

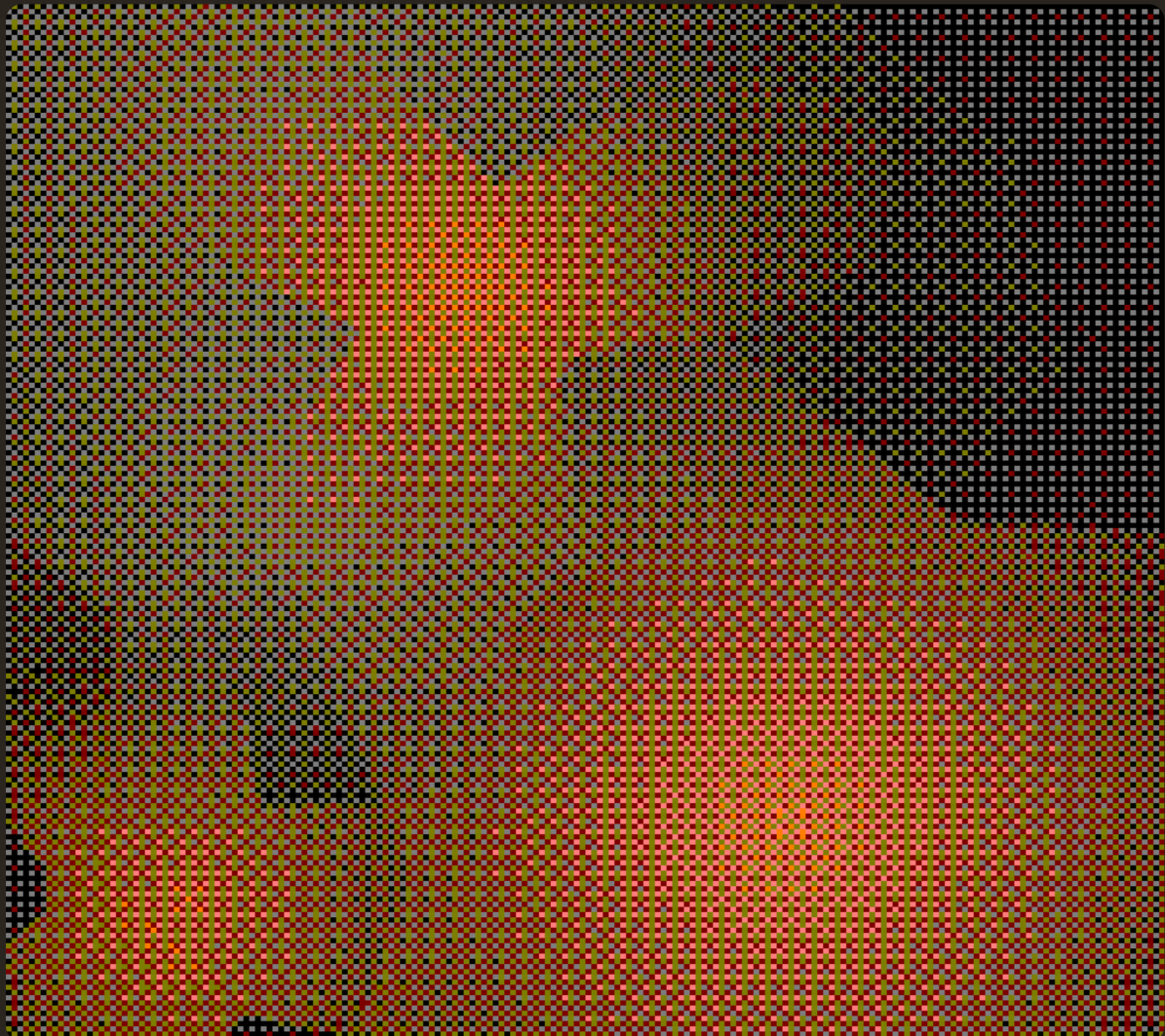
ZKSECURITY

Audit of Derive's orderbook v3

DERIVE

• SEPTEMBER 7TH,
2026

• TECHNICAL
REPORT



Introduction

On September 7th, 2026, zkSecurity began a two-week security audit of Derive's orderbook v3 with two consultants. We reviewed the [orderbook repository](#) at commit [8c53063c](#).

Scope

The audit covered the following components:

Rust

- The main entry point for the SP1 guest program: `crates/protocol/programs/program/src/main.rs`.
- The guest state management library: `crates/protocol/state/src/`, excluding legacy migration code.
- `crates/protocol/lib/src/batch/`, excluding migration-specific code.
- The guest data-availability verification library: `crates/protocol/celestia-verify/src/`.

Solidity

We reviewed the following Solidity contracts and scripts:

- `contracts/src/VerifiableApp.sol`
- `contracts/src/OnchainActionManager.sol`
- `contracts/src/WithdrawalOutbox.sol`

- `contracts/src/SpotVault.sol`
- `contracts/src/encoding/ActionEncoding.sol`

The v3 rollup is expected to start from a state migrated from the running v2 deployment. The correctness of that migration, and of the resulting initial state root, is out of scope.

We also reviewed the deployment scripts. At the time of the audit, `contracts/script/DeployMainnet.s.sol` was only a stub, so the review covered the applicable non-CEX portions of `contracts/script/DeployTestnet.s.sol`, which is expected to be representative of the mainnet deployment.

§ 03

Threat model

During the engagement, we considered two critical security properties of a rollup: correctness of state transitions, and liveness. Correctness is enforced by the contracts verifying an SP1 zkVM proof of a correct guest execution. Liveness depends on permissionless batch submission and on a data-availability layer checked by the guest.

