Veridise

Auditing Report

Hardening Blockchain Security with Formal Methods

FOR

RISC ZERO

Kailua Protocol

Veridise Inc.

May 22, 2025

► **Prepared For:**

RISC Zero
https://risczero.com/

► **Prepared By:**

Benjamin Mariano
Tyler Diamond
Victor Faltings

► **Contact Us:**

contact@veridise.com

► **Version History:**

May 22, 2025 V2
May 19, 2025 V1

Contents

1 Executive Summary

From May 5, 2025 to May 15, 2025, RISC Zero engaged Veridise to conduct a security assessment of their Kailua Protocol. The Kailua Protocol enables OP-stack optimistic rollups to utilize zero-knowledge proofs for resolving disputes between proposed output roots. Compared to the earlier version, which Veridise has audited previously*, the new version addresses issues brought up in the initial audit and introduces functionality of utilizing validity proofs to prove the legitimacy of a proposal. This report only addresses the smart contract implementation, not the off-chain zero-knowledge components. Veridise conducted the assessment over 6 person-weeks, with 3 security analysts reviewing the project over 2 weeks on commit `7eb9869`. 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 smart contracts involved in the Kailua Protocol. These contracts are responsible for storing the proposals of state transitions of the given rollup, and implement processes for resolving disputes between proposals.

Notably, there are three methods for determining the legitimacy of proposals:

1. `Output Fault`: A proposal is proven in zero-knowledge to contain an illegitimate output root from the OP-stack L2 state transition, and should therefore be eliminated.
2. `Null Fault`: A proposal is found to have an improperly formatted sequence of output roots, and should therefore be eliminated.
3. `Validity Proven`: A proposal is proven in zero-knowledge to contain the correct state transitions for its final output root, and therefore should be marked as the correct proposal for finalization.

These three methods, along with a process for eliminating disputing proposals, implement the core functionality of the smart contracts.

Code Assessment. The Kailua Protocol developers provided the source code of the Kailua Protocol contracts for the code review. The source code appears to be mostly original code written by the Kailua Protocol developers. It contains some documentation in the form of READMEs, a gitbook† and documentation comments on functions and storage variables. To facilitate the Veridise security analysts' understanding of the code, the Kailua Protocol developers met with the Veridise security analysts and quickly answered any questions that arose.

The source code contained a test suite, which the Veridise security analysts noted contained both positive and negative tests and achieved 100% coverage of the smart contracts under review.

* The previous audit report, if it is publicly available, can be found on Veridise's website at `https://veridise.com/audits/`

† `https://risc0.github.io/kailua/`

Summary of Issues Detected. The security assessment uncovered 6 issues. No high or critical severity issues were found. Veridise analysts identified 2 medium-severity issues, including eliminating a resolved proposal (V-KLA-VUL-002) and losing bonds for duplicate proposals (V-KLA-VUL-001). Analysts also identified 1 low-severity issue, 2 warnings, and 1 informational finding. RISC Zero has fixed all of the identified issues.

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
Kailua Protocol	`7eb9869`	Solidity	Ethereum

Table 2.2: Engagement Summary.

Dates	Method	Consultants Engaged	Level of Effort
May 5–May 15, 2025	Manual & Tools	3	6 person-weeks

Table 2.3: Vulnerability Summary.

Name	Number	Acknowledged	Fixed
Critical-Severity Issues	0	0	0
High-Severity Issues	0	0	0
Medium-Severity Issues	2	2	2
Low-Severity Issues	1	1	1
Warning-Severity Issues	2	2	2
Informational-Severity Issues	1	1	1
TOTAL	6	6	6

Table 2.4: Category Breakdown.

Name	Number
Logic Error	4
Maintainability	2

3 Security Assessment Goals and Scope

3.1 Security Assessment Goals

The engagement was scoped to provide a security assessment of Kailua Protocol's smart contracts. During the assessment, the security analysts aimed to answer questions such as:

- Can invalid proposals become resolved?
- Are zero-knowledge proofs correctly validated?
- Do the proving functions implement their intended functionality?
- Can a proposer ever be double eliminated, and therefore cause a Denial-of-Service in the protocol?
- Can validity proofs be submitted as fault proofs or vice versa?
- Can a user steal payment funds from an honest prover?
- Can any of the previously reported issues be triggered?
- Is the protocol subject to common smart contract bugs such as reentrancy attacks?

