Veridise

Auditing Report

Hardening Blockchain Security with Formal Methods

FOR

RISC ZERO

Kailua Protocol

Veridise Inc.

Feb. 18, 2025

► **Prepared For:**

RISC Zero
https://risczero.com/

► **Prepared By:**

Benjamin Mariano
Tyler Diamond
Victor Faltings

► **Contact Us:**

contact@veridise.com

► **Version History:**

Feb. 18, 2025	V3
Feb. 17, 2025	V2
Feb. 4, 2025	V1
Feb. 3, 2025	Initial Draft

Contents

1 Executive Summary

From Jan. 6, 2025 to Jan. 28, 2025, RISC Zero engaged Veridise to conduct a security assessment of their Kailua Protocol, which aims to create an infrastructure for optimistic rollups that resolve disputes with a zero-knowledge virtual machine (zkVM) application. The security assessment covered the smart contracts, off-chain components, and zkVM application used to implement the protocol. Veridise conducted the assessment over 9 person-weeks, with 3 security analysts reviewing the project over 3 weeks on commit `6e2ce8f`. 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 Kailua Protocol implementation, including both on-and-off-chain components. The implementation can be roughly separated into the following main components:

- **Smart contracts**: The smart contracts maintain prospective and finalized proposals (a set of rollup blocks proposed for finalization). The contracts process and store dispute proofs and resolve said disputes in order to eventually increment the finalized rollup block.
- **zkVM Application**: The RISC Zero zkVM is used to prove the execution of the OP-stack (Optimism's optimistic rollup implementation) chain derivation function. This application produces the deterministic rollup state root by deriving it from the rollup data that has been posted to the base network. In other words, it produces a proof that a given L2 block has a particular state root which can be used to invalidate proposals made on-chain when necessary.
- **Off-chain components**: The off-chain portion primarily consists of the zkVM application and two addition components: the Proposer and Validator. The former will propose new proposals when the layer-2 has created enough blocks to do so, and will attempt to finalize proposals if they are valid and all disputes can be resolved. The Validator monitors the base network for disputes between proposals that need to be proven in zero-knowledge, and submits those zero-knowledge proofs to 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, documentation comments in functions and storage variables. To facilitate the Veridise security analysts understanding of the code, the Kailua Protocol developers frequently met with the security analysts and quickly answered any questions that arose.

The source code did not contain a test suite for any of the components.

Summary of Issues Detected. The security assessment uncovered 24 issues, 13 of which are assessed to be of high or critical severity by the Veridise analysts. For instance, V-KLA-VUL-004 notes an issue in which the rollup ceases to function when a malicious smart contract is set as

the recipient of a successful dispute game reward. Additionally, V-KLA-VUL-006 describes how certain elimination orders for proposals can again cause the rollup to cease progressing the finalization of the chain. Veridise analysts also identified 3 medium-severity issues, including an issue in which validators do not re-attempt to submit proofs when handling erroneous conditions (V-KLA-VUL-014), as well as 2 low-severity issues, 4 warnings, and 2 informational findings.

The Kailua Protocol developers have acknowledged all of the issues identified. While all issues have been addressed, it should be noted that some issues have been marked only as "addressed and partially verified." The changes introduced to fix these issues were stacked on top of a number of substantial changes to the code; Veridise auditors validated that each fix removed the original identified issue, but were unable to verify whether the fixes introduced new/related bugs with respect to the additional logic. Thus, a status of "addressed and partially verified" does not necessarily indicate that the fix is known to be wrong or incomplete, but rather that Veridise auditors were unable to validate whether or not the fixes may have introduced related or entirely new issues.

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

Documentation. Veridise auditors believe the code quality and security can be substantially improved by increasing and improving documentation of the project. Although there are extensive comments on the code within functions, both the smart contracts and Rust code lack comments at higher-level that help explain the expected use/behavior of functions, function parameter explanations, etc. Furthermore, there are a number of very subtle assumptions that are made in the code implicitly that are never explained explicitly. As an example, there is an implicit assumption in the smart contracts that there is only a single deployed treasury contract that every subsequent dispute game references. This is never stated explicitly but is a critical assumption as to the expected behavior of the protocol. Additionally, the analysts believe documentation describing the expected relationship between on-and-off-chain components is important for continued security of the protocol, especially if future development on the codebase occurs. As an example, much of the code in the database is designed to mimic the eventual behavior of the on-chain dispute resolution. However, the relationship between this logic off-chain and the on-chain counterparts is never explicitly stated; thus, subtle changes to either the on or off-chain components could lead to bugs with significant security implications.