We considered a malicious batch submitter. In both the permissioned and permissionless cases, the submitter should not be relied on for correctness. We also assumed that the contract owner is trusted, since the owner can perform sensitive operations, for example modifying the rollup state arbitrarily.

Technical Overview

Smart Contracts

The contracts are the L1 settlement and custody layer for the protocol. State transitions are executed off-chain and proven correct inside the SP1 zkVM. The smart contracts store a commitment to the state, verify the proof for each state transition, record L1 actions that must be visible to the zkVM, and hold and release the underlying tokens.

There are four core contracts, which are deployed behind `TransparentUpgradeableProxy` instances. Their implementations contain immutable references to the other contracts with which they are allowed to interact.

- `VerifiableApp` is responsible for state management and proof verification. Every batch is submitted to this contract, which verifies the SP1 proof, advances the state, action queues and Celestia DA anchor commitment. Proven withdrawals are sent to the withdrawal outbox.
- `OnchainActionManager` is responsible for accepting L1 actions. It commits them to two hash-chain queues, one for admin actions and one for public actions. Deposits and withdrawals are issued as actions; this contract is the main entry point for transfers.
- `SpotVault` is a simple vault that holds ERC-20 tokens and grants the outbox an allowance.
- `WithdrawalOutbox` executes proven withdrawals using a set of queues that ensure that fund outflow is rate-limited. The main motivation for this mechanism is that if a security incident happens, it is not possible to withdraw all funds immediately, and the monitoring systems will have time to respond.

Batch submission

A batch is a list of off-chain operations and L1 actions. Executing those actions advances the state. The correct batch execution is proven using the SP1 zkVM.

In particular, the sequencer can invoke `submitBatch` to advance the rollup state. The caller has to provide:

- the current and next state roots,
- the current and next L1 queue indices and action accumulators,
- a list of all withdrawals sequenced in the batch, together with its digest,
- the current and next Celestia anchors, current and next blob commitments, the oldest Celestia header time, and
- the SP1 proof bytes.

These values are used as public inputs to the proof verification, so that they are tied to a correct execution. The contract checks that:

- the caller is allowed to submit,
- all the claimed “current” values are equal to those stored in the contract’s state,
- the batch timestamps are non-decreasing from the last batch and are not in the future,
- the withdrawal array hashes to the digest,
- the SP1 verifier accepts the proof for the configured `programVKey` and reconstructed public values.

A batch begins as a totally ordered list of off-chain operations and L1 actions. The sequencer orders the operations, the settler executes the same transition over protocol state, and the SP1 guest proves the result. The state diff is posted to Celestia and its availability is also verified by the guest.

The caller then invokes `VerifiableApp.submitBatch` with:

- the current and proposed state roots;

- the old and new public-queue indices and old and new admin-queue indices;
- the four accumulator values at those indices;
- a digest of the batch's withdrawals and the full withdrawal array;
- the minimum and maximum operation timestamps;
- the previous and new Celestia header anchors, previous and new blob-chain commitments, and the oldest Celestia header time; and
- the SP1 proof bytes.

If all these checks succeed, the contract atomically updates the state root, both consumed queue indices, `lastOpTimestamp`, and the Celestia header and blob-chain tip. Then, to process withdrawals, `WithdrawalOutbox.executeWithdrawals` is invoked.

The `submitBatch` entrypoint is callable only by a designated `batchSubmitter` under normal operations. The contract keeps track of the submission time of the last action and the current time. If a deposit or withdrawal action is not included in a batch before some expiry date, the contract allows anyone to call `clearBatchSubmitter`, which sets the `batchSubmitter` to `address(0)`, allowing anyone to submit a new batch permissionlessly.

The Withdrawal Outbox

The withdrawal outbox is responsible for rate limiting the outflowing funds from the rollup. The main entry point is `executeWithdrawals`, which is only callable by the immutable `VERIFIABLE_APP` address.

For each withdrawal, the outbox derives a withdrawal ID computed as the SHA-256 hash of all its parameters. All processed IDs are stored, and duplicate withdrawals with an already processed ID are skipped.

The rate-limiting feature is implemented using different buckets, which distribute funds at different rates and have independent queues. Every configured bucket contains:

```

Bucket {
  throttle: {
    capacity,          // maximum available allowance, USD with 6 decimals
    refillRate,        // allowance added per second, USD with 6 decimals
    bucket,            // currently available allowance
    lastRefill
  },
  queueHead,
  queueTail,
  lastGuardianRefill
}

```

A withdrawal resolves to the first configured bucket in this order:

1. `(recipient, asset)`: one specialized bucket for some specific address and asset,
2. `(recipient, address(0))`: one specialized bucket for all the assets with the same recipient,
3. `(address(0), address(0))`: the shared global bucket.

The capacity of a bucket represents the maximum USD allowed to flow out at one time, while the refill rate is the hourly rate of allowed outflow. Since these values are in USD, the payout cost needs to be computed from the asset's registered static price and decimals:

```
chargeUsdX6 = ceil(amount * staticPriceX6 / 10**tokenDecimals)
```

The minimum permitted configuration is a `$100` capacity and approximately `$100/hour` refill, and the refill rate cannot exceed capacity. The static price is not queried through an oracle but is set manually by the owner. This may lead to the static price misaligning with the real price of the asset, but this is fine as long as it corresponds to a rough estimate of the price.

