Veridise

Auditing Report

Hardening Blockchain Security with Formal Methods

FOR

Hokulea

Veridise Inc.
October 2, 2025

► **Prepared For:**

EigenDA
https://www.eigenda.xyz/

► **Prepared By:**

Jon Stephens
Tyler Diamond

► **Contact Us:**

contact@veridise.com

► **Version History:**

Nov. 5, 2025	V2
Oct. 27, 2025	V1
Oct. 23, 2025	Initial Draft

Contents

1 Executive Summary

From Oct. 2, 2025 to Oct. 19, 2025, EigenDA engaged Veridise to conduct a security assessment of their Hokulea library. The security assessment covered the Hokulea library which integrates EigenDA with Optimism's Kona library, and Canoe — the zkVM application to prove the validity of EigenDA certs by calling on-chain verification contracts. Veridise conducted the assessment over 4 person-weeks, with 2 security analysts reviewing the project over 2 weeks on commit `3e61825`. The review strategy involved a tool-assisted analysis of the program source code performed by Veridise security analysts as well as thorough code review.

Project Summary. The security assessment covered the Hokulea library which extends Optimism's Kona library to interact with EigenDA and provide utilities to integrate EigenDA as an alternative Data Availability Provider (DAP) in OP-Stack rollups.

OP-Stack rollups provide the functionality for a layer-2 (L2) network to anchor their state to the base layer-1 (L1), which is usually Ethereum. Specifically, observers of the L1 chain should be able to deterministically construct the state of the L2 by only observing data that is provided to a smart contract on the L1 (although the data that is provided may eventually expire). This smart contract is referred to as the *inbox address*. Optimism, which is an optimistic rollup, uses dispute games to resolve disputes against the claimed state root of the L2. This is done by running the derivation process from L1 data in the Fault Proof VM (FPVM). The program ran in the FPVM will produce the source of truth for deriving the output root.

As mentioned in the previous paragraph, it is crucial for the data used by the FPVM to be available to the public in order to carry out the dispute process. In normal Optimism chains, this data is either Ethereum calldata or EIP-4844 blobs. Hokulea provides the utilities for integrating EigenDA as a data availability provider. At a high-level, the EigenDA network contains a committee that attests to the availability of a certain piece of data. This committee will create a signed `AltDACommitment` that is submitted to the inbox address.

The project has two main parts. The first part is named Canoe. This utilizes either the SP1-cc or RiscZero Steel to produce a zkVM proof of execution against the smart contract responsible for validating the legitimacy of `AltDACommitments` in order to determine canonical pieces of data.

The second part of the project are the utilities created for retrieving data from EigenDA and creating immutable data sources to use in zkVMs or FPVMs. When properly used, the data source will validate the Canoe zk proof and provide the recency and validity of the `AltDACommitments`, along with the corresponding KZG-verified payloads that the `AltDACommitments` are attesting to. These data structures then adhere to Kona's traits for integrating into the derivation process, and therefore users can write an FPVM or zkVM program to use as the dispute game source of truth.

Code Assessment. The Hokulea developers provided the source code of the Hokulea library for the code review. The library consists of original code written by EigenDA developers. The code interacts with other libraries such as Kona, Steel, SP1-cc and custom EigenDA libraries to implement its behaviors. It contains documentation about EigenDA itself, describing how to safely interact with the DA layer, some examples and documentation in the form of READMEs and in-line documentation explaining parts of the source code. To facilitate the Veridise security

analysts' understanding of the code, the Hokulea developers provided an initial walk-through of the project and code, documentation regarding how to interact with EigenDA and an example project that interacts with the library to demonstrate how users are expected to interact with Hokulea.

The source code contained a test suite, which the Veridise security analysts noted contained both positive and negative test cases for the important functionality in the library. The code also contained examples to demonstrate how to use the library end-to-end. The developers also shared a Hokulea integration in the op-succinct repository to demonstrate how Hokulea integrates with zkVM applications.

Summary of Issues Detected. The security assessment uncovered 13 issues, 3 of which are assessed to be of high or critical severity by the Veridise analysts. Specifically, V-EIG-VUL-001 identifies a missing length check that allows mutated blobs to pass KZG verification and potentially allowing mutated transaction data to be accepted. V-EIG-VUL-002 identifies how the EVM environment in the Canoe SP1 zkVM application can be manipulated to compromise the validity of cert validity proofs. Finally, V-EIG-VUL-003 describes that under certain circumstances, the EVM environment in the Canoe RiscZero zkVM application can be manipulated to compromise the validity of cert validity proofs. The Veridise analysts also identified 2 medium-severity issues that identifies a deviation from the Kona specification, potentially allowing for the creation of forks due to discrepancies between Kona and Hokulea (V-EIG-VUL-004). In addition to these findings, 4 low-severity issues and 4 warnings. The develoeprs have been notified of these findings and have addressed all of the reported issues.

Recommendations. After conducting the assessment of the protocol, the security analysts had a few suggestions to improve the Hokulea library.

Improve Hokulea Documentation. During the course of the review, Veridise analysts found little documentation regarding how users are intended to interact with Hokulea even though the library makes assumptions regarding the integration as discussed in issue V-EIG-VUL-012. We would therefore strongly recommend including documentation regarding the creation of secure integrations with the Hokulea library and ensure any examples are consistent with respect to this documentation.

Disclaimer. We hope that this report is informative but provide no warranty of any kind, explicit or implied. The contents of this report should not be construed as a complete guarantee that the system is secure in all dimensions. In no event shall Veridise or any of its employees be liable for any claim, damages or other liability, whether in an action of contract, tort or otherwise, arising from, out of or in connection with the results reported here.

2 Project Dashboard

Table 2.1: Application Summary.

Name	Version	Type	Platform
Hokulea	`3e61825`	Rust	SP1, RiscZero

Table 2.2: Engagement Summary.

Dates	Method	Consultants Engaged	Level of Effort
Oct. 2–Oct. 19, 2025	Manual	2	4 person-weeks

Table 2.3: Vulnerability Summary.

Name	Number	Acknowledged	Fixed
Critical-Severity Issues	2	2	2
High-Severity Issues	1	1	1
Medium-Severity Issues	2	2	2
Low-Severity Issues	4	4	4
Warning-Severity Issues	4	4	4
Informational-Severity Issues	0	0	0
TOTAL	13	13	13

Table 2.4: Category Breakdown.