3.2 Security Assessment Methodology & Scope

Security Assessment Methodology. To address the questions above, the security assessment involved a combination of human experts and automated program analysis & testing tools. In particular, the security assessment was conducted with the aid of the following techniques:

- *Static analysis.* To identify potential common vulnerabilities, security analysts leveraged Veridise's custom smart contract analysis tool Vanguard, as well as the open-source tool Slither. These tools are designed to find instances of common smart contract vulnerabilities, such as reentrancy and uninitialized variables.

Scope. The scope of this security assessment is limited to the four Solidity files defined in the `crates/contracts/foundry/src` folder of the source code provided by the Kailua Protocol developers. It should be noted that the files in `crates/contracts/foundry/src/vendor` contain imported code and were not in the scope of the audit. Additionally, this version of the review did not consider any of the off-chain components.

Methodology. Veridise security analysts reviewed the reports of previous audits for Kailua Protocol, inspected the provided tests, and read the Kailua Protocol documentation. They then began a review of the code assisted by static analyzers.

During the security assessment, the Veridise security analysts met with the Kailua Protocol developers to ask questions about the code and share their findings.

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.

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

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

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

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

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 Kailua Protocol.

- The `DisputeGameFactory` owner will only set the implementation of the Kailua `GameType` to the compilation of the version of `KailuaGame.sol` reviewed in this report.
- The bond in the `KailuaTreasury` will be set to a reasonable amount.

4.1.1 Off-chain Component Assumptions

The smart contracts in Kailua Protocol interact closely with an off-chain RISC Zero zkVM application that is used to create both validity and fault proofs for proposals on chain. Correct behavior of the smart contracts relies on a number of assumptions about the behaviors of the off-chain component, so we make some of the most important of those assumptions about the off-chain component explicit here:

- All L2 block header data used for derivation proofs are fetched from an L1 block with a hash matching the L1 head hash in the journal.
- There is a valid derivation proof from the L2 block header corresponding to `acceptedOutputHash` from the journal to the L2 block header corresponding to `computedOutputHash` also from the journal.
- The L2 block header hash `computedOutputHash` in the journal has the L2 block number `claimedBlockNumber` also from the journal.
- The off-chain component image ID matches the on-chain image ID.
- The off-chain rollup configuration hash matches the on-chain configuration hash.
- The off-chain specified the payout recipient in the journal as the entity which should receive payout for the proof.
- If the precondition hash in the journal is non-zero, the number of L2 blocks in the derivation match the expected length on-chain and the intermediate roots have a hash matching the one provided on-chain.
- An off-chain validity proof verifies that all trailing field elements which do not correspond to actual roots are set to zero.
- The off-chain prover will only validate a given root if the hashes provided to determine that root are valid field elements.

We shared these assumptions with the Kailua Protocol developers, who indicated that these assumptions should be satisfied by the off-chain component. Because there could be vulnerabilities in the smart contracts if any of these assumptions are violated, we strongly recommend that the Kailua Protocol developers independently verify that the off-chain component does in fact satisfy these assumptions.

4.2 Privileged Roles.

Roles. This section describes in detail the specific roles present in the system, and the actions each role is trusted to perform. The roles are grouped based on their privilege-level. *Highly-privileged* roles may have a critical impact on the protocol if compromised, while *limited-authority* roles have a negative, but manageable impact if compromised.

During the review, Veridise analysts assume that the role operators perform their responsibilities as intended. Protocol exploits relying on the below roles acting outside of their privileged scope are considered outside of scope.

- Highly-privileged roles:
  - `DisputeGameFactory` owner can set the implementation address of dispute games, set the participation bond required in the `KailuaTreasury` and set the Vanguard address and advantage duration.
- Limited-authority roles:
  - `KailuaTreasury.vanguard` is given a set amount of time at the beginning of each round of dispute games, configured with the `vanguardAdvantage` variable, in which they are the only proposer allowed to create a proposal.

Operational Recommendations. Highly-privileged, non-emergency operations should be operated by a multi-sig contract or decentralized governance system.