If a bucket's FIFO queue is empty, a new withdrawal can be paid fully or partially from the current allowance. Any unpaid remainder is appended to that bucket's queue. If the queue is already non-empty, the whole new withdrawal is appended to the queue. Anyone may call

`processQueue(recipient, asset, maxItems)` after allowance has refilled. It processes queued withdrawals from `queueHead` up to the current bucket allowance.

When the outbox is paused, proven withdrawals are still admitted and marked consumed, but they are enqueued. Withdrawals resume normally from the queues after unpauses.

Any failed settlement is removed from the ordinary flow and stored in `deferredWithdrawals` as a commitment to the withdrawal fields. The owner can later call `deferredProcess` with matching fields and a chosen native-gas limit to re-attempt the withdrawal. This manual recovery bypasses the normal throttle, so it is a trusted owner power.

For a native-ETH withdrawal, the outbox pulls WETH from the vault, unwraps it, and tries to send ETH with a 50,000-gas limit. If the native transfer fails, it wraps the value back into WETH and transfers WETH to the recipient so a receiver that rejects ETH does not block its bucket's FIFO queue.

The Guardian role

The owner can set an address to be the outbox guardian, which is an emergency operational role with two powers: it can pause payouts and, while the contract is unpaused, it can refill an already configured bucket immediately. This refill is also subject to a rate limit, so that a guardian that just refilled a bucket cannot immediately invoke another refill of the same bucket. Only the owner can unpauses the outbox.

Protocol State

Derive represents its protocol state as **16 sparse Merkle trees and seven global records**, all committed by a single 32-byte root. Together, they describe account ownership and permissions, trading positions, market configuration, lending, risk management, and vault holdings. A batch changes this state, and the SP1 guest proves the transition from the previous root to the next one.

The guest receives a compact representation called `SparseMerkleState`. It contains the records that the batch accesses and the hashes needed to connect those records to the full state commitment. This allows execution to depend on a small portion of the state while still committing to the state as a whole.

What the Trees Represent

Each tree represents a distinct kind of record. This separates, for example, a user's positions from an asset's configuration or a vault shareholder's balance.

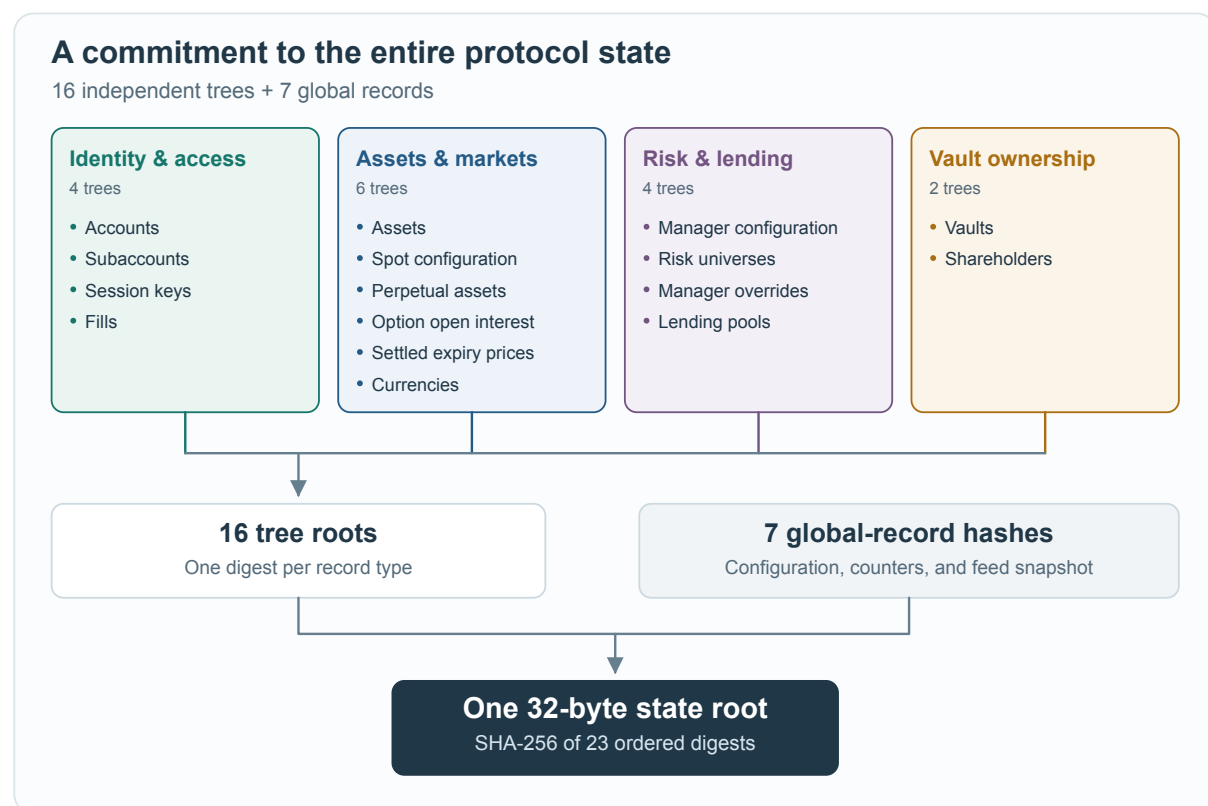


Figure 1. The trees are grouped by purpose for illustration. The protocol combines their roots in a fixed order, followed by the hashes of the seven global records.

The following table describes all 16 trees. A risk universe is a grouping under which market exposure and risk-management parameters are configured.