Name	Number
Data Validation	7
Maintainability	3
Logic Error	2
Usability Issue	1

3 Security Assessment Goals and Scope

3.1 Security Assessment Goals

The engagement was scoped to provide a security assessment of Hokulea's source code. During the assessment, the security analysts aimed to answer questions such as:

- Will the canoe zkVM applications's execution be consistent with performing a query on mainnet?
- Is the data committed by the canoe zkVM application sufficient to validate the execution of the call?
- Is the verification of a canoe proof sufficient to establish the validity of a cert?
- Is the validation performed by the `PreloadedEigenDAPreimageProvider` sufficient to establish the correctness of the preloaded information?
- Will the EigenDA additions modify the intended behavior of the Kona library?
- Is it possible for malicious EigenDA inputs to fork an optimism-based rollup by presenting multiple valid interpretations of the data?
- Does Hokulea enforce all requirements from EigenDA to ensure a submitted blob is valid?

3.2 Security Assessment Methodology & Scope

Security Assessment Methodology. To address the questions above, the security assessment involved a manual review by human experts.

Scope. The scope of this security assessment is limited to the non-test and non-build rust code in the `bin`, `canoe` and `crates` directories of the source code provided by the Hokulea developers.

Methodology. The Veridise security analysts reviewed the provided documentation and then inspected the Hokulea tests and examples. They then began a manual review of the provided source code.

During the security assessment, the Veridise security analysts regularly met with the Hokulea developers to ask questions about the code and report new findings. The Veridise analysts also discussed potential findings with developers throughout the audit via AuditHub.

3.3 Classification of Vulnerabilities

When Veridise security analysts discover a possible security vulnerability, they must estimate its severity by weighing its potential impact against the likelihood that a problem will arise.

The severity of a vulnerability is evaluated according to the Table 3.1.

The likelihood of a vulnerability is evaluated according to the Table 3.2.

The impact of a vulnerability is evaluated according to the Table 3.3:

Table 3.1: Severity Breakdown.

	Somewhat Bad	Bad	Very Bad	Protocol Breaking
Not Likely	Info	Warning	Low	Medium
Likely	Warning	Low	Medium	High
Very Likely	Low	Medium	High	Critical

Table 3.2: Likelihood Breakdown

Not Likely	A small set of users must make a specific mistake
Likely	Requires a complex series of steps by almost any user(s) - OR - Requires a small set of users to perform an action
Very Likely	Can be easily performed by almost anyone

Table 3.3: Impact Breakdown

Somewhat Bad	Inconveniences a small number of users and can be fixed by the user
Bad	Affects a large number of people and can be fixed by the user - OR - Affects a very small number of people and requires aid to fix
Very Bad	Affects a large number of people and requires aid to fix - OR - Disrupts the intended behavior of the protocol for a small group of users through no fault of their own
Protocol Breaking	Disrupts the intended behavior of the protocol for a large group of users through no fault of their own

4 Trust Model

4.1 Operational Assumptions

In addition to assuming that any out-of-scope components behave correctly, Veridise analysts assumed the following properties held when modeling security for Hokulea.

- The Canoe zkVM Image IDs are set by a trusted entity and correspond to elf executables compiled from the canoe source code reviewed.
- The `CanoeVerifierAddressFetcher` has been configured to reference the proper EigenDA verification contracts by a trusted entity.
- The EigenDA proxy is a trusted entity and will serve correct information with respect to the cert provided. This means that fetched blobs have already been verified with respect to a cert's KZG commitment, and that the provided recency and validity information is accurate.
- The KZG parameters used are from a trusted setup that is not compromised.
- The project using Hokulea is configured to reference a trusted proxy.
- Data in the Kona oracle is trusted.

5 Vulnerability Report

This section presents the vulnerabilities found during the security assessment. For each issue found, the type of the issue, its severity, location in the code base, and its current status (i.e., acknowledged, fixed, etc.) is specified. Table 5.1 summarizes the issues discovered:

Table 5.1: Summary of Discovered Vulnerabilities.

ID	Description	Severity	Status
V-EIG-VUL-001	Missing length check allows blob . . .	Critical	Fixed
V-EIG-VUL-002	Sp1 canoe zkVM application may use . . .	Critical	Fixed
V-EIG-VUL-003	Risc0 Canoe zkVM application may use . . .	High	Fixed
V-EIG-VUL-004	Version byte check incorrectly parses data	Medium	Fixed
V-EIG-VUL-005	Eager data processing can change error . . .	Medium	Fixed
V-EIG-VUL-006	Recency disagreements break determinism	Low	Fixed
V-EIG-VUL-007	AnchorHash assumed to be block hash	Low	Fixed
V-EIG-VUL-008	SP1-cc verification is skipped outside of . . .	Low	Fixed
V-EIG-VUL-009	Uninitialized information in structs	Low	Fixed
V-EIG-VUL-010	Maintainability Improvements	Warning	Fixed
V-EIG-VUL-011	Improperly formatted EncodedPayloads . . .	Warning	Fixed
V-EIG-VUL-012	Integration requirements should be . . .	Warning	Fixed
V-EIG-VUL-013	Defensive programming improvements	Warning	Fixed

5.1 Detailed Description of Issues

5.1.1 V-EIG-VUL-001: Missing length check allows blob manipulation

Severity	Critical	**Commit**	3e61825
Type	Data Validation	**Status**	Fixed
Location(s)	`crates/proof/src/preloaded_eigenda_provider.rs:56-123`		
Confirmed Fix At	hokulea/pull/210/..2701eb7e		

Description The function `from_witness` accepts a list of encoded data blobs and associated KZG proofs, verifies them in a batch, and constructs a `PreloadedEigenDAPreimageProvider`. However, it lacks an explicit length check to validate that each blob matches the expected size declared in its corresponding certificate. Note that under EIP-4844, such checks are performed during the KZG verification of a blob by checking that the blob has a fixed size (4096 field elements). Since EigenDA supports variable-length blobs, such a check is not performed in the KZG verification procedure, requiring external validation.

Due to the missing length checks, it is possible to construct a larger-than-expected blob that matches the original polynomial commitment by evaluating the committed polynomial at additional points. This results in multiple valid blob encodings for a single commitment, opening the door to data manipulation.

