Security Audit Report

Celestia Q2 2025:
Hana Library

Last Revised
2025/08/04

Authors:
Ivan Gavran, Martin Hutle, Marius Poke, Carlos Rodriguez

Contents

Audit overview

The Project

In June 2025, Celestia engaged Informal Systems↗ to work on a partnership and conduct a security audit of the following items:

- `celestiaorg/hana`↗, branch main, commit `0bfe515`↗

Scope of this Report

Hana library is a rust component used in ZK-proving OP Stack chains through OP Succinct or Kailua. The purpose of Hana is to serve as a library for OP Stack x Celestia chains derivation pipeline. Its crates are used to fetch the data and verify that the data was published to Celestia and properly referenced by a Blobstream deployment.

The audit scope was:

- The hana crates↗, as well as
- The host↗ (excluding `bin/host.rs`)

In particular the host re-uses a lot of features from Kona. The audit was inspecting the Hana code only, assuming the parts that are used from Kona are correct.

Audit Plan

The audit was conducted between June 5th, 2025, and June 24th, 2025 by the following personnel:

- Ivan Gavran
- Martin Hutle
- Marius Poke
- Carlos Rodriguez

Conclusions

The audit was conducted on a stable code version. The audit was performed using a property/threat-based approach, in which the code was fully inspected analyzed along the execution paths.

The code showed a high level of quality and organization. During the audit we identified no major issues, and reported 2 low-severity findings and 3 informational findings.

Audit Dashboard

Target Summary

- **Type:** Implementation
- **Platform:** Rust
- **Artifacts:**
  - `https://github.com/celestiaorg/hana/tree/main/crates`↗
  - `https://github.com/celestiaorg/hana/tree/main/bin/host`↗ (excluding `bin/host.rs`)

Engagement Summary

- **Dates**: 5.6.2025 - 24.6.2025
- **Method**: Manual code review

Severity Summary

Finding Severity	Number
Critical	0
High	0
Medium	0
Low	2
Informational	3
Total	5

System Overview

Celestia Hana is a library that adapts the Optimism (OP) Stack's derivation pipeline to work with Celestia as a modular data availability (DA) layer instead of Ethereum. In Optimistic Rollups, the derivation pipeline is responsible for reconstructing the L2 chain state from posted data. Hana interacts with and implements interfaces of the Kona's `derive` derivation library. Hana enables secure transaction data retrieval from Celestia, ensuring correctness through cryptographic proofs that link Celestia's DA commitments back to Ethereum via the Blobstream bridge.

Following Kona's pipeline structure, Hana integrates DA abstractions, an asynchronous hint system for data prefetching, and cryptographic verification of the obtained data.

Architecture Overview

Main components (under audit)

- **Host**: Retrieves and verifies data from Ethereum and Celestia.
- **Oracle System**: Coordinates data flow between client and host.
- **DA Provider**: Abstracts over Celestia and Ethereum for data retrieval.

External components (out of scope)

- **Client (Kona-based)**: Drives execution of L2 blocks for state verification.
- **Ethereum**: Stores rollup state commitments and frame references.
- **Celestia**: Provides DA for transaction blobs.
- **SP1 Blobstream Contract**: Posts and verifies ZK proofs linking Celestia data roots to Ethereum.

Basic usage

1. A rollup state transition is challenged.
2. The Hana pipeline is instantiated with trusted boot data.
3. The pipeline retrieves transaction data using L1 pointers.
4. Verified data is executed using Kona.
5. A computed output root is compared with the claimed one to resolve the challenge.

Protocol Overview

The protocol logic is triggered during a challenge to a claimed rollup state transition. The Hana pipeline reconstructs the L2 chain state from Celestia blobs and validates it against the claimed output root. The following diagram describes the main steps of the protocol.