TREE	WHAT IT REPRESENTS
Accounts	Wallet-level settings, including the fallback subaccount, subaccount count, and permitted recipients.
Subaccounts	Individual trading portfolios: ownership, the assigned risk manager, spot balances, derivative positions, and liquidation-related state.
Session keys	Delegated signing permissions, including the authorizing wallet, permitted actions and subaccounts, and expiry time.
Fills	Cumulative filled amounts for each subaccount and order nonce, together with the time after which the order can no longer be filled.
Assets	The protocol's registered asset definitions and configuration.
Spot configuration	Spot-asset metadata such as token decimals and deposit limits.
Perpetual assets	Funding state and market configuration for a perpetual asset within a risk universe, including its frozen price when applicable.
Option open interest	Aggregate outstanding option exposure for an option asset within a risk universe.
Settled expiry prices	Final settlement prices indexed by currency and expiry time.
Currencies	Oracle configuration for each currency, including configuration for the different feed types.
Manager configuration	The risk-management configuration selected by a subaccount's manager identifier.

TREE	WHAT IT REPRESENTS
Risk universes	Universe-wide parameters, including exposure caps, security-module settings, and liquidation auction parameters.
Manager parameter overrides	Per-subaccount risk-manager parameter overrides used for delayed liquidations.
Lending pools	Supply and borrowing indices and interest-rate models for each asset within a risk universe.
Vaults	Vault-level accounting and configuration, including total shares and fee-related state. The vault's trading portfolio is held in its subaccount.
Shareholders	Each holder's vault shares, deposit timing, and nonce used to prevent replay of vault intents.

Seven records are committed directly, without separate Merkle trees: the latest subaccount identifier, a global auction-parameter record, protocol fee configuration, the wrapped native-ETH asset, action-signing domain parameters, global vault configuration, and the held feed snapshot.

The commitment has several further design choices:

- **Zero represents emptiness at every level.** An absent leaf has a zero digest. A parent whose two children are zero also has a zero digest. Other parents are the SHA-256 hash of their ordered child digests. Unlike sparse trees that derive a different empty-subtree hash at each depth, this construction uses the same zero value throughout an empty region.
- **Leaf hashes identify the tree.** A present record is hashed together with a tag identifying its tree. Identically encoded values in different trees therefore have different hash inputs. The key selects the leaf's position, while the tag and encoded value determine its digest.

- **The final commitment combines several trees.** Each tree contributes its root, and each global record contributes a hash of its value. The protocol hashes the 23 digests in a fixed order to obtain the final state root. This last step is one hash of a concatenation, rather than another binary Merkle tree.

The ordering and record encodings are part of the commitment definition. Changing them changes the root even when the interpreted state appears unchanged.

A Compact Witness for Each Batch

The host maintains the full state and identifies the keys that the batch reads or writes. It then supplies their values from the **pre-batch state**, together with the sibling hashes needed to reconstruct the relevant tree roots. Untouched subtrees are represented by their hashes.

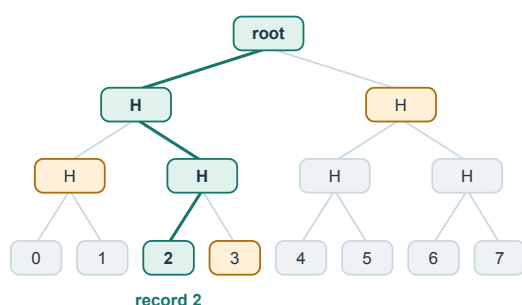
The guest uses this witness to reconstruct the state commitment and check it against the claimed old root. It then executes the batch and recomputes the affected tree roots, reusing the hashes of unchanged subtrees. Combining the resulting tree roots with the global-record hashes produces the new state root.

Expand only the path the batch needs

An eight-leaf example; real key spaces are much larger.

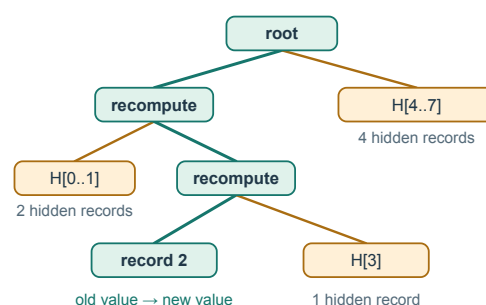
Full tree

Host retains all records



Compact witness for record 2

Guest receives a value and three sibling hashes



● **Recompute this path**

● **Reuse these sibling hashes**

● **Other values stay on the host**

The same neighboring hashes authenticate both the old and the updated tree root.

Figure 2. An illustrative three-level tree. To update record 2, the guest needs its old value and the three highlighted sibling hashes. The sibling hashes are reused when computing the new root. The actual protocol uses much deeper trees.

When several records are accessed, their paths can share branches. The witness combines those paths and avoids supplying a separate, redundant proof for every record. Entirely untouched trees contribute their existing roots. The seven global records are included in full.

Three cases must remain distinct:

CASE	MEANING
Present record	The witness supplies the value at the key's committed position.
Proven absence	The witness includes that position as an empty leaf. A batch can create a new record there.
Key omitted from the witness	No value or absence has been supplied for that key. It cannot be treated as proof that the record does not exist.

A deletion turns an accessed leaf into an empty leaf while retaining its position in the witness. Creation replaces a witnessed empty leaf with a present value. In both cases, the guest reuses the hashes of unchanged neighboring subtrees. A compact witness proves the supplied records and positions; it does not by itself establish that an enumerated set of records contains every live entry in a tree.