For example, an attacker could construct a larger blob from a degree-n polynomial and inject controlled or semi-controlled values at extra positions. Due to how the utilized KZG verification library constructs the roots of unity, this larger blob would not just contain additional data after the original blob's content, but rather additional content would be interspersed with the original blob's data. As such, the blob's prefix associated with the encoded payload would be manipulated but such manipulations would go undetected.

Impact A malicious actor is able to manipulate the content of a blob while maintaining the validity of the expected KZG commitment. More specifically, a truncated or extended blob could be constructed by a malicious actor that would pass the KZG commitment validation.

In the case of truncation (e.g. if a blob is all 0's I can construct a shorter blob of all 0's), this would eventually result in an error being thrown by later EigenDA validation and the associated block would be skipped. In this case, the computation of the "valid" state would not be deterministic as two states (one with the manipulated blob and one with the real blob) would appear equally valid.

In the case of extension (e.g. if a blob is appended with additional data), this would result in the eventual acceptance of the extended blob by EigenDA's and Kona's logic. While only a subset of a blob contains the actual data, a blob extension also results in a rotation due to how a library computes the roots of unity and so any prefix of the manipulated blob will not necessarily be equal to the original blob. This would eventually allow manipulated data to be accepted by Kona.

Recommendation Introduce an explicit length check between the blob and the expected length specified in the DA certificate when checking the validity of the blob's data. To prevent similar mistakes, we would recommend adding this check to EigenDA's KZG verification API as well.

Proof of Concept Below a proof of concept can be found to show how multiple blobs can be accepted by a single commitment.

```
diff --git a/rust-toolchain b/rust-toolchain
index 142f153..cbdb9fc 100644
--- a/rust-toolchain
+++ b/rust-toolchain
@@ -1,5 +1,5 @@
 [toolchain]
-channel = '1.80'
+channel = '1.81'
 profile = 'minimal'
 components = ['clippy', 'rustfmt']
 targets = ["x86_64-unknown-linux-gnu", "aarch64-apple-darwin", "i686-unknown-linux-gnu"]
diff --git a/verifier/Cargo.toml b/verifier/Cargo.toml
index 7c8a41e..d5de04e 100644
--- a/verifier/Cargo.toml
+++ b/verifier/Cargo.toml
@@ -21,6 +21,7 @@ criterion = "0.5"
 lazy_static = "1.5"
 ark-std = { workspace = true, features = ["parallel"] }
 rust-kzg-bn254-prover.workspace = true
+ark-poly.workspace = true

 [[bench]]
 name = "bench_kzg_verify"
diff --git a/verifier/tests/tests.rs b/verifier/tests/tests.rs
index 806885c..2c92384 100644
--- a/verifier/tests/tests.rs
+++ b/verifier/tests/tests.rs
@@ -1,13 +1,17 @@
 #[cfg(test)]
 mod tests {
-    use ark_bn254::{Fq, G1Affine};
+    use ark_bn254::{Fq, Fr, G1Affine};
     use ark_ec::AffineRepr;
-    use ark_ff::UniformRand;
+    use ark_ff::{UniformRand, Zero};
+    use ark_poly::{EvaluationDomain, GeneralEvaluationDomain};
     use lazy_static::lazy_static;
     use rand::Rng;
     const GETTYSBURG_ADDRESS_BYTES: &[u8] = "Fourscore and seven years ago our fathers brought forth, on this continent, a new nation, conceived in liberty, and dedicated to the proposition that all men are created equal. Now we are engaged in a great civil war, testing whether that nation, or any nation so conceived, and so dedicated, can long endure. We are met on a great battle-field of that war. We have come to dedicate a portion of that field, as a final resting-place for those who here gave their lives, that that nation might live. It is altogether fitting and proper that we should do this. But, in a larger sense, we cannot dedicate, we cannot consecrate-we cannot hallow-this ground. The brave men, living and dead, who struggled here, have consecrated it far above our poor power to add or detract. The world will little note, nor long remember what we say here, but it can never forget what they did here. It is for us the living, rather, to be dedicated here to the unfinished work which they who fought here have thus far so nobly advanced. It is rather for us to be here dedicated to the great task remaining before us-that from these honored dead we take increased devotion
```

```
     to that cause for which they here gave the last full measure of devotion-that we
     here highly resolve that these dead shall not have died in vain-that this nation
    , under God, shall have a new birth of freedom, and that government of the people
    , by the people, for the people, shall not perish from the earth.".as_bytes();
40       use ark_std::{str::FromStr, One};
41  -    use rust_kzg_bn254_primitives::blob::Blob;
42  +    use rust_kzg_bn254_primitives::{
43  +        blob::Blob,
44  +        polynomial::{PolynomialCoeffForm, PolynomialEvalForm},
45  +    };
46       use rust_kzg_bn254_prover::{kzg::KZG, srs::SRS};
47       use rust_kzg_bn254_verifier::{
48           batch::verify_blob_kzg_proof_batch,
49  @@ -379,6 +383,66 @@ mod tests {
50           }
51       }
52
53  +    #[test]
54  +    fn test_single_commitment_valid_for_multiple_blobs() {
55  +        let mut kzg = KZG_INSTANCE.clone();
56  +
57  +        let polynomial = PolynomialCoeffForm::new(vec![
58  +            Fr::from(11u64),
59  +            Fr::from(3u64),
60  +            Fr::from(7u64),
61  +        ]);
62  +
63  +        let domain_small = GeneralEvaluationDomain::<Fr>::new(4).unwrap();
64  +        let evals_small = domain_small.fft(polynomial.coeffs());
65  +        let eval_poly_small = PolynomialEvalForm::new(evals_small.clone());
66  +        let blob_small = Blob::new(&eval_poly_small.to_bytes_be()).unwrap();
67  +
68  +        let mut coeffs_large = polynomial.coeffs().to_vec();
69  +        coeffs_large.resize(8, Fr::zero());
70  +        let domain_large = GeneralEvaluationDomain::<Fr>::new(8).unwrap();
71  +        let evals_large = domain_large.fft(&coeffs_large);
72  +        let eval_poly_large = PolynomialEvalForm::new(evals_large.clone());
73  +        let blob_large = Blob::new(&eval_poly_large.to_bytes_be()).unwrap();
74  +
75  +        assert_ne!(
76  +            blob_small.data(),
77  +            blob_large.data(),
78  +            "blobs derived from different domains must differ"
79  +        );
80  +        println!("Small: {:?}", blob_small);
81  +        println!("Large: {:?}", blob_large);
82  +
83  +        let commitment = kzg
84  +            .commit_coeff_form(&polynomial, &SRS_INSTANCE)
85  +            .unwrap();
86  +        let commitment_from_small = kzg.commit_blob(&blob_small, &SRS_INSTANCE).
    unwrap();
87  +        let commitment_from_large = kzg.commit_blob(&blob_large, &SRS_INSTANCE).
    unwrap();
88  +
89  +        assert_eq!(commitment, commitment_from_small);
90  +        assert_eq!(commitment, commitment_from_large);
```

```
+
+        kzg.calculate_and_store_roots_of_unity(blob_small.len() as u64)
+            .unwrap();
+        let proof_small = kzg
+            .compute_blob_proof(&blob_small, &commitment, &SRS_INSTANCE)
+            .unwrap();
+        assert!(
+            verify_blob_kzg_proof(&blob_small, &commitment, &proof_small).unwrap(),
+            "proof for blob built on smaller domain should verify"
+        );
+
+        kzg.calculate_and_store_roots_of_unity(blob_large.len() as u64)
+            .unwrap();
+        let proof_large = kzg
+            .compute_blob_proof(&blob_large, &commitment, &SRS_INSTANCE)
+            .unwrap();
+        assert!(
+            verify_blob_kzg_proof(&blob_large, &commitment, &proof_large).unwrap(),
+            "proof for blob built on larger domain should verify"
+        );
+    }
+
     // Helper function to generate a point in the wrong subgroup
     fn generate_point_wrong_subgroup() -> G1Affine {
         let x = Fq::from_str(
```

