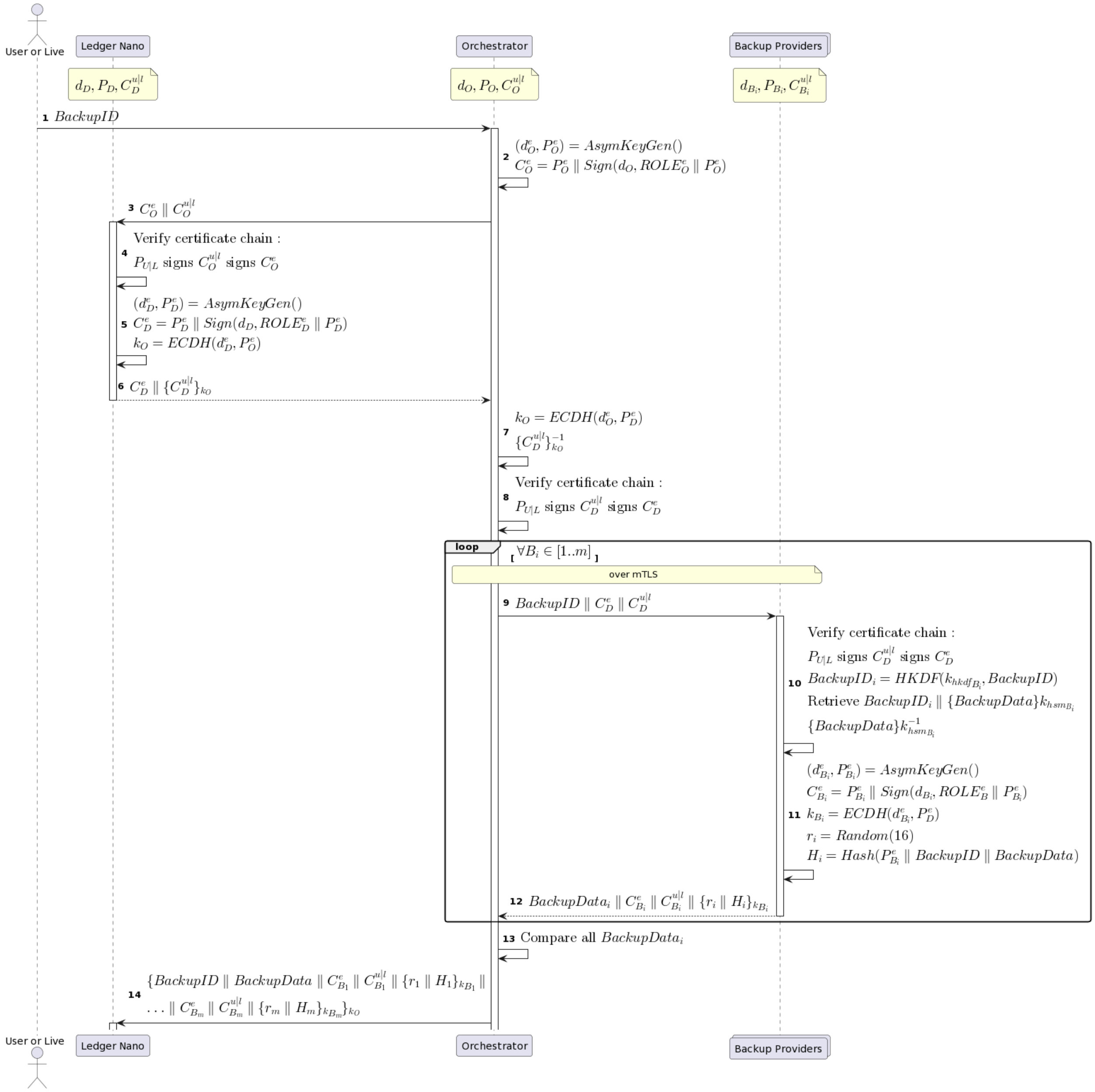
The Delete flow is used to delete a seed's shares backups securely, by proving that a user has a real device with the corresponding seed initialized (so deleting the backup won't leave the funds potentially unreachable).