Full validation of operational security practices is beyond the scope of this review. Users of the protocol should ensure they are confident that the operators of privileged keys are following best practices such as:

- Never storing a protocol key in plaintext, on a regularly used phone, laptop, or device, or relying on a custom solution for key management.
- Using separate keys for each separate function.
- Storing multi-sig keys in a diverse set of key management software/hardware services and geographic locations.
- Enabling 2FA for key management accounts. SMS should *not* be used for 2FA, nor should any account which uses SMS for 2FA. Authentication apps or hardware are preferred.
- Validating that no party has control over multiple multi-sig keys.
- Performing regularly scheduled key rotations for high-frequency operations.
- Securely storing physical, non-digital backups for critical keys.
- Actively monitoring for unexpected invocation of critical operations and/or deployed attack contracts.
- Regularly drilling responses to situations requiring emergency response such as pausing/unpausing.

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-KLA-VUL-001	Duplicate proposal bonds are lost	Medium	Fixed
V-KLA-VUL-002	Late validity proof allows eliminating a . . .	Medium	Fixed
V-KLA-VUL-003	Bonds for proposals made after challenge . . .	Low	Fixed
V-KLA-VUL-004	Proofs can be submitted after resolution	Warning	Fixed
V-KLA-VUL-005	Unused program constructs	Warning	Fixed
V-KLA-VUL-006	Typos, incorrect comments, and small code . . .	Info	Fixed

5.1 Detailed Description of Issues

5.1.1 V-KLA-VUL-001: Duplicate proposal bonds are lost

Severity	Medium	**Commit**	7eb9869
Type	Logic Error	**Status**	Fixed
File(s)	`KailuaTournament.sol`		
Location(s)	pruneChildren()		
Confirmed Fix At	`https://github.com/risc0/kailua/pull/46`		

When a proposal has a duplicate signature (an attestation to the state roots of a proposal) of the current `contender`, it is added to the `contenderDuplicates` array.

```
bytes32 opponentSignature = opponent.signature();
if (opponentSignature == contenderSignature) {
    contenderDuplicates.push(v);
    continue;
}
```

Snippet 5.1: Snippet from `KailuaTournament.sol:pruneChildren()`

If said `contender` eventually resolves as the correct proposal, then the duplicates are not eliminated (as they are also correct proposals), but they are also not set to resolved. The `KailuaTreasury` implements no method of claiming back the bonds of these duplicates.

Impact The bonds of duplicates proposals become stuck in the `KailuaTreasury` contract. There may also be issues with claiming bonds unexpectedly with this combined with V-KLA-VUL-002.

Recommendation Add functionality to claim duplicate proposal bonds once resolution has occurred.

Developer Response The developers have modified `KailuaTreasury.claimProposerBond()` to enable a proposer to withdraw their bond if they are not eliminated, and if the most recent game they have proposed in is resolved. This means duplicate proposals of the correct proposal can have their bonds reclaimed once the tournament they are in is resolved because these proposals are not eliminated in `KailuaTournament.pruneChildren()` if they are duplicates of the correct proposal.

5.1.2 V-KLA-VUL-002: Late validity proof allows eliminating a resolved proposal

Severity	Medium	**Commit**	7eb9869
Type	Logic Error	**Status**	Fixed
File(s)	`KailuaTournament.sol`		
Location(s)	pruneChildren()		
Confirmed Fix At	`https://github.com/risc0/kailua/pull/43/`, c11ab54		

A `KailuaGame` P may be resolved in one of two ways:

1. After a period `MAX_CLOCK_DURATION` has expired without being challenged by another proposal or fault proof.
2. By submitting a validity proof for P.

In the case of the first situation, an issue may arise if P is later shown to be invalid.

While a proposer cannot submit a fault proof for P after the challenge period, they can instead submit the correct proposal C along with a validity proof of C. Both of these operations are permitted by the treasury and tournament contracts.