Developer Response The developers now iterate over the `value.encoded_payloads` and validate the equality between the `encoded_payload` length and the length specified in the corresponding `AltDACommitment`.

5.1.2 V-EIG-VUL-002: Sp1 canoe zkVM application may use incorrect EVM fork

Severity	Critical	**Commit**	3e61825
Type	Data Validation	**Status**	Fixed
Location(s)	`canoe/sp1-cc/client/src/main.rs`		
Confirmed Fix At	hokulea/pull/205/..6b588a4e		

Description The SP1 canoe zkVM application initializes an Ethereum execution environment using a user-supplied EvmSketchInput, which can embed a `Genesis::Custom` configuration. During execution, it validates and emits important information about the EVM state, including the `chain_id` and `head_block_hash` but it does **not** commit any information about the execution configuration (e.g., fork rules, genesis hash, or chain spec version) to the output journal.

Since canoe takes in the genesis configuration as input, this omission allows an attacker to supply a forged genesis that matches a valid `chain_id` but diverges in fork parameters, hardfork activation points, or gas accounting logic. Since only the chain ID and head block hash are verified, such a modified configuration would produce a valid proof under incorrect execution semantics.

Impact An attacker can forge a valid proof over an execution environment that differs from the canonical Ethereum configuration for the same chain ID. This could result in:

- Incorrect verification of smart contract calls or proofs due to missing EVM instructions or precompiles
- Inconsistencies between the zkVM and the L1 consensus chain.
- Potential false positives in cert verification or incorrect attestations in downstream applications.

Because the execution environment is not committed to the journal, downstream verifiers cannot distinguish between valid and spoofed configurations.

Recommendation Include a hash of the full execution configuration (e.g., the `genesis` configuration) in the journal and enforce that the verifier checks this hash. Additionally:

- Consider hard-coding the supported `genesis` configurations and restrict supported chain IDs to those described by the supported configurations
- Panic or reject execution when an unrecognized chain ID or configuration is encountered.

This ensures that the zkVM can only produce proofs under trusted and verifiable network configurations.

Developer Response The supplied fix adds two hashes to the Journal. One hash corresponds to an identifier for the EVM fork the canoe execution is performed on. The second hash corresponds to the "genesis" configuration used by Canoe to configure the setup of the EVM environment, including the fork and several other important configuration parameters. These are then validated when checking a Canoe proof by reconstructing both the fork hash and genesis hash, ensuring the EVM environment matches the expected environment.

5.1.3 V-EIG-VUL-003: Risc0 Canoe zkVM application may use incorrect EVM fork

Severity	High	**Commit**	3e61825
Type	Data Validation	**Status**	Fixed
Location(s)	`canoe/steel/methods/guest/src/bin/cert_verification.rs`		
Confirmed Fix At	hokulea/pull/212/..4de8f8e1, hokulea/pull/226/..773dde0d, 35e24cde		

Description The Risc0 Canoe zkVM takes as input a chain ID corresponding to the L1 of the optimism-based rollup. From this chain ID, an appropriate `ChainSpec` is selected to configure the EVM execution environment. Currently chain specifications are provided for mainnet, sepolina and holesky, but other chain IDs default to `EthChainSpec::new_single(l1_chain_id, Default::default())` as the chain configuration. Note that this sets the environment's chain ID to the correct value but configures the hard-fork to a pre-determined default (currently Prague).

This approach introduces two risks:

1. **Outdated Configuration**: If the zkVM image is not kept up to date with recent hard forks, it may select an outdated `ChainSpec` even for known chains, resulting in execution divergence from the canonical chain.
2. **Unknown Chain IDs**: When an unrecognized `chain_id` is used, the fallback logic constructs a `ChainSpec` with the correct `chain_id` but with a default fork configuration. This default (currently Prague) may not match the actual fork rules of the target chain, leading to silent but incorrect execution semantics.

In both cases, the zkVM could produce a valid proof over an execution trace that does not match the intended network's behavior.

Impact If the zkVM uses an incorrect or outdated `ChainSpec`, it may produce a valid proof over invalid execution semantics. This can lead to:

- Incorrect verification of smart contract calls or proofs due to missing EVM instructions or precompiles
- Inconsistencies between the zkVM and the L1 consensus chain.
- Potential false positives in cert verification or incorrect attestations in downstream applications.

Since information about the EVM environment is not committed to the journal, it is difficult to determine when such issues impacted canoe's execution.