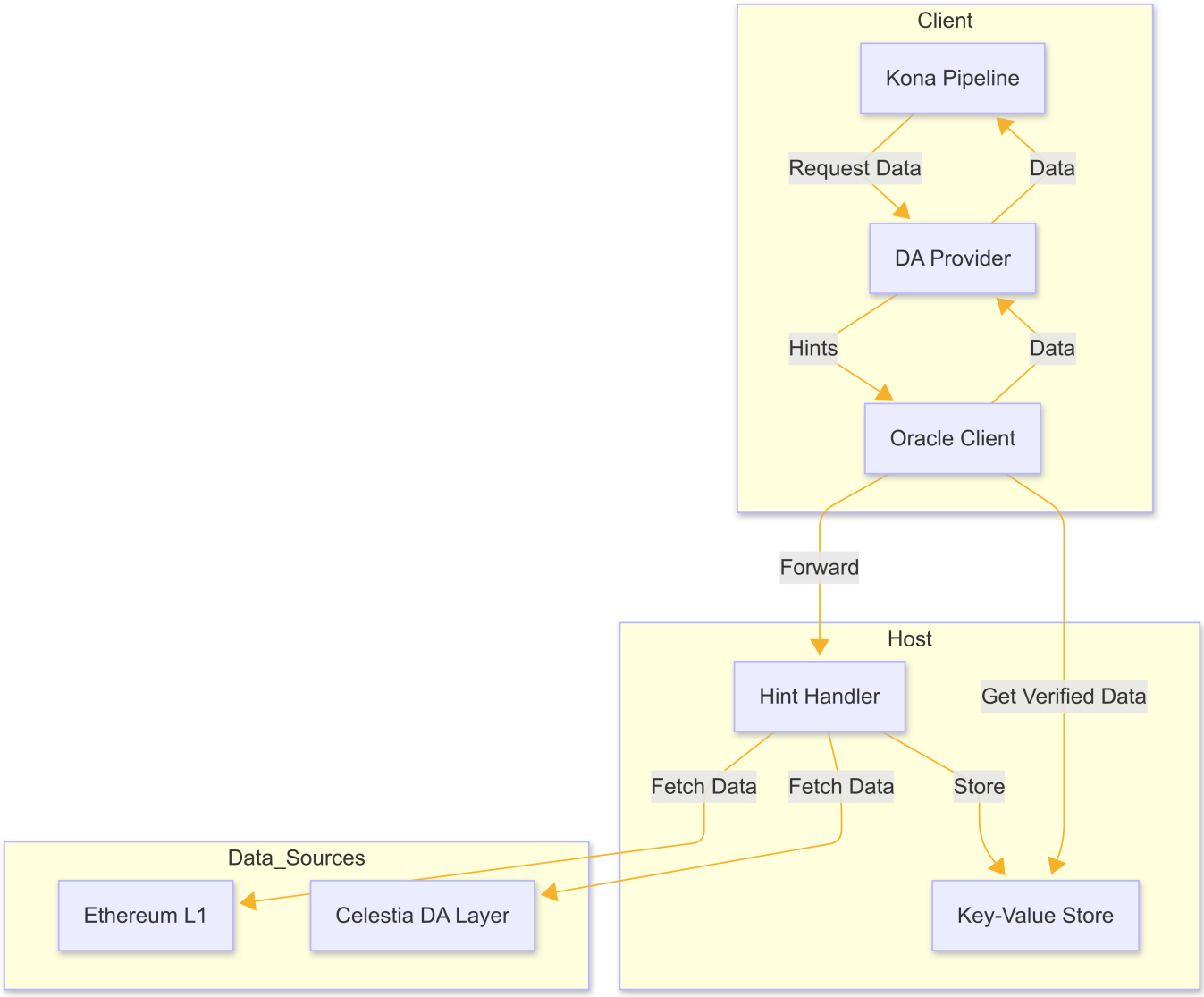

Figure 1: Architecture overview

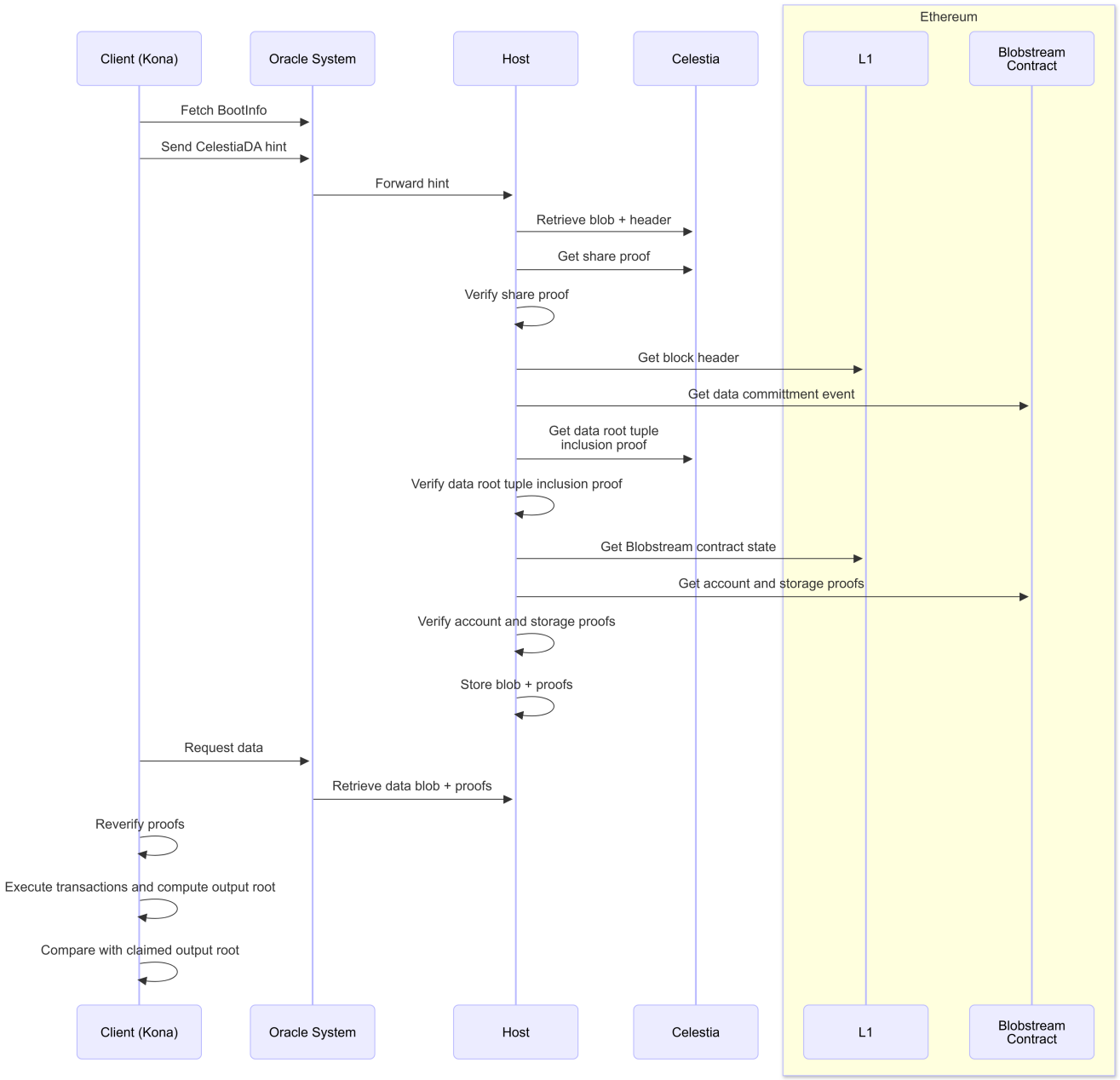

Figure 2: Protocol overview

Threat Model and Inspection Results

Hana implements data fetching for data published to Celestia into the Kona derivation pipeline. Upon receiving next block request from the derivation pipeline, Hana fetches the ***requested*** data ***efficiently***. This combines L1 (Ethereum) and DA (Celestia) sources: when fetching the data, there is no trust given but data's correctness is ***verified*** by Hana. Fetching is done using client and host processes, where the client only ***communicates*** to the host, which then communicates to external sources.