Data Availability

The DA light client proves that the state changes produced by a batch were published in Celestia blocks that descend from a Celestia header already trusted on the L1. This mechanism is essential for the liveness of the rollup, as this ensures that all data needed to continue sequencing is available to

users. In particular, a diff of the state is serialized, and then it is proved that it is contained in a sequence of blobs on Celestia. The guest additionally runs a light client of the Celestia blockchain, which ensures the validity of DA proofs.

The system has two related continuity mechanisms:

1. A **Celestia header anchor** authenticates the Celestia chain and its validator set. It represents a trusted header, which is assumed to be a real header on Celestia. It is used as an anchor for validating the next headers.
2. A **blob commitment chain** orders the state-diff blobs within a batch and across batches.

Trust anchor

The light client's trust anchor is a Celestia block identified by:

- `height` : the Celestia block height; and
- `header_hash` : the SHA-256 Tendermint/CometBFT header hash.

Ethereum stores this pair as `celestiaTrustedHeight` and `celestiaTrustedHash`. It also stores `celestiaLastCommitment`, the separate application blob-chain tip. Together, these three values are the anchor on which the next batch's DA is based.

The contract owner can bootstrap the checkpoint by calling `setCelestiaAnchor`. The normal bootstrap tool reads the current Celestia head and initializes the blob tip to zero. The owner can also re-point all three values for recovery. We assume that the initial anchor has been set honestly by the owner of the contract.

The prover supplies the complete header at the anchored height as witness data in the guest. The guest then:

1. protobuf-decodes and self-validates it,
2. checks that its height equals the on-chain anchor height, and

3. recomputes its header hash and checks that it equals the on-chain anchor hash.

The full header also contains the set of all validators, which is now taken as a “trusted validator” set.

Header and validator-signature checking

The prover also provides additional headers, one for each block containing a relevant blob which stores diff data. Each header is first decoded and validated, including the verification of the number of validators which signed that header. The guest calls into the Rust libraries from the [Lumina](#) Celestia light node implementation.

Then, each header is also checked against the trusted anchor. If the header is exactly the next block after the anchor, the guest checks that:

- the target's validator-set hash equals the anchor's `next_validators_hash`, and
- the target's `last_block_id` points to the anchor header hash.

This link suffices to trust the new header given the anchor. If instead one or more heights are skipped, the guest verifies signatures on the target commit from validators in the **anchor's** validator set. The required overlap is:

```
tallied anchor-set voting power > floor(total anchor-set voting power / 3)
```

Additionally, the client requires that:

- More than two-thirds of the target's own validator set signed the target.
- More than one-third of the anchor validator set also signed the target.
- The target height and time are greater than the anchor height and time.
- The target is within a five-day trusting period.

Given that all these conditions hold true, the light client can trust the new header. This assumes that more than two-thirds of the trusted anchor set follow the protocol during the trusting period: A set containing more than

one-third of the trusted voting power must contain honest voting power and cannot authenticate a conflicting target.

Blob verification

The SP1 guest computes the state diff after executing the batch, serializes it, and splits it into chunks. These chunks are posted in Celestia blobs under the 10-byte namespace ID that consists of the first 10 bytes of `sha256("derive")`:

```
46023b2bfb8ba6dc6fdc
```

For each blob witness, the guest performs the following checks:

1. Decode and validate the blob's header against the trusted anchor, as mentioned before.
2. Read the witnessed `chunk_len` and consume exactly that many bytes from the serialized state diff.
3. Construct the expected blob data as `previous_blob_commitment (32 bytes) || state_diff_chunk`.
4. Construct a Celestia `Blob` object containing those bytes.
5. For each blob, read and verify the corresponding inclusion proof.

This proves that, under the assumption that the trusted header was an honest header, all blobs have been posted on Celestia. The commitment-chain mechanism ensures that the data is sequentially ordered and can be reconstructed by an external user.

Checks performed by the contracts

The guest commits the following DA fields as SP1 public values:

- previous anchor height and hash,
- new anchor height and hash,
- previous blob-chain tip,
- new blob-chain tip, and

- the oldest target header timestamp used by the batch.

`VerifiableApp` reconstructs those public values and verifies the SP1 proof. It also checks that the previous anchor and previous blob tip equal its stored values. This prevents a valid proof for a different DA history from advancing the current application state.

The contracts additionally check that the oldest target header is no more than two days old relative to `block.timestamp`, as the oldest target bounds all target blocks used by the batch.

If every contract check and the SP1 proof succeed, the batch submission stores the new state root, new Celestia anchor, and new blob-chain tip, which become the trust anchor and hash-chain seed for the next batch.

Findings

Below are listed the findings found during the engagement. High severity findings can be seen as so-called "priority 0" issues that need fixing (potentially urgently). Medium severity findings are most often serious findings that have less impact (or are harder to exploit) than high-severity findings. Low severity findings are most often exploitable in contrived scenarios, if at all, but still warrant reflection. Findings marked as informational are general comments that did not fit any of the other criteria.