Recommendation Replace the default `EthChainSpec::new_single(...)` fallback with a hardcoded panic to reject unknown `chain_id` values. This ensures only explicitly supported and version-controlled configurations are used. Additionally:

- Commit the `SpecId` or fork configuration used during execution to the journal for traceability.
- Periodically update supported `ChainSpec` configurations as upstream forks evolve.

Developer Response The developers now panic on unknown `chain_id` values and commit the active fork to the journal.

5.1.4 V-EIG-VUL-004: Version byte check incorrectly parses data

Severity	Medium	**Commit**	3e61825
Type	Data Validation	**Status**	Fixed
Location(s)	`crates/eigenda/src/eigenda.rs:153-186`		
Confirmed Fix At	hokulea/pull/193/..15b5301e		

Description The `load_eigidenda_or_calldata()` function parses the data it receives from the `ethereum_source` based on the value of the first byte of the data, as can be seen below:

```
if data[0] == DERIVATION_VERSION_0 {
          info!(
              target = "eth-datasource",
              stage = "hokulea_load_encoded_payload",
              "use ethda at l1 block number {}",
              block_ref.number
          );
          self_contained_data.push(EigenDAOrCalldata::Calldata(data.clone()));
      } else {
          // retrieve all data from eigenda
          match self.eigenda_source.next(data, block_ref.number).await {
                        // Veridise elided
                        // ...
                        Ok(encoded_payload) => {
                              self_contained_data.push(EigenDAOrCalldata::EigenDA(
    encoded_payload));
                  }
}
```

Snippet 5.1: Snippet from `load_eigenda_or_calldata()`:

The `DERIVATION_VERSION_0` case represents the default OP-stack frame format. According to this specification, the alt-da payloads (which EigenDA is one of) must have the version byte be equal to `0x01` as OP-nodes will reject any value that is not one of these two cases. However, the implementation in `load_eigenda_or_calldata()` treats any non-`DERIVATION_VERSION_0` byte as the alt-da case. This leads to an inconsistency with regards to the OP-stack derivation process.

Impact A malicious batcher that sends an incorrect version byte can cause a fork between the Hokulea implementation and the OP-stack nodes.

Recommendation Handle the version byte in accordance with the spec.

Developer Response The developers now only query the `eigenda_source` when the version byte is equal to `ALTDA_DERIVATION_VERSION` (`0x1`). Outside of this, the data is pushed as `EigenDAOrCalldata::Calldata` and will be handled downstream by Kona.

5.1.5 V-EIG-VUL-005: Eager data processing can change error semantics

Severity	Medium	**Commit**	3e61825
Type	Logic Error	**Status**	Fixed
Location(s)	`crates/eigenda/src/eigenda.rs:121-192`		
Confirmed Fix At	hokulea/pull/233/..7828189d		

Description The `load_eigenda_or_calldata` function processes Ethereum L1 calldata and EigenDA payloads eagerly, preloading and processing all relevant data during the first call to the function. It uses a loop to collect all calldata items from `ethereum_source` and then, for each item, conditionally retrieves associated data from `eigenda_source`.

This design causes all errors from either source (e.g., parsing, decoding, EOF) to occur immediately when the function is called. These include temporary and critical errors triggered by a single malformed or malicious input from the batcher.

This aggressive loading design contrasts with a lazy iterator approach, where items are processed one-by-one only when consumed. Lazy evaluation would allow the caller to defer or avoid errors by simply not accessing problematic items or handling them granularly.

```
async fn load_eigenda_or_calldata(
    &mut self,
    block_ref: &BlockInfo,
    batcher_addr: Address,
) -> PipelineResult<()> {
    if self.open {
        return Ok(());
    }

    let mut calldata_list: Vec<Bytes> = Vec::new();
    // drain all the ethereum calldata from the l1 block
    loop {
        match self.ethereum_source.next(block_ref, batcher_addr).await {
            Ok(d) => calldata_list.push(d),
            Err(e) => {
                // break out the loop after having all batcher calldata for that
    block number
                // OP has different struct for handling pre and post ecotone. But
    both returns PipelineError::Eof
                if let PipelineErrorKind::Temporary(PipelineError::Eof) = e {
                    break;
                }
                return Err(e);
            }
        };
    }

    ...
}
```

Snippet 5.2: Snippet from `load_eigenda_or_calldata` showing the eager processing

Impact A a single entry provided by a batcher can cause the entire`load_eigenda_or_calldata` invocation to fail, blocking any processing of the data provided by the batcher. This enables

denial-of-service by pausing the pipeline at the earliest data availability stage. In a lazy processing design, only the individual item that causes the error would be affected, allowing the rest of the items to proceed unaffected. The current eager design gives attackers a more powerful disruption vector.

Recommendation Redesign `load_eigenda_or_calldata` to adopt lazy evaluation semantics. Avoid eagerly collecting and decoding all calldata and EigenDA blobs during the first invocation. Instead, yield items incrementally to consumers via an iterator or stream interface, and propagate errors at the point of access rather than during loading. This change will improve resilience against malformed or adversarial inputs and reduce the surface for batcher-induced denial-of-service attacks.

Developer Response The developers have changed the functionality of `EigenDADataSource::next()` to only load one piece of data at a time from the `ethereum_source`. Additionally, the parsing of the data and retrieval of the corresponding attributes from the `eigenda_source` is done during this call in order to return errors at the time of access.

5.1.6 V-EIG-VUL-006: Recency disagreements break determinism

Severity	Low	**Commit**	3e61825
Type	Data Validation	**Status**	Fixed
Location(s)	`bin/host/src/handler.rs:79-92`		
Confirmed Fix At	hokulea/pull/227, daf7c407		

Description The `fetch_eigenda_hint` function performs recency and validity checks on an `AltDACommitment` fetched from a proxy, then conditionally stores results in a key-value store. The current logic stores the recency window **before** querying the proxy and exits early if the proxy reports the certificate as not recent, skipping storage of the certificate's validity status.

This ordering introduces a consistency issue: if the proxy uses a **different recency window** than the configured recency window in the host (e.g., Hokulea), the proxy may return `is_recent_cert = false` even though the commitment is considered recent by the host. In this case, the client:

- Will not store the `is_valid_cert` flag in the key-value store
- Will later query the key-value store and fail to find validity information
- May cause downstream components like `OracleEigenDAWitnessProvider` or `load_eigenda_or_calldata` to error

Because this failure path depends on mismatched configuration across otherwise honest actors, it could result in non-deterministic network behavior. Nodes with the preimage may successfully load the blob and produce a proof, while others fail, leading to divergence in attestations.