A Deletion flow has three main stage:

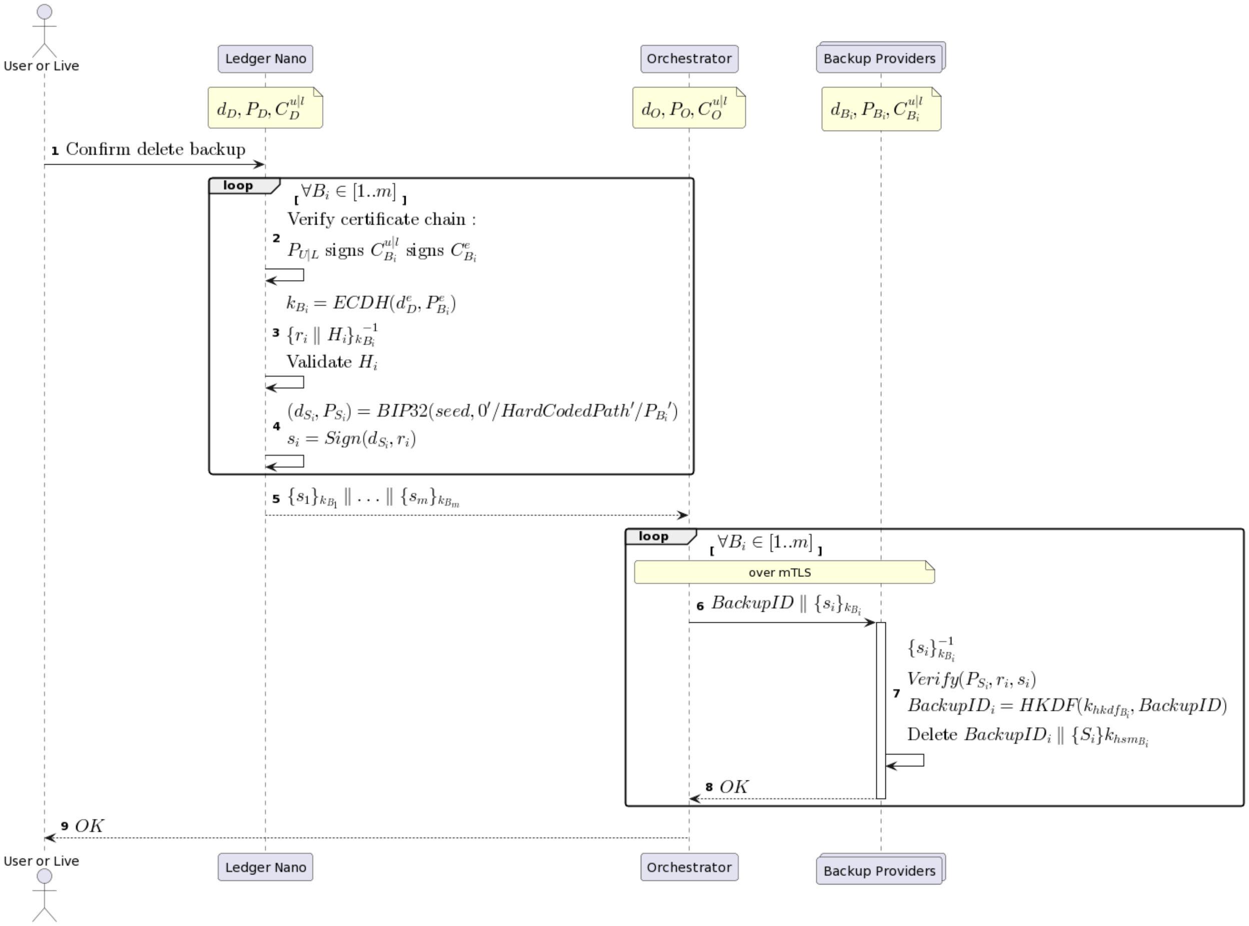
- Trigger stage: the user authenticates to Ledger Recover, and triggers a secure shares deletion. This is out of scope of this paper.
- Secure Channel Setup: the secure channels are established between device and backup providers (and orchestrator), bound together and to the identity data. Challenges are exchanged between backup providers and the device.
- Shares deletion: Challenges are signed and exchanged to prove the seed on the device is the right one. Shares are deleted once checked.