The integration with Kona is done by implementing the function `next` of the Kona's `DataAvailabilityProvider` interface in`CelestiaDADataSource.next(block_ref, batcher_address) → PipelineResult<Bytes>`. This function takes a block reference as `BlockInfo` and an Ethereum address as parameters, and returns the fetched block.

The main property of Hana is that when a request for data arrives, it should deliver correct data back to the pipeline, provided that the data exists. It needs to do so without placing trust into any external entity. In the following list of properties we detail the building blocks of this main property:

Data Blobs Fetching

1. If the next data blob of `block_ref` is a Celestia pointer, the blob where this pointer refers to is returned. Else, the data blob from Ethereum is returned. This disambiguation between when to fetch Celestia's data versus only Ethereum data is done correctly.

    - A blob that is a Celestia pointer has the following structure:

Byte	Description	Value
0	version_byte	0x01
1	commitment_type	0x01
2	da_layer_byte	0x0c
3..10	height (u64 = 8 bytes)	
11..42	commitment (Merkle hash) (32 bytes)	

    - The `version_byte` and the `da_layer_byte` are checked by the implementation. The `commitment_type` is not checked, and all blobs that are not Celestia pointers are deferred to Etherum.
    - Although the Hana library is supposed to be used in a specific context only (namely only Celestia as data source, and only normal Ethereum blobs as fallback) the header should be completely verified and in case of a non-fitting header an error should be raised. See Informational Finding "Blob header not fully verified".
    - Based on the information we obtained we assume that every blob (even if there are several blobs per `block_ref`) is prefixed by the batcher with a correct header (starting with 0 for an Ethereum blob and 0x01 for a Celestia blob).
    - Kona ensures that `next()` is called for the same `block_ref` until an EOF temporary error is thrown and then calls `clear()` to reset before blobs from a new block are requested:
        - Each call to `next_data`↗ will result in an attempt to read data calling `next()`↗. If there is no data to return (`self.data` empty raising a `Temporary` error here↗, propagated in `CelestiaDASource.next`↗, propagate further here↗), a `Temporary` error will be returned, resulting in calling `self.provider.clear()`, which calls `DataAvailabilityProvider`'s `clear`↗.

- The result of calling `CelestiaDASource`'s `clear` will be that↗
  - → `self.data`↗ is emptied
  - → `self.open`↗ is set to false, enabling operations for the new block

2. In subsequent calls of `next()`, no blob is fetched twice.

- In both, the Celestia and Ethereum implementation of `DataAvailabilityProvider`, the flag `self.open` is used to ensure that only in the first call to `next()` (after a `clear()`) the data is loaded (`load_blobs()` return `Ok()` immediately if `open` is true).
- The data is obtained from sending the hint (hash of height and commitment) to the oracle.
- Then `next()` returns always the first element of the vector data, and removes this element. Thus no blob is returned twice.

3. The host correctly interprets what blob was requested by the client.

   Checks:

   1. Correct determination of `height` to fetch from `pointer_data`.
      - Height and commitment are correctly extracted from the `pointer_data`
      - Both are passed to `celestia_source.next` → `load_blobs` → `celestia_fetcher.blob_get`
      - Both are encoded as `Hint` (little-endian height, hash(commitment))
   2. The correct blob is placed in the KV store.
      - The hint handler correctly decodes the hint and fetches the right blob
      - The blob is placed in the KV store with hash(hint)
   3. The client fetches the correct block from the KV store.
      - The client fetches↗ the blob using hash(hint)

Data Verification

1. The correctness of the fetched data is properly verified. That is, the implementation of checks terminates for every input, accepts all correct data blobs, and rejects all incorrect data blobs.

   Checks:

   1. For every data blob, the chain of proofs establishes correctness (share proofs, storage proofs, data root proofs). These proofs are given the parameters corresponding to the relevant block.

      Conclusion: The checks are done both on the host and on the client. The client's checks are performed in the function `blob_get` (← `load_blobs` ← `next`), upon receiving the tuple `(blob, blobstream_proof)` from the oracle (code ref↗). Then the following sequence of verification steps is taken:

      1. `verify_data_commitment`: the Blobstream contract posted the commitment to the L1 state (code ref↗)
         1. the proof's `block_header`'s hash matches the boot-provided `l1_block_hash`
         2. the Blobstream contract exists at the (boot-provided) expected address
         3. the data commitment exists at the expected storage slot
      2. `share_proof.verify`: the data blob was posted on Celestia (code ref↗).
      3. `data_root_tuple_proof.verify`: The data root (on Celestia) was included in the commitment posted by the Blobstream contract (code ref↗).

      Overall, the three verification steps create a chain of reasoning that says 1) The data was really published on Celestia, as witnessed by the Celesetia's header (*data root*); 2) That *data root* is among the ones that the Blobstream contract committed to in its commitment and emitted an event about; 3) This commitment indeed exists in L1 storage at the expected place, posted by the correct Blobstream contract.