Impact Disagreement on the recency window across the proxy and local configuration may cause:

- Valid but slightly stale certificates to be discarded prematurely
- Downstream errors in components relying on deterministic availability of cert validity
- Divergence in network state even among honest nodes

In the worst case, this could allow an honest zkVM to submit a valid proof using a cert that other clients failed to load due to early exit, enabling disputes or skipped blobs in consensus.

Recommendation Consider committing to the recency window of a given cert to ensure consistency across EigenDA components. This could be performed by adding the window to the cert itself, or adding the ability to prove the correctness of a given recency window. Note that while proxies may be trusted, to communicate the correct window, later modifications in trust assumptions or recency configurations could result in the disagreements reported in the impact section.

Developer Response This fix addresses the issue of misconfiguring recency value on the host and in the compiled ELF file. There is a corresponding change in the integration spec that makes exception for recency value=0. We had it because some users relies on early version of proxy that does not enforce recency check. For those users, a hardfork process is necessary before enabling the recency check. To address the consistency between proxy instance and hokulea host, there is a WIP PR on proxy right now that allows host to query recency_window on proxy.

The recommendation for this issue is pointing at a larger issue on how to ensure all proxy run by op-node are using the same value including the hokulea host and ELF. The team had a thorough discussion, and reached a consensus on putting recency window in the smart contract. But it would take at least 2 weeks to get PR ready, as we are evaluating tradeoff on more granular design space. In the short term to address the issue, we place additional checks and warnings to reduce the likelihood of misconfigurations between the proxy and service.

5.1.7 V-EIG-VUL-007: AnchorHash assumed to be block hash

Severity	Low	**Commit**	3e61825
Type	Data Validation	**Status**	Fixed
Location(s)	`canoe/sp1-cc/client/src/main.rs:91-92`		
Confirmed Fix At	hokulea/pull/195/..f0c81bc3		

Description The Canoe SP1 zkVM application creates an executor from a provided state sketch and performs contract calls to verify certificate validity. For each call, it asserts that the returned `anchor_hash` matches the `l1_head_block_hash` provided in the input to ensure that the current blockchain state matches the intended state described by the call.

This logic, however, does not verify the *type* of anchor returned. The SP1-cc infrastructure allows different anchor types to be used as inputs. This is reflected in the `public_vals.anchorHash` returned from a call as if a block anchor is provided as input, this indeed corresponds to a block hash. If a beacon anchor type is provided as input instead the stored value corresponds to a beacon root.

Impact Most likely, this will result in the canoe zkVM application panicking as it is unlikely for a beacon root to collide with a block hash. In the event such a collision is found, this could result in the call being executed on unintended blockchain state. Although hash collisions between block hashes and beacon roots are highly unlikely, this remains a correctness issue that could break invariants in downstream components or validation logic.

Recommendation Either explicitly check that the anchor type corresponds to a block, or consider using `executor.header().hash_slow()` to get the block header. This ensures the journal and all assertions refer unambiguously to the correct anchor context.

Developer Response The developers check that the `anchorType` of each call's `ContractPublicValues` is a `BlockHash`, and panic otherwise.

5.1.8 V-EIG-VUL-008: SP1-cc verification is skipped outside of zkvm environment

Severity	Low	**Commit**	3e61825
Type	Logic Error	**Status**	Fixed
Location(s)	`canoe/sp1-cc/verifier/src/lib.rs:66-68`		
Confirmed Fix At	hokulea/pull/215/..9a5c24ae		

Description The `validate_cert_receipt()` implementation for the `CanoeSP1CCVerifier` does not perform proof verification when outside of the `zkvm` target os. Rather, it prints a warning to the user that proof verification is being skipped as shown below.

```
#[allow(unused_variables)]
fn validate_cert_receipt(
    &self,
    cert_validity_pair: Vec<(AltDACommitment, CertValidity)>,
    canoe_proof_bytes: Option<Vec<u8>>,
) -> Result<(), HokuleaCanoeVerificationError> {
    info!("using CanoeSp1CCVerifier with v_key {:?}", V_KEY);

    assert!(!cert_validity_pair.is_empty());

    cfg_if::cfg_if! {
        if #[cfg(target_os = "zkvm")] {
            ...

        } else {
            warn!("Skipping sp1CC proof verification in native mode outside of zkVM,
    because sp1 cannot take sp1-sdk as dependency which is needed for verification in
     the native mode");
        }
    }
    Ok(())
}
```

Impact If Canoe proofs are verified outside of a zkVM, such as in a fault proof program, then a caller can set the validity of any `AltDACommitment`.

Recommendation Panic when not operating in the zkVM environment, as there is a no-op verifier that can be used for testing.

Developer Response The developers now `panic!` instead of `warn!` when not inside of the `zkvm`.

5.1.9 V-EIG-VUL-009: Uninitialized information in structs

Severity	Low	**Commit**	3e61825
Type	Maintainability	**Status**	Fixed
Location(s)	▶ `canoe/verifier/src/cert_validity.rs:11-23`		
Confirmed Fix At	hokulea/pull/213/..d71dcc3b		

Description Hokulea makes use of several structs such as `CertValidity` and `EigenDAWitness` that need to gather data from several different sources. During this process, some values may be left uninitialized as information is gathered, as is shown below:

```
async fn get_validity(
    &mut self,
    altda_commitment: &AltDACommitment,
) -> Result<bool, Self::Error> {
    // get cert validity
    match self.provider.get_validity(altda_commitment).await {
        Ok(validity) => {
            let mut witness = self.witness.lock().unwrap();

            let cert_validity = CertValidity {
                claimed_validity: validity,
                // the rest of the field needs to be supplied within zkVM
                l1_head_block_hash: B256::ZERO,
                l1_chain_id: 0,
                verifier_address: Address::default(),
            };

            witness
                .validities
                .push((altda_commitment.clone(), cert_validity));
            Ok(validity)
        }
        Err(e) => Err(e),
    }
}
```

Snippet 5.3: Snippet from `EigenDAPreimageProvider:get_validity()`

Impact As users of the Hokulea library may interact with these structs while some data is unavailable, it is possible that they may accidentally use the library incorrectly. For example, the `EigenDAWitness` expected to be passed to a zkVM application likely contains uninitiated data. A user is then expected to invoke `eigenda_witness_to_preloaded_provider()` to initialize the remaining data and convert the struct to a `PreloadedEigenDAPreimageProvider`. However, users may instead directly invoke `PreloadedEigenDAPreimageProvider::from_witness()` which does not initialize the missing data and therefore can lead to programming mistakes.