Testing. As mentioned in an earlier section, there is currently no testing for this protocol. Thus, the Veridise team strongly recommends testing common behaviors as well as exceptional cases for at least the critical portions of the code, including the FPVM and smart contracts. Furthermore, Veridise auditors suggest running tests in CI/CD to prevent regressions. Veridise auditors believe adding testing in this way could help avoid a number of the high and critical issues we identified in this report (e.g., V-KLA-VUL-001).

Redesign considerations. The auditing team noticed there are a few design decisions for Kailua that introduce significant complexity into the system. It may be that these decisions are necessary to achieve desired performance, compatibility, or interface; however, the team wanted to explicitly point these out as they believe these could be sources of vulnerabilities in the future.

- **Inheritance hierarchy and use of OP interfaces:** The on-chain contracts have a somewhat complex inheritance/use hierarchy: all contracts are tournaments, there is one treasury that is also a tournament, and every game that is created is a tournament that also has a reference to the treasury. Furthermore, these contracts inherit from OP's `IDisputeGame` interface, which defines a number of constructs which are not quite applicable to the actual use-case in Kailua (such as the `GameStatus` enum, where the `CHALLENGER_WINS` status is never used).
- **Code/logic duplication:** A number of the bugs identified arise from a disconnect between off-chain logic and on-chain logic which are designed to compute the same thing (e.g. V-KLA-VUL-008). As much as possible, the Veridise team suggests querying contracts directly to avoid duplication of logic and guarantee consistent state on-and-off-chain.
- **Dispute resolution and player elimination:** In the audit, the Veridise team discovered a number of bugs that arose from (1) the ability of a proposer to propose as many proposals as they want with a single bond and (2) assumptions made about the order of challenge resolution and the state of a contender. As a result, we suggest the developers revisit these design decisions to simplify as much as possible. For example, is it necessary to have one bond cover infinite proposals until one is successfully disputed or can each proposal receive its own bond?

Out-of-scope code. During the review of the code, Veridise auditors noted that some behaviors that are of critical importance to the correctness/safety of the protocol are not in-scope. This includes interactions with Kona*, an external library used by the zkVM application for chain derivation and interactions with Zeth†, an additional external library used to populate the oracle with L1 information. Since these libraries contain complex logic essential for Kailua's proper functioning, and given that Kona is still under active development and is not recommended for production use, Veridise auditors recommend including these libraries in future security reviews.

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.

* `https://github.com/op-rs/kona`
† `https://github.com/risc0/zeth`

2 Project Dashboard

Table 2.1: Application Summary.

Name	Version	Type	Platform
Kailua Protocol	`6e2ce8f`	Soldity, Rust	Ethereum, RISC Zero

Table 2.2: Engagement Summary.

Dates	Method	Consultants Engaged	Level of Effort
Jan. 6–Jan. 28, 2025	Manual & Tools	3	9 person-weeks

Table 2.3: Vulnerability Summary.

Name	Number	Acknowledged	Fixed
Critical-Severity Issues	6	6	5
High-Severity Issues	7	7	7
Medium-Severity Issues	3	3	1
Low-Severity Issues	2	2	1
Warning-Severity Issues	4	4	3
Informational-Severity Issues	2	2	2
TOTAL	24	24	19

Table 2.4: Category Breakdown.

Name	Number
Logic Error	17
Denial of Service	3
Maintainability	2
Reentrancy	1
Authorization	1

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 and off-chain components. During the assessment, the security analysts aimed to answer questions such as:

- Are common smart contract pitfalls (reentrancy, ownership validation, etc) avoided?
- Are the smart contracts or off-chain components vulnerable to Denial-of-Service attacks?
- Can smart contract calls be frontrun or replayed?
- Can invalid state transitions be proven by the zkVM application?
- Can a valid proposal be successfully disputed?
- Can a user steal bond from another user without providing a successful dispute proof against that user?
- Are the inputs to the RISC Zero zkVM correctly validated?
- Can an invalid proposal maliciously prevent honest users from challenging it and thus become finalized without competition?
- Will the Proposer always post valid proofs?

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 technique:

- *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 following folders of the source code provided by the Kailua Protocol developers:

- `bin/`
  - `cli/`: All files except `bench.rs`, `fast_track.rs`, `fault.rs` and the `providers/` folder.
  - `client/`
  - `host/`
- `build/risczero/`: `fpvm/src/main.rs` and `src/lib.rs`
- `crates/common/src`
- `crates/common/contracts/src`, excluding the `vendor` folder

Methodology. Veridise security analysts read the provided Kailua Protocol documentation to understand the high-level goals of the code. They then began a review of the code assisted by static analyzers.