As a result of the validity proof of C, the signature of P will no longer be valid according to `isViableSignature()`, causing `pruneChildren()` to eliminate it.

```
function isViableSignature(bytes32 childSignature) public view returns (bool
    isViableSignature_) {
  if (validChildSignature != 0) {
    isViableSignature_ = childSignature == validChildSignature;
  }
  // ...
}
```

Snippet 5.2: Snippet from `isViableSignature()`

```
// ...
if (!isViableSignature(contenderSignature) || isChildEliminated(contender)) {
    // ...
  // Eliminate contender
  if (!isChildEliminated(contender)) {
    KAILUA_TREASURY.eliminate(address(contender), payoutRecipient);
  }
  // ...
}
// ...
```

Snippet 5.3: Snippet from `pruneChildren()`

The proposal C on the other hand, can be freely resolved as it has been proven to be valid. Upon resolution of C, both P and its proposer will be marked as eliminated (despite P already having been resolved and potentially extended with more child proposals).

Impact As a consequence of the situation described above, both P and C will be marked as resolved (and thus can be extended by more proposals), creating a fork in the chain.

Another dangerous effect of this is that the proposer of P may be eliminated out-of-order, which can cause additional issues.

Proof of Concept The following test case can be added to `ClaimDispute.t.sol` to test for this issue:

```
function test_pruneChildrenAfterAccepting() public {
  vm.warp(
    game.GENESIS_TIME_STAMP() + game.PROPOSAL_TIME_GAP()
      + game.PROPOSAL_OUTPUT_COUNT() * game.OUTPUT_BLOCK_SPAN() * game.L2_BLOCK_TIME
    ()
  );

  // Submit invalid proof
  vm.startPrank(address(0x1));
  KailuaTournament invalid = treasury.propose(
    Claim.wrap(0x0001010000010100000010100000101000001010000010100000010100000101),
    abi.encodePacked(uint64(128), uint64(anchor.gameIndex()), uint64(0))
  );
  vm.stopPrank();

  // Finalize claim after clock period expires
  vm.warp(
    game.GENESIS_TIME_STAMP() + game.PROPOSAL_TIME_GAP()
      + game.PROPOSAL_OUTPUT_COUNT() * game.OUTPUT_BLOCK_SPAN() * game.L2_BLOCK_TIME
    () * 2
  );
  invalid.resolve();

  // Submit late proposal
  vm.startPrank(address(0x2));
  KailuaTournament valid = treasury.propose(
    Claim.wrap(0x0001010000010100000010100000101000001010000010100000010100000100),
    abi.encodePacked(uint64(128), uint64(anchor.gameIndex()), uint64(0))
  );
  vm.stopPrank();
  assert(invalid.signature() != valid.signature());

  // Submit validity proof
  bytes memory proof = mockValidityProof(
    address(this),
    valid.l1Head().raw(),
    valid.parentGame().rootClaim().raw(),
    valid.rootClaim().raw(),
      uint64(valid.l2BlockNumber()),
    uint64(valid.PROPOSAL_OUTPUT_COUNT()),
    uint64(valid.OUTPUT_BLOCK_SPAN()),
    valid.blobsHash()
  );

  // Resolve late proposal through validity proof
  valid.parentGame().proveValidity(address(this), uint64(1), proof);
  valid.resolve();
```

```
46
47    // 2 siblings marked as accepted
48    assert(invalid.status() == GameStatus.DEFENDER_WINS);
49    assert(valid.status() == GameStatus.DEFENDER_WINS);
50    assert(invalid.parentGame() == valid.parentGame());
51
52    // Original proposer eliminated
53    assert(treasury.eliminationRound(address(0x1)) > 0);
54  }
```

Recommendation We recommend modifying the logic of the contracts to explicitly and carefully handle proposals submitted after a child of a tournament has been successfully resolved.

Developer Response The developers have added a check in `pruneChildren()` to see if the current `contender` has been marked as resolved. If so, it returns it immediately. They have also added a test case for this issue.

5.1.3 V-KLA-VUL-003: Bonds for proposals made after challenge duration are lost

Severity	Low	**Commit**	7eb9869
Type	Logic Error	**Status**	Fixed
File(s)	`KailuaTournament.sol`		
Location(s)	pruneChildren()		
Confirmed Fix At	`https://github.com/risc0/kailua/pull/46`		

Children of a proposal may be added via `appendChild()` at any point, even after resolution has occurred. `pruneChildren()` should eliminate all other proposals that were not resolved as the winning contender. However, once the challenge duration against a contending proposal has passed, the `pruneChildren()` function will no longer eliminate children.

```
if (contender.getChallengerDuration(opponent.createdAt().raw()).raw() == 0) {
    // Note: This implies eliminationLimit > 0
    break;
}
```

Snippet 5.4: Snippet from `KailuaTournament:pruneChildren()`

Therefore, any proposals made after the challenge duration of a valid proposal will not be eliminated.