Delete shares
Trigger Delete
Authenticate & trigger Share delete
Out-of-scope
Trigger delete
for Identity Data
Secure Channel Setup
Establish secure channels
Binds identity data to channels
Generate challenges for device
Sends Id data and challenges to device
Confirm Delete
Shares Deletion
Signs & verify challenges
Retrieve & delete shares from storage
Done

Secure Channels Setup


User or Live
Ledger Nano
Orchestrator
Backup Providers
$d_D, P_D, C_D^{u|l}$
$d_O, P_O, C_O^{u|l}$
$d_{B_i}, P_{B_i}, C_{B_i}^{u|l}$
1 $BackupID$
2 $(d_O^e, P_O^e) = AsymKeyGen()$
$C_O^e = P_O^e \parallel Sign(d_O, ROLE_O^e \parallel P_O^e)$
3 $C_O^e \parallel C_O^{u|l}$
4 Verify certificate chain :
$P_{U|L}$ signs $C_O^{u|l}$ signs C_O^e
5 $(d_D^e, P_D^e) = AsymKeyGen()$
$C_D^e = P_D^e \parallel Sign(d_D, ROLE_D^e \parallel P_D^e)$
$k_O = ECDH(d_D^e, P_O^e)$
6 $C_D^e \parallel \{C_D^{u|l}\}_{k_O}$
7 $k_O = ECDH(d_O^e, P_D^e)$
$\{C_D^{u|l}\}_{k_O}^{-1}$
8 Verify certificate chain :
$P_{U|L}$ signs $C_D^{u|l}$ signs C_D^e
loop
$[\forall B_i \in [1..m]]$
over mTLS
9 $BackupID \parallel C_D^e \parallel C_D^{u|l}$
10 Verify certificate chain :
$P_{U|L}$ signs $C_D^{u|l}$ signs C_D^e
$BackupID_i = HKDF(k_{hkdf_{B_i}}, BackupID)$
Retrieve $BackupID_i \parallel \{BackupData\}k_{hsm_{B_i}}$
$\{BackupData\}k_{hsm_{B_i}}^{-1}$
11 $(d_{B_i}^e, P_{B_i}^e) = AsymKeyGen()$
$C_{B_i}^e = P_{B_i}^e \parallel Sign(d_{B_i}, ROLE_B^e \parallel P_{B_i}^e)$
$k_{B_i} = ECDH(d_{B_i}^e, P_D^e)$
$r_i = Random(16)$
$H_i = Hash(P_{B_i}^e \parallel BackupID \parallel BackupData)$
12 $BackupData_i \parallel C_{B_i}^e \parallel C_{B_i}^{u|l} \parallel \{r_i \parallel H_i\}_{k_{B_i}}$
13 Compare all $BackupData_i$
14 $\{BackupID \parallel BackupData \parallel C_{B_1}^e \parallel C_{B_1}^{u|l} \parallel \{r_1 \parallel H_1\}_{k_{B_1}} \parallel$
$\ldots \parallel C_{B_m}^e \parallel C_{B_m}^{u|l} \parallel \{r_m \parallel H_m\}_{k_{B_m}}\}_{k_O}$
User or Live
Ledger Nano
Orchestrator
Backup Providers