During the security assessment, the Veridise analysts regularly met with the Kailua Protocol developers to ask questions about the code.

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 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 4.1 summarizes the issues discovered:

Table 4.1: Summary of Discovered Vulnerabilities.

ID	Description	Severity	Status
V-KLA-VUL-001	Correct proposal can be . . .	Critical	Fixed
V-KLA-VUL-002	On-chain conversion to . . .	Critical	Fixed
V-KLA-VUL-003	Reentrancy allows . . .	Critical	Fixed
V-KLA-VUL-004	Malicious payout recipient . . .	Critical	Fixed
V-KLA-VUL-005	Early exit in proof . . .	Critical	Addressed and Partially Verified
V-KLA-VUL-006	Unchallengable proposal . . .	Critical	Fixed
V-KLA-VUL-007	Bonds cannot be recovered . . .	High	Fixed
V-KLA-VUL-008	Elimination inconsistency . . .	High	Fixed
V-KLA-VUL-009	Matches between duplicate . . .	High	Fixed
V-KLA-VUL-010	Malicious duplicate game . . .	High	Fixed
V-KLA-VUL-011	Proposal with skipped . . .	High	Fixed
V-KLA-VUL-012	Validators blocked from . . .	High	Fixed
V-KLA-VUL-013	Off-chain conversion to . . .	High	Fixed
V-KLA-VUL-014	Validators skip submitting . . .	Medium	Addressed and Partially Verified
V-KLA-VUL-015	Insecure key management	Medium	Addressed and Partially Verified
V-KLA-VUL-016	Not all fields included in . . .	Medium	Fixed
V-KLA-VUL-017	Missing conversion to field . . .	Low	Fixed
V-KLA-VUL-018	Network issues can crash . . .	Low	Addressed and Partially Verified
V-KLA-VUL-019	Proofs cannot be submitted . . .	Warning	Addressed and Partially Verified
V-KLA-VUL-020	Elimination round . . .	Warning	Fixed
V-KLA-VUL-021	KZG precompile field . . .	Warning	Fixed
V-KLA-VUL-022	Fetching blob returns . . .	Warning	Fixed
V-KLA-VUL-023	Unused program constructs	Info	Fixed
V-KLA-VUL-024	Small code suggestions	Info	Fixed

4.1 Detailed Description of Issues

4.1.1 V-KLA-VUL-001: Correct proposal can be rejected when disputed root is after the last block

Severity	Critical	**Commit**	6e2ce8f
Type	Logic Error	**Status**	Fixed
File(s)	`KailuaTournament.sol`		
Location(s)	prove		
Confirmed Fix At	f402ec3		

The `prove` function takes in the value `uvo[2]` which stores the index of the first root in the blobs at which the contender and opponent disagree. This index is *assumed* to be less than `PROPOSAL_BLOCK_COUNT` which is intended to indicate the number of state roots considered by a proposal but this is never enforced.

Impact If this value is made *greater* than the intended number of state roots, it can be used by a malicious actor to disprove a correct proposal by the malicious proposal taking advantage of the unused blob elements after the intended last root.

Recommendation Enforce that `uvo[2]` is less than `PROPOSAL_BLOCK_COUNT`.

Disclosure This issue was found by developers during the early part of the audit and was shared with the audit team. The team verified that this is an issue and included it in the report for completeness.

Developer Response The developers now check that `uvo[2]` is less than `PROPOSAL_BLOCK_COUNT`.

4.1.2 V-KLA-VUL-002: On-chain conversion to field elements is incorrect

Severity	Critical	**Commit**	6e2ce8f
Type	Logic Error	**Status**	Fixed
File(s)	`KailuaLib.sol`		
Location(s)	hashToFe		
Confirmed Fix At	5586e1d		

The function `hashToFe` is a simple helper function used to take a `uint256` value and covert it to a field element in the BLS12-381 field. As described in EIP-4844, blob elements are assumed to be elements in the scalar field for BLS12-381. The logic for `hashToFe` is shown below.

```
function hashToFe(bytes32 hash) internal pure returns (bytes32 fe) {
    fe = ((hash << 2) >> 2);
}
```

Snippet 4.1: Implementation of `hashToFe`

As one can see, this converts a 32 byte value to a field element `fe` by zeroing out the first two bits (i.e., leaving `254` bits to represent the value). However, it turns out the scalar field prime for BLS12-381 is *greater than* $2^{254} - 1$. This means that for any field element x (represented as a `bytes32`) where $x > 2^{254} - 1$ there is another field element $x' = x \ mod \ 2^{254}$ such that `hashToFe(x) = hashToFe(x')`.

Impact This helper function is used in two places and can thus lead to two different exploits.