Additionally, allowing structs to contain such uninitiated data can cause maintainability issues in the future as developers of the library must determine whether a struct has been fully initialized or not.

Recommendations We would recommend splitting structs with uninitialized data such as `EigenDAWitness` and `CertValidity` into different structs depending on what data has been initialized or is available.

Developer Response The developers have created two new structs (`EigenDAPreimage` and `EigenDAWitnessWithTrustedData`) and have replaced `CertValidity` with a bool in the `EigenDAWitness`. The `EigenDAPreimage` is able to be converted to an `EigenDAWitness` when provided with kzg proofs and a Canoe proof. Additionally, the `OracleEigenDAWitnessProvider` has been changed to `OracleEigenDAPreimageProviderWithPreimage` and creates an `EigenDAPreimage` instead of an `EigenDAWitness`. Once an `EigenDAPreimage` has been converted to an `EigenDAWitness`, the `eigenda_witness_to_preloaded_provider()` will populate an `EigenDAWitnessWithTrustedData` with information from the `EigenDAWitness` and additional information from the Kona oracle. Additionally, the `PreloadedEigenDAPreimageProvider::from_witness()` now consumes the `EigenDAWitnessWithTrustedData` instead of the `EigenDAWitness` which validates information in the struct and populates the `PreoloadedEigenDAPreimageProvider` with trusted data.

5.1.10 V-EIG-VUL-010: Maintainability Improvements

Severity	Warning	**Commit**	3e61825
Type	Maintainability	**Status**	Fixed
Location(s)	`crates/` ▶ `eigenda/src/constant.rs`		
Confirmed Fix At	hokulea/pull/235, hokulea/pull/229, 92534ccc, 2e74a02c		

Description The security analysts identified the following places where the maintainability and understanding of the code can be improved:

1. `crates/eigenda/src/constants.rs`: The comment on the `BYTES_PER_FIELD_ELEMENT` should say the number of bytes for field element
2. `crates/egenda/src/eigenda_data.rs`:
   a) The comment on `encode()` should specify the length is at the `2-5` indices
3. `canoe/steel/apps/src/apps.rs`:
   a) The info log regarding the start of the proof generation should say that this is a Steel proof, not sp1-cc.
4. `bin/host/src/lib.rs`:
   a) The `init_tracing_subscriber()` has a duplicate definition in `bin/host/src/main.rs`
5. `crates/witgen/src/canoe_witness_provider.rs`:
   a) The `from_boot_info_to_canoe_proof()` function will set the `l1_head_block_hash` in each `cert_validity` in the `wit.validities` vector. However, these values are never used after being set.
6. `crates/proof/src/preloaded_eigenda_provider.rs`
   a) In the `PreloadedEigenDAPreimageProvider`, rather than popping and unwrapping entries from the queue, it would be useful to panic with an informative message so that users know that the host did not provide the `EigenDAPreimageProvider` with sufficient data to fulfill the request.

Impact The maintainability and understanding of the code may be reduced, potentially leading to issues in the future.

Recommendation Implement the recommended changes.

Developer Response The developers have fixed all above instances with the provided recommendations.

5.1.11 V-EIG-VUL-011: Improperly formatted EncodedPayloads may pass decoding

Severity	Warning	**Commit**	3e61825
Type	Data Validation	**Status**	Fixed
Location(s)	`crates/eigenda/src/eigenda_data.rs:133`		
Confirmed Fix At	hokulea/pull/236/..0b283438		

Description The `EncodedPayload::decode()` function is used to decode the `EncodedPayload` into the vector of Bytes of a `Payload`. To do so, it first checks that important properties regarding the `EncodedPayload` hold such as the length must be equal to a power of two as is required by their specification. While these length requirements are checked, other requirements in the specification are not checked such as:

1. The requirement that padding be made up of all 0's
2. The encoded payload consists of valid field elements which is enforced by ensuring that the first byte of every field element is `0x0`

While some of the above requirements are checked in other locations of the code, it is not required that paths to this location must always execute the code that checks the missing requirements.

Impact Payloads that do not properly conform to the specification and with respect to the `payload_len` may be accepted which could indicate that a blob is not formatted correctly and therefore may not be parsed correctly.

Recommendation Check all requirements for an `EncodedPayload` before decoding to ensure previous checks were not bypassed and the `EncodedPayload` has valid formatting.

Developer Response The developers have implemented the necessary checks during decoding to ensure the specification is met.

5.1.12 V-EIG-VUL-012: Integration requirements should be documented

Severity	Warning	**Commit**	3e61825
Type	Usability Issue	**Status**	Fixed
Location(s)	`README.md`		
Confirmed Fix At	hokulea/pull/223, 5755fda6		

Description Hokulea's crates are intended to be used as libraries by OP-stack implementations to add support for using EigenDA as an off-chain data availability provider. Therefore, it is crucial for the library to provide clear documentation on how to securely integrate it such as what validations must be made, and the library's operational assumptions.