Impact The bonds of the proposals that are not eliminated will be stuck in the `KailuaTreasury` contract.

Recommendation Have a method of eliminating children after a contender has been resolved. Note that if the behavior of submitting proposals after resolution is desirable for fork resolution, then care must be taken on propagating elimination to later proposals.

Developer Response The developers have modified `KailuaTreasury.claimProposerBond()` to enable a proposer to withdraw their bond if they are not eliminated, and if the most recent game they have proposed in is resolved. This allows withdrawal of any proposals submitted after the challenge duration as these proposals are never processed (and thus their proposers are not eliminated).

5.1.4 V-KLA-VUL-004: Proofs can be submitted after resolution

Severity	Warning	**Commit**	7eb9869
Type	Logic Error	**Status**	Fixed
File(s)	`KailuaTournament.sol`		
Location(s)	proveNullFault(), proveOutputFault(), proveValidity()		
Confirmed Fix At	`https://github.com/risc0/kailua/pull/44`, dbfd8d4		

The tournament allows proofs to be submitted for children. In the proving functions (`proveNullFault`, `proveOutputFault`, and `proveValidity`), there are no checks that the tournament is still on-going (i.e., there is not already a resolved child).

Impact In V-KLA-VUL-002 we show how actions after resolution can be used to create unexpected behaviors.

Recommendation As far as we can tell, there is no reason to continue to allow proving after resolution. We would suggest disallowing proving after resolution.

Developer Response The developers have added extra checks to the tournaments to prevent the following behavior:

- Appending a new child tournament when the contender child has already been successfully resolved.
- Submitting a proof (through `proveNullFault`, `proveOutputFault` or `proveValidity`) for a child tournament when the contender child has already been successfully resolved.

With these changes, the state of a tournament is now locked once one of its children is resolved.

5.1.5 V-KLA-VUL-005: Unused program constructs

Severity	Warning	**Commit**	7eb9869
Type	Maintainability	**Status**	Fixed
File(s)	See issue description		
Location(s)	See issue description		
Confirmed Fix At	`https://github.com/risc0/kailua/pull/45`, d41694c		

Description The following program constructs are unused:

1. `KailuaLib.sol`:
   a) `error GameTypeMismatch(GameType parentType, GameType expectedType)`
   b) `error BlockCountExceeded(uint256 l2BlockNumber, uint256 rootBlockNumber)`
   c) `error BlobHashMismatch(bytes32 found, bytes32 expected)`

Impact These constructs may become out of sync with the rest of the project, leading to errors if used in the future.

Developer Response The developers have removed the unused errors from the codebase.

5.1.6 V-KLA-VUL-006: Typos, incorrect comments, and small code suggestions

Severity	Info	**Commit**	7eb9869
Type	Maintainability	**Status**	Fixed
File(s)	See issue description		
Location(s)	See issue description		
Confirmed Fix At	`https://github.com/risc0/kailua/pull/45`, d41694c		

Description In the following locations, the auditors identified minor typos and potentially misleading comments:

1. `KailuaTreasury.sol`:
    a) Instead of defining your own `nonReentrant` modifier, consider using OpenZeppelin's standard implementation.
    b) The comment on line 35 above the definition of `L2_BLOCK_NUMBER` incorrectly says it is the root claim.
2. `KailuaTournament.sol`:
    a) In `pruneChildren`, a duplicate of the contender will count *twice* towards the elimination count - once when added to the `contenderDuplicates` array and once when eliminated from that array (if ever eliminated). This may ultimately be fine, but means that contender duplicates count doubly towards elimination count when no other children will.

Impact These minor errors may lead to future developer confusion.

Developer Response The developers have fixed the incorrect documentation comment and renamed the `eliminationLimit` variable to `stepLimit` to more accurately reflect its intended behavior (for example regarding handling of duplicates). For the `nonReentrant` modifier, they have noted that using the OpenZeppelin library is something they are open to in the future, should they require more of its functionality.

Glossary

Denial-of-Service An attack in which the liveness or ability to use a service is hindered . 4

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 . 17

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. 1, 17