The host does the same sequence of checks, but it also retrieves all the proofs for the client and creates a `BlobstreamProof` data structure (code ref↗). The `BlobstreamProof` consists of the following fields:

- `data_root`: a hash of the Celestia's header at a given height (as received in the hint↗), obtained from the Celestia light client (code ref↗ & ref↗).
- `data_commitment`, `proof_nonce`: determined through the `find_data_commitment()` function (code ref↗); see below, for more details on the analysis of this function.
- `data_root_tuple_proof`: a data root tuple inclusion proof obtained from the Celestia light client (code ref↗).
- `share_proof`: a share proof obtained from the Celestia light client (code ref↗). Its parameters (transforming EDS based indices into ODS) were calculated correctly, as established below.
- `storage_root`, `storage_proof`, `account_proof`: data necessary to verify the L1 storage proofs (code ref↗), obtained from the L1 provider, through the `get_proof` method (code ref↗).
- `state_root`, `block_header`: the L1 state root and block header, obtained from the L1 provider through the `get_block_by_hash` method (code ref↗), with the L1 block hash set during the host configuration to the hash of the L1 head block (code ref↗).
- `blobstream_balance`, `blobstream_nonce`, `blobstream_code_hash`: data necessary to construct a `TrieAccount` data structure on the L1, obtained from L1 provider's functions (code ref↗).

2. The location of the share proof is determined↗ correctly.

   Conclusion: We checked the calculations↗ transforming between Original Data Square (ODS) and Extended Data Square (EDS) based indices and they are done correctly. As we suggested in the set of improvements in the finding "Miscellaneous code findings", a separate function would improve the clarity of the calculation.

3. `find_data_commitment()` will locate the data commitment if it exists. If it does not exist, it will raise an error.

   Conclusion: The function iterates backwards from the `l1_head_block_number` parameter to 0 and checks if there is a `DataCommitmentStored` event for which the parameter `celestia_height` is in between its `startBlock` and `endBlock`. It is guaranteed to find such event if it exists between blocks 0 and `l1_head_block_number`. If it does not exist, it is guaranteed to terminate (because↗ `start` decreases, eventually becomes 0, and the function returns at `start == 0`).

   Finally, there is a separate question if the loop will ever reach all the way to 0, which would cause a long outer loop (combined with an inner loop over logs).

   There is a calculation inside the `fetch` function↗, in which it calculates safe L1 head↗ (the L1 block for which the Celestia heights mentioned in the transactions are smaller than the height of the latest Celestia block committed to in the blobstream). Thus calculated `l1_head` is then used as `cfg.l1_head` in the prover, which is the source↗ of the `l1_head_block_number` parameter used in `find_data_commitment()`.

Interfacing with Kona

1. Hana handles well all types of requests that may come, not only Celestia-related.

   The host implements trait `OnlineHostBackendCfg`↗:

```
impl OnlineHostBackendCfg for CelestiaChainHost {
    type HintType = HintWrapper;
    type Providers = CelestiaChainProviders;
}
```

   where `HintType` is defined as `HintWrapper`, an enumeration with the possible types of hint requests that Hana's host can handle:

```rust
pub enum HintWrapper {
    Standard(HintType),
    CelestiaDA,
}
```

where `HintType` is an enumeration of all possible hint types supported by Kona↗.

When the client wants to read data from the host, it first sends a hint request to the host through the hint file descriptor, which signals a request for the host to prepare the data for reading. The host implements the `HintHandler` trait↗, which handles hint requests for both values of the `HintWrapper` enumeration↗.

2. Hana implements correctly all Kona's interfaces, respecting the assumptions on the behaviour of these components, and checking its own assumptions about the inputs.

   Hana's host implements here↗ the `PreImageServerStarter` trait↗:

```rust
#[async_trait]
pub trait PreimageServerStarter {
  async fn start_server<C>(
    &self,
    hint: C,
    preimage: C,
  ) -> Result<JoinHandle<Result<(), SingleChainHostError>>,
 ↪  SingleChainHostError>
  where
    C: Channel + Send + Sync + 'static;
}
```

The `PreimageServerStarter` trait is needed to abstract the process of starting and running the preimage server (i.e., the host process that manages data access through the hint-based system). It standardises how a preimage server is initialised and run with communication channels for hints and preimage requests, a backend for handling those requests (online or offline host), and a key-value storage for the preimage data↗.

Hana's implementation is exactly the same as the one found in Kona for a single chain setup↗.

The `start_server()` function of `PreimageServerStarter` trait takes two parameters:

1. `hint: C`: A channel for hint communication used by the preimage server to receive hints from the client about what data it needs. In the file-based implementation, it uses `FileChannel` with `HintRead` / `HintWrite` descriptors.
2. `preimage: C`: A channel for preimage data communication used to receive preimage requests and send back the requested data. In the file-based implementation, it uses `FileChannel` with `PreimageRead` / `PreimageWrite` descriptors.

Hana implements here↗ the `HintHandler` trait↗:

```rust
/// A [HintHandler] is an interface for receiving hints, fetching remote
 ↪  data, and storing it in the
/// key-value store.
#[async_trait]
pub trait HintHandler {
  /// The type configuration for the [HintHandler].
  type Cfg: OnlineHostBackendCfg;

  /// Fetches data in response to a hint.
  async fn fetch_hint(
    hint: Hint<<Self::Cfg as OnlineHostBackendCfg>::HintType>,
```

```
      cfg: &Self::Cfg,
      providers: &<Self::Cfg as OnlineHostBackendCfg>::Providers,
      kv: SharedKeyValueStore,
    ) -> Result<()>;
  }
```

The `HintHandler` trait is necessary because it serves as an abstraction layer that receives hints from the client about what data it needs, fetches the data from the appropriate sources and processes it into the required format, and makes it available through the preimage store.