HIGH

- #00** Malicious provers can settle a blob commitment that was never posted to Celestia
`crates/protocol/celestia-verify/src/verify.rs`

MEDIUM

- #01** The submitter stall timeout can outlast the Celestia anchor
`VerifiableApp.sol, crates/protocol/celestia-verify/src/verify.rs`

LOW

- #02** Admin actions can be sequenced out-of-order with user actions
`OnchainActionManager.sol, deposit/pricing.rs, set_spot_config.rs, withdraw/resolve.rs`
- #03** A deposit above the guest's uint128 range blocks the public queue
`OnchainActionManager.sol, onchain_action/deposit.rs`
- #04** Under-received deposits create unbacked protocol balances
`OnchainActionManager.sol`

- #05** A bucket configured under the native ETH label is not used for native withdrawals
`WithdrawalOutbox.sol`
- #06** A non-spot asset label can block the public action queue
`WithdrawalOutbox.sol`, `OnchainActionManager.sol`, `deposit/pricing.rs`

INFORMATIONAL

- #07** Batch proofs are not bound to an EVM deployment
`VerifiableApp.sol`, `batch/codec.rs`, `WithdrawalOutbox.sol`
- #08** Celestia header verification does not expire the trusted anchor
`crates/protocol/celestia-verify/src/verify.rs`
- #09** Operation timestamps can remain arbitrarily stale
`VerifiableApp.sol`
- #0a** Sparse Merkle witness updates rely on a fixed key set
`crates/protocol/state/src/sparse_merkle_state/mod.rs`, `merkel_tree.rs`

Malicious provers can settle a blob commitment that was never posted to Celestia

crates/protocol/celestia-verify/src/verify.rs

Description. `verify_chunk` proves that a batch's state-diff blob is available on Celestia by rebuilding the blob from the guest's own bytes and verifying a `ShareProof` against the target block's data root. The witness supplies only the proof halves (`share_proofs` and `row_proof`); the share bytes are derived locally.

```
pub fn verify_chunk(
    expected_chunk: Vec<u8>,
    namespace: Namespace,
    target: &ExtendedHeader,
    blob_witness: &BlobWitness,
) -> Result<[u8; COMMITMENT_LEN], DaError> {
    let blob = Blob::new(namespace, expected_chunk,
None).map_err(DaError::Blob)?;
    let data: Vec<[u8; SHARE_SIZE]> = blob
        .to_shares()
        .map_err(DaError::Blob)?
        .iter()
        .map(|share| *share.data())
        .collect();
    let proof = ShareProof {
        data,
        namespace_id: namespace,
        share_proofs: blob_witness.share_proofs.clone(),
        row_proof: blob_witness.row_proof.clone(),
    };
    proof
        .verify(target.dah.hash())
        .map_err(DaError::ShareProof)?;
    Ok(*blob.commitment.hash())
}
```

`ShareProof::verify` (celestia-types 1.0.0) authenticates each per-row NMT range independently and checks that those ranges' lengths sum to the reconstructed share count. It does not require the proven shares to occupy a contiguous row-major span of a single blob. `RowProof::verify` likewise checks that each claimed row root is in the DAH; it does not bind `start_row` / `end_row` to the merkle-proof indices, so the rows themselves need not be adjacent in the square.

The figure below shows how independently valid row ranges can mimic one target blob without forming a contiguous span in row-major order.

Correct: one contiguous blob

5 × 4 block

0	1	2	S₀ 3	S₁ 4
S₂ 5	S₃ 6	S₄ 7	S₅ 8	S₆ 9
S₇ 10	S₈ 11	12	13	14
15	16	17	18	19

Row-major array

...	S₀	S₁	S₂	S₃	S₄	S₅	S₆	S₇	S₈	...
0-2	3	4	5	6	7	8	9	10	11	12-19

Malformed: shares from separate runs

5 × 4 block

S₀ 0	S₁ 1	2	3	4
5	S₂ 6	S₃ 7	8	9
10	S₄ 11	S₅ 12	S₆ 13	14
S₇ 15	S₈ 16	17	18	19

Row-major array

S₀ 0	S₁ 1	...	S₂ 6	S₃ 7	...	S₄ 11	S₅ 12	S₆ 13	.	S₇ 15	S₈ 16	...
		2-5			8-10				14			17-19

A five-column, four-row original data square and its row-major array. Numbers mark share positions in row-major order; gray cells and dots are unrelated shares.

The upper panel shows the target shares as one continuous sequence from 3rd cell through 11th cell. The lower panel places the same shares in four separated ranges. Each run is contiguous within its row, so the per-row proofs can authenticate all nine shares even though they do not form one posted blob.

`verify_chunk` nevertheless succeeds and returns the target blob's commitment. `verify_blob_da` writes that value into `new_last_blob_commitment`, which L1 stores as `celestiaLastCommitment`. The commitment does not identify any blob submitted to Celestia, so `blob.Get` cannot retrieve the target blob.

Even if `ShareProof` rejected non-contiguous shares, as its Rustdoc suggests, that check would still not guarantee that the shares cover exactly one blob. A malicious prover could supply the state diff as a contiguous subset of a larger blob.

Impact. In-circuit DA is supposed to guarantee that reconstructors can fetch the state-diff blob from Celestia using the L1-stored `celestiaLastCommitment`. After this proof, that lookup returns nothing.

Independent reconstruction of rollup state from L1 therefore fails for the settled batch.

A malicious sequencer or prover can execute a valid state transition, post only decoy blobs, and settle. Withdrawals in that batch still pay out, because they are committed in the public values, but the state diff that reconstructors need in order to replay the new root is not retrievable as a Celestia blob.

Recommendation. Wait for the Celestia team to provide a patch, then bump `celestia-types` to version `1.2.0-rc.1`. In `verify_chunk`, replace `ShareProof::verify` with `BlobProof::verify`, which contains additional checks to ensure those shares form exactly one blob.