As an example, for zkVM applications, a crucial assumption made is that the user of the library will make critical data from the underlying oracle such as the `BootInfo` available for validation. Without doing so, it would be possible for users of the library to manipulate state in ways that are not observable to the user. It is therefore important to make such assumptions clear in documentation about how users should interact with the clients when creating a zkVM application and a fpVM client. It is also important that these assumptions be made explicit in any examples such as the one given in `example/preloader/src/main.rs`. Currently it demonstrates how a zkVM application could use the library but does not make explicit what data must be made public.

```
#[allow(clippy::type_complexity)]
pub async fn run_within_zkvm<O, Evm>(
    oracle: Arc<O>,
    evm_factory: Evm,
    canoe_verifier: impl CanoeVerifier,
    canoe_address_fetcher: impl CanoeVerifierAddressFetcher,
    witness: EigenDAWitness,
) -> anyhow::Result<()>
where
    O: CommsClient + FlushableCache + Send + Sync + Debug,
    Evm: EvmFactory<Spec = OpSpecId> + Send + Sync + Debug + Clone + 'static,
    <Evm as EvmFactory>::Tx: FromTxWithEncoded<OpTxEnvelope> + FromRecoveredTx<
    OpTxEnvelope>,
{
    info!("start the code supposed to run inside zkVM");

    let beacon = OracleBlobProvider::new(oracle.clone());
    let preloaded_preimage_provider = eigenda_witness_to_preloaded_provider(
        oracle.clone(),
        canoe_verifier,
        canoe_address_fetcher,
        witness,
    )
    .await?;

    // this is replaced by fault proof client developed by zkVM team
    fp_client::run_fp_client(oracle, beacon, preloaded_preimage_provider, evm_factory
    ).await?;

    Ok(())
}
```

Snippet 5.4: Snippet of the preloader example that is intended to be implemented in a zkVM application

Similarly, while this library is intended for use by zkVM applications, if other applications make use of this library and do not create a `PreloadedEigenDAPreimageProvider`, it is expected that either they or a trusted upstream application has checked a blob's KZG commitment before serving it to the oracle for consumption by an `OracleEigenDAWitnessProvider`.

Impact Incorrect usage of the library may invalidate the security assumptions it operates under.

Recommendation Provide documentation and example usage of the validations that must be made when integrating the library.

Developer Response The developers have provided more detailed documentation surrounding integration in their markdown documents.

5.1.13 V-EIG-VUL-013: Defensive programming improvements

Severity	Warning	**Commit**	3e61825
Type	Maintainability	**Status**	Fixed
Location(s)	▶ `bin/host/src/handler.rs:263-276` ▶ `canoe/` • `sp1-cc/client/src/main.rs:10-105` • `steel/` * `apps/src/apps.rs:60-63` * `methods/guest/src/bin/cert_verification.rs:32-89` • `verifier/src/verifier.rs:62-72` ▶ `crates/` • `eigenda/src/eigenda_data.rs:125`		
Confirmed Fix At	hokulea/pull/230, hokulea/pull/229, 7a1181a7, 2e74a02c		

Description The security analysts identified the following locations where the the code could be strengthened to prevent potential corner cases or mistakes:

1. `canoe/verifier/src/verifier.rs`
   a) The `CanoeNoopVerifier` could be wrapped in a testing module, deprecated or could print a warning when used to prevent potential use in production which would cause proofs not to be verified.
2. `crates/eigenda-cert/src/lib.rs`:
   a) The `G2Point` struct uses a `Vec<U256>` to represent each coordinate, but a better guarantee on the size would be to use a size 2 array or tuple.
3. `crates/eigenda/src/eigenda_data.rs`:
   a) In `decode_payload` rather than downcasting the `decoded_body` length, consider upcasting the `payload_len` to prevent potential trucation
4. `bin/host/src/handler.rs`
   a) Currently there is a check that the content of an encoded payload is within the range of a field element, but the first field which contains metadata about the payload is unchecked even though it also must be a field element. Consider checking this as well rather than relying on an external library
5. `canoe/steel/apps/src/apps.rs`
   a) In the `get_steel_proof` function that acts as the host for the steel-based canoe zkVM application, the block hash is not checked against the environment and the canoe inputs are not checked to ensure they have a consistent block hash. While this will be checked in the client, doing so here allows the application to exit early if used improperly.
6. `canoe/sp1-cc/client/src/main.rs` and `canoe/steel/methods/guest/src/bin/cert_verification.rs`
   a) In both canoe zkVM applications, the canoe input includes a `claimed_validity` that is never used. It is possible that users will think a succeeding zkVM proof implies that the `claimed_validity` is correct.

Impact Such corner cases while unlikely now could result in more significant errors in the future, especially if requirements change.

Recommendation Implement the recommended changes.

Developer Response The developers have addressed all provided instances as recommended.

Glossary

AuditHub A security platform used by Veridise to perform audits. It provides auditors and developers the ability to engage in discussions regarding the code, hosts the audit issues and facilitates the fix review process. In addition, AuditHub provides DeFi and ZK security tools that are used to identify issues and provide security guarantees during the audit process. More information can be found at https://audithub.dev/ . 4

Data Availability Provider (DAP) A network that provides data availability requirements as an alternative to EIP-4844. Normally, these networks provide an attestation and commitment to Ethereum for layer-2s to utilize with similar data availability guarantees as provided by EIP-4844. Note that these layer-2s are now considered Validiums, as there are possible security tradeoffs and rollups must strictly use Ethereum as a data availability source . 1

EIP-4844 Also known as Proto-Danksharding, EIP-4844 introduces "blob" carrying transactions to Ethereum. A blob is a large amount of data that may expire and only be stored by a subset of nodes (a process known as data availability sharding (DAS)). This data is not available to EVM execution, but a commitment to the data is. This commitment uses the KZG commitment scheme in order to prove the blob data that is a part of the commitment. The motivation of blobs is to provide data availability to rollups at a much cheaper cost than calldata, as the data is not necessary to exist outside of the rollup's challenge window . 1, 30

optimistic rollup A rollup in which the state transition of the rollup is posted "optimistically" to the base network. A system involving stake for resolving disputes during a challenge period is required for economic security guarantees surrounding finalization . 1

rollup A blockchain that extends the capabilities of an underlying base network, such as higher throughput, while inheriting specific security guarantees from the base network. Rollups contain smart contracts on the base network that attest the state transitions of the rollup are valid . 1, 30

smart contract A self-executing contract with the terms directly written into code. Hosted on a blockchain, it automatically enforces and executes the terms of an agreement between buyer and seller. Smart contracts are transparent, tamper-proof, and eliminate the need for intermediaries, making transactions more efficient and secure. 30

zero-knowledge circuit A cryptographic construct that allows a prover to demonstrate to a verifier that a certain statement is true, without revealing any specific information about the statement itself. See https://en.wikipedia.org/wiki/Zero-knowledge_proof for more. 30

zkVM A general-purpose zero-knowledge circuit that implements proving the execution of a virtual machine. This enables general purpose programs to prove their execution to outside observers, without the manual constraint writing usually associated with zero-knowledge circuit development . 1