Hana implements `CelestiaChainHintHandler` specifically to handle:

- Standard L1/L2 data requests (via `SingleChainHintHandler`)
- Celestia-specific data requests (via `CelestiaDA` hint type)

The `fetch_hint()` function of `HintHandler` takes four parameters:

1. `hint: Hint<Cfg::HintType>`: Contains the type of hint and associated data.
2. `cfg: &Cfg`: Contains configuration data (type `Cfg` is defined as `CelestiaChainHost`).
3. `providers: &Cfg::Providers`: Provides access to different data sources (L1, L2, Celestia).
4. `kv: SharedKeyValueStore`: Used to store the fetched (preimage) data on the correct hash key. The implementation handles concurrent access appropriately.

Hana's host implements here↗ the `OnlineHostBackendCfg` trait↗:

```
/// The [OnlineHostBackendCfg] trait is used to define the type
 ↪ configuration for the
/// [OnlineHostBackend].
pub trait OnlineHostBackendCfg {
  /// The hint type describing the range of hints that can be received.
  type HintType: FromStr<Err = HintParsingError> + Hash + Eq + PartialEq +
   ↪ Clone + Send + Sync;

  /// The providers that are used to fetch data in response to hints.
  type Providers: Send + Sync;
}
```

The `OnlineHostBackendCfg` trait is needed to configure how the `OnlineHostBackend` operates. It defines two associated types:

- `HintType`: Specifies what types of hints the host can handle (in this case `HintWrapper` for both standard and Celestia hints)
- `Providers`: Specifies what data providers are available (in this case `CelestiaChainProviders` combining L1/L2 and Celestia providers)

The host implements it for `CelestiaChainHost` because:

- It needs to extend standard OP Stack functionality with Celestia-specific features
- `HintWrapper` adds Celestia DA hint support alongside standard hints
- `CelestiaChainProviders` adds Celestia node access alongside standard L1/L2 providers

Hana's implementation follows the example of the implementation found in Kona for a single chain setup↗.

Hana's DA provider implements here↗ the `DataAvailabilityProvider` trait↗:

```rust
/// Describes the functionality of a data source that can provide data
 ↪ availability information.
#[async_trait]
pub trait DataAvailabilityProvider {
  /// The item type of the data iterator.
  type Item: Send + Sync + Debug + Into<Bytes>;

  /// Returns the next data for the given [BlockInfo], looking for
   ↪ transactions sent by the
  /// `batcher_addr`. Returns a `PipelineError::Eof` if there is no more data
   ↪ for the given
  /// block ref.
  async fn next(
    &mut self,
    block_ref: &BlockInfo,
    batcher_addr: Address,
  ) -> PipelineResult<Self::Item>;

  /// Clears the data source for the next block ref.
  fn clear(&mut self);
}
```

The `DataAvailabilityProvider` trait is needed to abstract data availability layer functionality. It serves as a high-level interface for data availability operations, sitting above the basic `CelestiaProvider` trait↗ which just handles raw blob retrieval.

The `next()` function of `DataAvailabilityProvider` trait takes two parameters:

1. `block_ref: &BlockInfo`: Is the reference to a pointer stored on L1 with the height of the block that has the transaction data on Celestia and the commitment to verify the inclusion of data.
2. `batcher_address: Address`: Is the Ethereum address of the L1 batch submitter contract responsible for submitting L2 batch data.

3. Hana correctly uses Kona's functions.

   1. `SingleChainHintHandler::fetch_hint`↗: This call processes hints of type `Hint<HintType>` with the following parameters:

      1. `inner_hint`: Constructed from the inner value of `HintWrapper::Standard` type, representing the `HintType`↗ being processed.
      2. `&cfg.single_host.clone()`: A reference to the cloned `single_host` configuration from `CelestiaChainHost` containing the `agreed_l2_head_hash` and `agreed_l2_output_root`.
      3. `&providers.inner_providers`: A reference to the inner providers↗ from the structure of type `CelestiaChainProviders`↗, which supplies the necessary providers for processing the hint.
      4. `kv`: The shared key-value store (`SharedKeyValueStore`) used for storing or retrieving data during hint processing.

      The result of `SingleChainHintHandler::fetch_hint` is matched:

      - Success (`Ok(_)`): No further action is taken.
      - Error (`Err(err)`): An error message is constructed and the process is aborted using `anyhow::bail`.

   2. `next()` on `CelestiaDADataSource`'s `ethereum_source`↗: The `next()` function in `EthereumDataSource` processes data based on the `ecotone_timestamp` configuration, taking the following parameters:

      1. `block_ref`: Represents the reference to the L1 block from which data is retrieved. This L1 block,

when using Celestia as DA, is a pointer to the block on Celestia when L2 transaction data is stored.
2. `batcher_address`: Represents the L1 address of the batcher contract responsible for submitting the data.

Both values are provided by Kona when executing `next()` of `CelestiaDADataSource`. The call is asynchronous and the `.await?` call ensures that the asynchronous operation is completed before proceeding, and it either propagates errors or extracts the successful result. If an error occurs, the function exits early with the error; otherwise, the execution continues with the retrieved

3. `clear()` on `CelestiaDADataSource`'s `ethereum_source`↗: Clears the internal state of both blob and calldata sources. This ensures that the derivation pipeline processes each block independently, avoiding residual data interference from previous blocks. It maintains consistency by discarding stale data, frees up memory for efficient resource usage, and re-initialises the sources to prevent errors during data retrieval.