Client response. Fixed in [#2453](#) by bumping `celestia-types` to `1.2.0-rc.1` and using `BlobProof::verify` in `verify_chunk`.

The submitter stall timeout can outlast the Celestia anchor

VerifiableApp.sol, crates/protocol/celestia-verify/src/verify.rs

Description. `setSubmitterStallTimeout` permits a timeout of up to 28 days before `clearBatchSubmitter` opens batch submission to anyone:

```
if (newTimeout > 28 days) revert InvalidSubmitterStallTimeout();
```

The guest limits the gap between the stored Celestia anchor and each blob header to five days:

```
pub const TRUSTING_PERIOD_SECS: u64 = 5 * 24 * 60 * 60;
```

On L1, `_consumeCelestia` requires the oldest blob header in the batch to be no more than two days old:

```
uint64 public constant MAX_DA_HEADER_AGE = 2 days;
```

Therefore, the stored anchor must be at most seven days old to accept a batch. If the submitter stall timeout is set longer than that, the permissionless recovery path may be blocked by an expired anchor.

Impact. A permissionless submitter may be unable to produce an acceptable DA proof from the stale anchor. Batch submission then remains blocked until the owner changes the anchor.

Recommendation. Ensure permissionless recovery can begin while the stored anchor can still reach fresh Celestia headers, or provide an anchor recovery path that does not depend on the stalled submitter.

Client response. Acknowledged. [#2401](#) added a Blobstream recovery anchor, so that, after `clearBatchSubmitter`, a replacement submitter could chain from a relayed header at or above the stored head. Celestia is

deprecating Blobstream, and [#2472](#) removed that path. The client will carry this gap until an Ethereum data-availability path is added in v3.1.

Admin actions can be sequenced out-of-order with user actions

OnchainActionManager.sol, deposit/pricing.rs, set_spot_config.rs,
withdraw/resolve.rs

Description. Public deposits and admin configuration changes advance through separate action queues. For example, enqueueing a correction to a spot asset's `underlying_erc20_decimals` does not require the sequencer to apply it before a later deposit. This matters if, for example, the correction is approved while the asset has zero L2 supply, but a deposit is queued before the correction is consumed.

L1 accepts the deposit as a raw token amount. When the sequencer processes it, the guest converts that amount using the spot configuration then in effect:

```
let gross_amount_e12 =  
    token_amount_to_e12(intent.amount(),  
        spot_config.underlying_erc20_decimals)?;
```

The sequencer can then apply the admin action that changes `underlying_erc20_decimals`. A subsequent withdrawal uses the updated configuration to convert the requested token amount:

```
let amount_e12 = token_amount_to_e12(  
    data.amount_in_underlying,  
    spot_config.underlying_erc20_decimals,  
)?;
```

Impact. If the old decimal setting credits more canonical units than the deposited tokens warrant, the sequencer can inflate the value of one account that it owns by choosing the sequencing order, and can withdraw more tokens than it deposited after the correction.

Client response. Acknowledged. The public and admin queues are independent by design. The client is considering a future hardening by making the decimals an immutable property of the asset configuration.

A deposit above the guest's uint128 range blocks the public queue

OnchainActionManager.sol, onchain_action/deposit.rs

Description. Both L1 deposit functions accept a `uint256` amount and commit all 32 bytes to a public action. The guest's `decode_asset_amount` rejects any amount with nonzero high 128 bits. This decode runs before the fallback branch. Deposits are not skippable public actions, so the same leaf fails on every attempted batch and later public actions cannot be folded into the proven accumulator.

Impact. A listed token that can transfer more than `uint128::MAX` base units can stall the public queue, including later L1 withdrawals. This is reachable if, for example, a malicious actor has unlimited minting power.

Recommendation. Require `amount <= type(uint128).max` on L1 before moving funds or appending the action.

Client response. Fixed in [#2496](#).

Under-received deposits create unbacked protocol balances

OnchainActionManager.sol

Description. `_pullAssetFunds` calls `safeTransferFrom` for the user-supplied amount and does not compare the vault's token balance before and after the transfer. `deposit` and `depositToNewSubaccount` commit that requested amount to the public action. The guest credits it without knowing whether the vault received less because the listed token charges a transfer fee or changes balances during transfer.

Impact. If a fee-on-transfer token is listed, L2 liabilities can exceed the tokens held in its L1 vault.

Recommendation. Reject tokens whose transfers do not deliver the requested amount, or commit and credit the measured amount actually received. Apply the same rule to every supported deposit route.

Client response. Fixed in [#2391](#).

A bucket configured under the native ETH label is not used for native withdrawals

WithdrawalOutbox.sol

Description. `_resolveAsset` maps the `NATIVE_ETH` sentinel to `WRAPPED_ETH_ASSET` for settlement:

```
native = asset == NATIVE_ETH;  
label = native ? WRAPPED_ETH_ASSET : asset;
```

therefore, `_settle` selects a bucket using the resolved label. However, `setBucketLimit` stores the supplied asset label directly:

```
Throttle storage t = _buckets[recipient][asset].throttle;
```

Therefore `setBucketLimit` could be misused: calling `setBucketLimit(recipient, NATIVE_ETH, ...)` may appear like it configures a limit for native ETH withdrawals, but this limit is not applied. The correct way to configure a limit for native ETH withdrawals is to call `setBucketLimit(recipient, WRAPPED_ETH_ASSET, ...)`, which may be less obvious.

Impact. An owner who uses the native ETH sentinel to configure a recipient's limit may believe that limit applies, while withdrawals instead use the wrapped ETH bucket or a fallback bucket.