Shares Deletion


User or Live
Ledger Nano
Orchestrator
Backup Providers
$d_D, P_D, C_D^{u|l}$
$d_O, P_O, C_O^{u|l}$
$d_{B_i}, P_{B_i}, C_{B_i}^{u|l}$
1 Confirm delete backup
loop
[$\forall B_i \in [1..m]$]
2 Verify certificate chain :
$P_{U|L}$ signs $C_{B_i}^{u|l}$ signs $C_{B_i}^{e}$
3 $k_{B_i} = ECDH(d_D^e, P_{B_i}^e)$
$\{r_i \parallel H_i\}_{k_{B_i}}^{-1}$
Validate H_i
4 $(d_{S_i}, P_{S_i}) = BIP32(seed, 0'/HardCodedPath'/P_{B_i}')$
$s_i = Sign(d_{S_i}, r_i)$
5 $\{s_1\}_{k_{B_1}} \parallel \ldots \parallel \{s_m\}_{k_{B_m}}$
loop
[$\forall B_i \in [1..m]$]
over mTLS
6 $BackupID \parallel \{s_i\}_{k_{B_i}}$
7 $\{s_i\}_{k_{B_i}}^{-1}$
$Verify(P_{S_i}, r_i, s_i)$
$BackupID_i = HKDF(k_{hkdf_{B_i}}, BackupID)$
Delete $BackupID_i \parallel \{S_i\}k_{hsm_{B_i}}$
8 OK
9 OK
User or Live
Ledger Nano
Orchestrator
Backup Providers

A Note on Identity Verification

Identity Verification as an authentication mechanism

The underlying assumption for choosing Identity Verification as an authentication mechanism, is that Ledger Recover is intended as a service used for *Disaster Recovery*: a situation in which our customer has lost access to potentially all his devices (wallets, PCs or phones), lost his PINs, email passwords and all traditional electronic verification forms.

In this context, the legal citizen identity is the only form of strong authentication that can be reconstructed from scratch and a good basis for an authentication in case of massive failure of everything else.

During the backup process, it’s enough to perform only one IDV because the user is going to confirm their identity on the device before exporting the shares.

However, each backup provider should perform an independent IDV during the restore. The rationale is that it would be harder for an attacker to compromise multiple IDV services instead of one. There are two possible attacks:

- The attacker is able to fool the identity verification and impersonate the user.
- The attacker is able to bypass the IDV process and send the expected IDV data to the Backup Providers without verifying their identity.

Both attacks are harder if we use multiple IDV services during the restore, which is the most critical step since it can lead to the loss of funds.

Step	**Number of IDVs**
Backup	1
Restore	1 per backup provider

Backup/Restore data

The backup & restore data, extracted from IDVs and verified on the device during both backup and restore flows, is designed to hold a *pivotal identity*: the minimal set of attributes that uniquely and legally identify a physical person.

Most governments have a way to restore or create a legal identity document from proving physically this set of attributes in person.

Attribute	**Description**
First Name	All legal first names of the person, as seen on ID documents
Last Name	Legal last name as seen on ID documents
Date of Birth	Legal date of birth
Place of Birth	Full name of the place of birth

Note that the corresponding fields in the protocol contain the full representation of these attributes even if the device itself is not able to display them fully.

Conclusion

In conclusion, the Ledger Recover cryptographic protocol provides a robust, secure, and efficient system for backing up and restoring a device seed. With its end-to-end encryption, non-repudiation characteristics, privacy measures, and minimalistic yet effective design, it ensures a secure environment for data operations. The resilience provided by the 2-out-of-3 scheme and the Shamir Secret Sharing variant, coupled with the high degree of share transport and storage security, makes this protocol not only resistant to potential threats but also adaptable to various scenarios.

Moreover, the ability for expert users to run the protocol independently from Ledger underscores its flexibility and commitment to self-custody. These unique features collectively affirm the Ledger Recover cryptographic protocol as a comprehensive solution for device seed backup and restoration. Its innovative strategies stand as a testament to the potential of cryptographic technology, providing a reliable and user-centric choice in the ever-evolving digital landscape.

Afterwords

Ledger Recover is eligible for the Ledger Bug Bounty program. Please send feedback to bounty@ledger.fr.