Host-Client Communication

1. The asynchronous communication between the host and the client ensures safe access to the data.

   Checks:

   1. Access to the KV store is protected for concurrent access
      - The KV store is accessed only by a `RWLock` on the pointer to the store.
   2. The client always gets the data it requested (and not some other data due to e.g., storing under the wrong key or reordering-related problems).
      - See Property 3.
   3. The vector `data` that is used to buffer between `CelestiaDASource.load_blobs` and `CelestiaDASource.next_data` ensures that there is no read from am empty vector.
      - Before reading from the vector, it is checked that the vector is non-empty.

2. The asynchronous communication between the host and the client ensures communication without delays.

   Checks:

   1. No too long/infinite blocks on locks
      - The only lock that is used in Hana is the `RWLock` on the KV store, which is created in `CelestiaChainHost.create_key_value_store` and used in `CelestiaChainHintHandler.fetch_hint`. There, the lock is acquired before writing to the store, and immediately released after that when the associated `RwLockWriteGuard` goes out of scope.
   2. The key-value store used for communication between the client and the host will never get overfilled.
      - There is no explicit clearing of the KV store after a blob has been fetched by hana. However, it is the same mechanism that is used for kona, and the assumption is that Kona implements this correctly for the intended usage of the library.
   3. The vector `data` that is used to buffer between `CelestiaDASource.load_blobs` and `CelestiaDASource.next_data` never gets overfilled.
      - The vector is supposed to hold at most one element, as each celestia pointer refers only to one blob. If next() is invoked two times concurrently, because of the open flag only the first call to load_blobs will push an element to the vector.

3. The client and the host process are correctly started and are able to communicate with each other.

   - The code creates a `PreimageServer` with a `OracleServer` and `HintReader` and a backend (`OnlineHostBackend` or `OfflineHostBackend`).

- If it is a server only, a `FileChannel` is used to read hints and write preimages.
- In case of native mode, also a client task is started and the communication is with `BidirectionalChannels`.
- Note: All the referenced code is from Kona.

Non-functional Properties

1. All panics should be justified (panic only when it is an unrecoverable problem)

   We believe there are instances when panics are unjustified (see the "Unnecessary panic upon failed verification" finding)

2. All errors should be properly handled/propagated. There is one place where the error is not properly propagated, upon loading blobs (see the "Error when loading blobs not propagated" finding). However, this does not cause security concern.

3. No data that is coming from an external source (call parameter, chain data) or unverified internal source can lead to excessive space allocations or excessive computation. The only relevant place in the code where an excessive computation may be triggered is `find_data_commitment()` ↗ function. However, its parameters are sanitized at the op-succinct's side, in the function `calculate_safe_l1_head()` ↗.

Findings

Finding	Type	Severity	Status
Error when loading blobs not propagated	Implementation	Low	Resolved
Unnecessary panic upon failed verification	Implementation	Low	Resolved
DA Provider for Celestia Data Source mimics DA Provider for Ethereum Data Source too literally	Implementation	Informational	Resolved
Blob header not fully verified	Implementation	Informational	Resolved
Miscellaneous code findings	Implementation	Informational	Resolved

Error when loading blobs not propagated

Severity Low **Impact** 1 - Low **Exploitability** 1 - Low

Type Implementation **Status** Resolved

Involved artifacts

- `crates/celestia/src/source.rs`↗

Description

In the function `load_blobs()`, upon an error↗ from `self.celestia_fetcher.blob_get()`, the error is not propagated. This causes the calling function, `CelestiaDASource`'s `next()`↗ not to return early, but instead go into reading `next_data`↗. If `self.data` is empty (as it should be whenever there was an error in `blob_get()`), a `PipelineErrorKind::Temporary` is returned.

Problem scenarios

Such handling of errors masks other kinds of errors that may occur in the `load_blobs()` function.

Recommendation

At the place where an error occurs, distinguish between `Temporary` and other kind of errors and propagate the error immediately.

Status

Resolved↗.

Unnecessary panic upon failed verification

Severity Low **Impact** 1 - Low **Exploitability** 1 - Low

Type Implementation **Status** Resolved

Involved artifacts

- `hana/crates/orcle/src/provider.rs`↗
- `hana/crates/proofs/src/blobstream_inclusion.rs`↗

Description

At a number of places when the data commitment and availability are supposed to be verified, a panic is raised upon verification failure.

Concretely:

- data commitment verification↗ in `blob_get()` (client)
- data root tuple verification↗ in `blob_get()` (client)
- share proof verification↗ in `get_blobstream_proof()` (host)
- data root tuple verification↗ in `get_blobstream_proof()` (host)

Conversely, the verification failures at other places was handled by propagating an error:

- share proof verification↗ in `blob_get()` (client)
- data commitment verification↗ in `get_blobstream_proof()` (host)

Recommendation

To our understanding, a verification failure should be a recoverable failure, and thus an error should be returned consistently. If the trust between the host and the client is complete, perhaps it makes sense to panic upon verification failures on the client (because this suggests something if completely off, since the host should have done the verification already).

Status

Resolved↗.

DA Provider for Celestia Data Source mimics DA Provider for Ethereum Data Source too literally

Severity Informational **Impact** 1 - Low **Exploitability** 0 - None

Type Implementation **Status** Resolved

Involved artifacts

- `hana/crates/celestia/src/celestia.rs`↗
- `hana/crates/celestia/src/source.rs`↗
- `kona/crates/protocol/derive/src/sources/blobs.rs`↗

Description

In its design, the implementation↗ of `CelestiaDASource` closely follows the design of `BlobSource`↗ and its data source.