Recommendation. Resolve the asset label when setting a bucket limit, or require callers to specify `WRAPPED_ETH_ASSET` and reject `NATIVE_ETH`.

Client response. Acknowledged. The client is considering a future hardening by rejecting `NATIVE_ETH` in `setBucketLimit` and requiring `WRAPPED_ETH_ASSET` instead. In the meantime, there is no direct throttling bypass, since if an asset-specific bucket is not configured, the global fallback bucket is used.

A non-spot asset label can block the public action queue

WithdrawalOutbox.sol, OnchainActionManager.sol, deposit/pricing.rs

Description. `WithdrawalOutbox.registerAsset` accepts a protocol asset label without checking its asset type in the rollup.

`OnchainActionManager.deposit` then accepts a deposit under any registered label and commits it to the public action accumulator. If the same label identifies a perp or option on L2, `DepositPricing::new` rejects the action with `NonSpotAssetInBalanceOp`. This check also runs when the listener retries with `fallback = true`. The asset type is fixed by its first L2 registration, so a later registration cannot turn the label into a spot asset.

Impact. Once an owner has mistakenly listed a non-spot label on L1, a small deposit can leave an action that no valid batch can consume. Subsequent public actions, including L1 withdrawal requests, remain blocked behind it. Disabling the L1 listing prevents new deposits but does not remove the existing action.

Recommendation. Coordinate L1 listings with the L2 asset registry and reject non-spot labels before deposits can be accepted. Add a recovery rule for already queued invalid deposits that preserves custody of transferred funds.

Client response. Acknowledged.

Batch proofs are not bound to an EVM deployment

VerifiableApp.sol, batch/codec.rs, WithdrawalOutbox.sol

Description. The guest's public values include neither the `block.chainid` nor the `VerifiableApp` proxy address. If two deployments have the same prestate, both can accept the same proof and the same committed withdrawals. That prestate includes `programVKey`, the old root, the accumulators, and the Celestia predecessor.

Impact. A legitimate batch proof can be replayed against another funded deployment with matching prestate, causing the recipient committed in the proof to receive the same withdrawal from both deployments. The attacker cannot change the recipient or amount.

Recommendation. Commit an EVM domain, including the chain ID and settlement contract address, in the guest's public values and verify it on L1. Treat any cross-deployment state migration as a separate domain transition.

Client response. Acknowledged.

Celestia header verification does not expire the trusted anchor

crates/protocol/celestia-verify/src/verify.rs

Description. `verify_anchored_header` bounds the difference between the target and trusted header timestamps by `TRUSTING_PERIOD_SECS` :

```
let gap = header_time(target)?.saturating_sub(header_time(trusted)?);
if gap > TRUSTING_PERIOD_SECS {
    return Err(DaError::TrustingPeriodExceeded { gap });
}
```

It does not check whether the trusted header itself is still within that period relative to the current time. [Tendermint's light-client specification](#) requires an expired trusted header to lose its trusted status before it is used to verify another header.

There is a check on-chain ensuring that all *new* headers are at most `MAX_DA_HEADER_AGE` seconds old:

```
if (block.timestamp > uint256(da.oldestHeaderTime) + MAX_DA_HEADER_AGE) {
    revert CelestiaHeaderStale(da.oldestHeaderTime,
uint64(block.timestamp));
}
```

However, this check ensures that the anchor is at most `MAX_DA_HEADER_AGE + TRUSTING_PERIOD_SECS` seconds old.

Impact. At the current parameters, `TRUSTING_PERIOD_SECS` is 5 days and `MAX_DA_HEADER_AGE` is 2 days. Currently, the trusting period for Celestia is 7 days, so there is no immediate risk.

Recommendation. Ensure that the sum of `TRUSTING_PERIOD_SECS` and `MAX_DA_HEADER_AGE` is less than the trusting period for Celestia, or add a check that the trusted header is not expired relative to the current time.

Client response. Acknowledged. The current configuration already enforces that the timings are correct, so no change is needed.

Operation timestamps can remain arbitrarily stale

VerifiableApp.sol

Description. `submitBatch` checks that operation timestamps do not go backward and that the maximum is not in the future:

```
if (minOpTimestamp < lastOpTimestamp || maxOpTimestamp > block.timestamp)
{
    revert InvalidOpTimestamp(minOpTimestamp, maxOpTimestamp,
    lastOpTimestamp, block.timestamp);
}
```

The check does not require operation timestamps to advance or remain close to `block.timestamp`. A sequencer can submit successive batches whose operations all carry the same past timestamp.

Impact. If option expiry checks use operation timestamps, this may permit trading options that have expired in real time. That path has not been reviewed here.

Client response. Acknowledged.

Sparse Merkle witness updates rely on a fixed key set

crates/protocol/state/src/sparse_merkle_state/mod.rs, merkel_tree.rs

Description. Sparse Merkle state stores witnessed keys in `BTreeMap`s. The initial proof check authenticates their values against the existing roots. Operations update the maps, and root recomputation reuses the witness nodes:

```
let recomputed = tree.update_root(map.to_merkle_leaves(tag));
```

For this operation to be correct, each map must not insert or delete any key from the map, but can only modify the values of existing keys. Otherwise, the recomputed root may not match the expected root, and the witness nodes may not be consistent with the leaf set.

Impact. No operation that violates the fixed-key-set invariant has been identified. However, for defense-in-depth, the implementation could be hardened to fail on any insertion or deletion of keys.

Client response. Acknowledged.