In particular, the implementation of `DataAvailabilityProvider` for `BlobSource` maintains a vector `self.data`↗ and a boolean flag `self.open`↗. The flag is set↗ to `true` whenever data is written into `self.data`. While the flag is set, there can be no writing↗ into `self.data`. The function `self.next_data()` reads one after another↗ element from `self.data`. When there are no more elements in `self.data`, it returns an error↗. Upon receiving this error, `self.clear` is called↗, which resets the flag↗ `self.open` to `false`.

`CelestiaDASource` follows exactly the same pattern. However, in the context of Celestia, `self.data` will only ever have a single element. This makes the above mechanics unnecessary:

- There is no need for a data vector, a single element would suffice.
- There is no need for a flag `self.open`: each write is immediately followed by a read that exhausts all of data.

Problem scenarios

According to our inspection, closely following the pattern did not result in any security issues or serious inefficiencies. However, it creates false expectations, hindering maintainability.

Recommendation

We suggest being explicit about the expectations and implement member functions `next`, `next_data`, and `clear` without following by a letter the corresponding implementations for blobs.

Status

Resolved↗.

Blob header not fully verified

Severity Informational Impact: Exploitability:

Type Implementation **Status** Resolved

Involved artifacts

- `crates/celestia/src/celestia.rs`↗, function `next()`↗

Description

According to the OP stack specification, the first 1-3 bytes of a blob identify the type of blob.

version_byte	commitment_type	da_layer_byte	payload
0			frames
1	0		keccak_commitment
1	1	0	eigenda_commitment
1	1	0x0a	avail_commitment
1	1	0x0c	celestia_commitment
1	1	...	altda_commitment

The current implementation of Hana only checks the first and third byte, forwarding everything else to the Ethereum data provider. While this is ok under the assumption that the Hana library is used only in scenarios where blobs are either Celestia pointers or Ethereum blobs as fallback, for the propose of robustness the header should be completely checked and an error raised in case of unsupported blobs.

Problem scenarios

- Unindented usage of the library
- Later code changes

Recommendation

- Check for 0x01 0x01 0x0c for a Celestia pointer, else check for 0x00 for a standard blob, else return an error.

Status

Resolved↗.

Miscellaneous code findings

Severity Informational **Impact** 1 - Low **Exploitability** 0 - None

Type Implementation **Status** Resolved

In this finding, we describe a number of improvements to the code. Those typically do not affect the functionality or security, but improve the code readability, make code more robust with respect to future changes, or represent a good engineering practice.

1. The code part↗ in which `start_index` and `end_index` for obtaining the share proof are calculated should be factored out into a separate function. Since the calculation is not straightforward to understand, this will help clarity and maintainability.
2. Instead of repeated `unwraps` of `blob.index` here↗, better unwrap once with explicit error handling.
3. The `state_root` field in `BlobstreamProof` is never used as it's retrieved directly from the L1 block header (code ref↗).
4. The function `find_data_commitment()` relies on the fact that `l1_head_block_number` parameter will be such that there would be an earlier block that contains the `celestia_height` among its `DataCommitmentStored` logs. (In the overall system this is made sure by the `op-succinct`'s function `calculate_safe_l1_head()`↗. However, it would be useful if this assumption would be explicitly mentioned for the function (in case it gets used in a different context).
5. Typo `blostream_address`↗ instead of `blobstream_address`.

Status

Resolved↗.

Appendix: Vulnerability classification

For classifying vulnerabilities identified in the findings of this report, we employ the simplified version of Common Vulnerability Scoring System (CVSS) v3.1↗, which is an industry standard vulnerability metric. For each identified vulnerability we assess the scores from the *Base Metric Group*, the Impact score↗, and the Exploitability score↗. The *Exploitability score* reflects the ease and technical means by which the vulnerability can be exploited. That is, it represents characteristics of the *thing that is vulnerable*, which we refer to formally as the *vulnerable component*. The *Impact score* reflects the direct consequence of a successful exploit, and represents the consequence to the *thing that suffers the impact*, which we refer to formally as the *impacted component*. In order to ease score understanding, we employ CVSS Qualitative Severity Rating Scale↗, and abstract numerical scores into the textual representation; we construct the final *Severity score* based on the combination of the Impact and Exploitability sub-scores.

As blockchains are a fast evolving field, we evaluate the scores not only for the present state of the system, but also for the state that deems achievable within 1 year of projected system evolution. E.g., if at present the system interacts with 1-2 other blockchains, but plans to expand interaction to 10-20 within the next year, we evaluate the impact, exploitability, and severity scores wrt. the latter state, in order to give the system designers better understanding of the vulnerabilities that need to be addressed in the near future.

Impact Score

The Impact score captures the effects of a successfully exploited vulnerability on the component that suffers the worst outcome that is most directly and predictably associated with the attack.

ImpactScore	Examples
High	Halting of the chain; loss, locking, or unauthorized withdrawal of funds of many users; arbitrary transaction execution; forging of user messages / circumvention of authorization logic
Medium	Temporary denial of service / substantial unexpected delays in processing user requests (e.g. many hours/days); loss, locking, or unauthorized withdrawal of funds of a single user / few users; failures during transaction execution (e.g. out of gas errors); substantial increase in node computational requirements (e.g. 10x)
Low	Transient unexpected delays in processing user requests (e.g. minutes/a few hours); Medium increase in node computational requirements (e.g. 2x); any kind of problem that affects end users, but can be repaired by manual intervention (e.g. a special transaction)

ImpactScore	Examples
None	Small increase in node computational requirements (e.g. 20%); code inefficiencies; bad code practices; lack/incompleteness of tests; lack/incompleteness of documentation

Exploitability Score

The Exploitability score reflects the ease and technical means by which the vulnerability can be exploited; it represents the characteristics of the vulnerable component. In the below table we list, for each category, examples of actions by actors that are enough to trigger the exploit. In the examples below:

- *Actors* can be any entity that interacts with the system: other blockchains, system users, validators, relayers, but also uncontrollable phenomena (e.g. network delays or partitions).
- *Actions* can be
  - *legitimate*, e.g. submission of a transaction that follows protocol rules by a user; delegation/redelegation/bonding/unbonding; validator downtime; validator voting on a single, but alternative block; delays in relaying certain messages, or speeding up relaying other messages;
  - *illegitimate*, e.g. submission of a specially crafted transaction (not following the protocol, or e.g. with large/incorrect values); voting on two different alternative blocks; alteration of relayed messages.
- We employ also a *qualitative measure* representing the amount of certain class of power (e.g. possessed tokens, validator power, relayed messages): *small* for < 3%; *medium* for 3-10%; *large* for 10-33%, *all* for >33%. We further quantify this qualitative measure as relative to the largest of the system components. (e.g. when two blockchains are interacting, one with a large capitalization, and another with a small capitalization, we employ *small* wrt. the number of tokens held, if it is small wrt. the large blockchain, even if it is large wrt. the small blockchain)

ExploitabilityScore	Examples
High	illegitimate actions taken by a small group of actors; possibly coordinated with legitimate actions taken by a medium group of actors
Medium	illegitimate actions taken by a medium group of actors; possibly coordinated with legitimate actions taken by a large group of actors
Low	illegitimate actions taken by a large group of actors; possibly coordinated with legitimate actions taken by all actors
None	illegitimate actions taken in a coordinated fashion by all actors

Severity Score

The severity score combines the above two sub-scores into a single value, and roughly represents the probability of the system suffering a severe impact with time; thus it also represents the measure of the urgency or order in which vulnerabilities need to be addressed. We assess the severity according to the combination scheme represented graphically below.

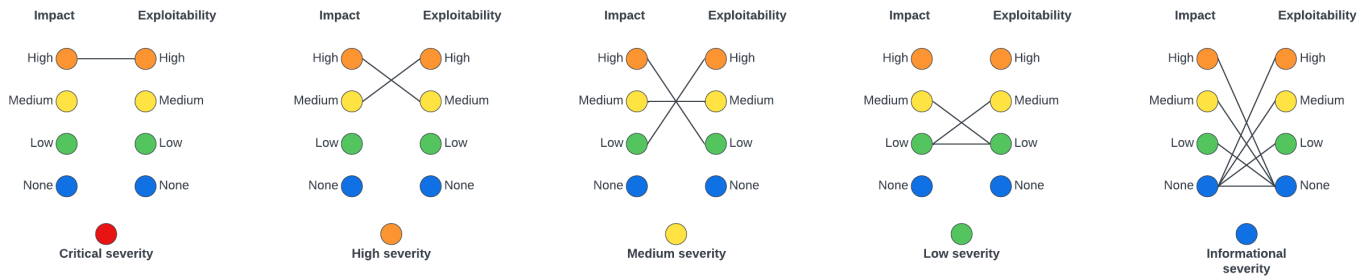

Figure 3: Severity classification

As can be seen from the image above, only a combination of high impact with high exploitability results in a Critical severity score; such vulnerabilities need to be addressed ASAP. Accordingly, High severity score receive vulnerabilities with the combination of high impact and medium exploitability, or medium impact, but high exploitability.

SeverityScore	Examples
Critical	Halting of chain via a submission of a specially crafted transaction
High	Permanent loss of user funds via a combination of submitting a specially crafted transaction with delaying of certain messages by a large portion of relayers
Medium	Substantial unexpected delays in processing user requests via a combination of delaying of certain messages by a large group of relayers with coordinated withdrawal of funds by a large group of users
Low	2x increase in node computational requirements via coordinated withdrawal of all user tokens
Informational	Code inefficiencies; bad code practices; lack/incompleteness of tests; lack/incompleteness of documentation; any exploit for which a coordinated illegitimate action of all actors is necessary

Disclaimer

This report is subject to the terms and conditions (including without limitation, description of services, confidentiality, disclaimer and limitation of liability, etc.) set forth in the associated Services Agreement. This report provided in connection with the Services set forth in the Services Agreement shall be used by the Company only to the extent permitted under the terms and conditions set forth in the Agreement.

This audit report is provided on an "as is" basis, with no guarantee of the completeness, accuracy, timeliness or of the results obtained by use of the information provided. Informal has relied upon information and data provided by the client, and is not responsible for any errors or omissions in such information and data or results obtained from the use of that information or conclusions in this report. Informal makes no warranty of any kind, express or implied, regarding the accuracy, adequacy, validity, reliability, availability or completeness of this report. This report should not be considered or utilized as a complete assessment of the overall utility, security or bugfree status of the code.

This audit report contains confidential information and is only intended for use by the client. Reuse or republication of the audit report other than as authorized by the client is prohibited.

This report is not, nor should it be considered, an "endorsement", "approval" or "disapproval" of any particular project or team. This report is not, nor should it be considered, an indication of the economics or value of any "product" or "asset" created by any team or project that contracts with Informal to perform a security assessment. This report does not provide any warranty or guarantee regarding the absolute bug-free nature of the technology analyzed, nor does it provide any indication of the client's business, business model or legal compliance. This report should not be used in any way to make decisions around investment or involvement with any particular project. This report in no way provides investment advice, nor should it be leveraged as investment advice of any sort.

Blockchain technology and cryptographic assets in general and by definition present a high level of ongoing risk. Client is responsible for its own due diligence and continuing security in this regard.