Rejecting valid proposals First, it is used in the `prove` function of `KailuaTournament` when comparing the proposed output of a competitor with the actual computed output by the circuit. In particular, the logic is shown below.

```
if (proposedOutput[0] == proposedOutput[1]) {
    revert NoConflict();
}
...
if (KailuaLib.hashToFe(proposedOutput[0]) != KailuaLib.hashToFe(computedOutput)) {
    // u lose
    if (KailuaLib.hashToFe(proposedOutput[1]) != KailuaLib.hashToFe(computedOutput))
    {
        // v lose
        proofStatus[uvo[0]][uvo[1]] = ProofStatus.U_LOSE_V_LOSE;
    } else {
        // v win
    proofStatus[uvo[0]][uvo[1]] = ProofStatus.U_LOSE_V_WIN;
    }
} else {
    // u win
    proofStatus[uvo[0]][uvo[1]] = ProofStatus.U_WIN_V_LOSE;
}
```

Snippet 4.2: Comparison logic in `prove`

As shown above, a challenge is only accepted if the two competitors disagree on the proposed output. However, an attacker can use this vulnerability to suggest a different proposed output from the *correct* proposal by first finding some output that is greater than $2^{254} - 1$, computing $x' = x \ mod \ 2^{254}$, and proposing that. In that case, their proposed output is not equal to the contender's proposed output *but* the check `KailuaLib.hashToFe(proposedOutput[0]) != KailuaLib.hashToFe(computedOutput)` will be true.

Unverifiable blobs In the `verifyKZGBlobProof` function, `hashToFe` is used to normalize a blob value before performing the KZG check in the following call.

```
bytes memory kzgCallData = abi.encodePacked(
    versionedHash, // proposalBlobHash().raw(),
    rootOfUnity,
    hashToFe(value),
    blobCommitment,
    proof
);
(success,) = KZG.call(kzgCallData);
```

Snippet 4.3: Implementation of `hashToFe`

As one can see, this converts the provided `value` (which is a `bytes32` representing a field element) to a field element using `hashToFe`. If this `value` is a field element that is greater than $2^{254} - 1$, than this will call will change the element to a different field element that will no longer pass the check.

Recommendation Replace `hashToFe` with a proper conversion to a scalar field element.

Developer Response The developers have added the proper conversion.

4.1.3 V-KLA-VUL-003: Reentrancy allows unchallengable proposal

Severity	Critical	**Commit**	6e2ce8f
Type	Reentrancy	**Status**	Fixed
File(s)	`KailuaTournament.sol`, `KailuaTreasury.sol`		
Location(s)	pruneChildren, eliminate		
Confirmed Fix At	5586e1d		

The following `pay` function in `KailuaTreasury` is called when a proposal is successfully eliminated (via a call to `eliminate`) and sends the funds to the recipient. The external call can be used as a reentry point by a malicious user who was registered as the payout `recipient` for a successful proof.

```
function pay(uint256 amount, address recipient) internal {
    (bool success,) = recipient.call{value: amount}(hex"");
    if (!success) revert BondTransferFailed();
}
```

Snippet 4.4: Implementation of `pay`

`eliminate` is called in the `pruneChildren` function in `KailuaTournament` when either a contender or an opponent is shown to be invalid. This call occurs in the following loop.

```
KailuaTournament contender = children[u];
for (; v < children.length && eliminationLimit > 0; (v++, eliminationLimit--)) {
    KailuaTournament opponent = children[v];
    ...
    ProofStatus proven = proofStatus[u][v];
    require(proven != ProofStatus.NONE);
    if (proven == ProofStatus.U_LOSE_V_WIN) {
        KAILUA_TREASURY.eliminate(address(contender), prover[u][v]);
        u = v;
        contender = opponent;
    } else {
        KAILUA_TREASURY.eliminate(address(opponent), prover[u][v]);
    }
}
contenderIndex = u;
opponentIndex = v;
```

Snippet 4.5: Calling `eliminate` in `prove` (some checks and comments have been elided)

An attacker can use this reentrant point in `eliminate` to reenter `pruneChildren` and critically to reset the `contenderIndex` to point to a contender who has already been eliminated. Once this happens, a contender can never be eliminated and thus will eventually either become the finalized proposal even though they are incorrect if there are no challenges *or* will block the protocol if there are challenges.

Illustrative Example To better understand how this attack can occur, consider the following example. Let us suppose that the protocol has been initialized (so the treasury is deployed, it has been resolved, the Kailua Game has been appropriately setup in the dispute game factory, etc.). Furthermore, let us suppose that `propose` has been called three times to create children `c0`,

`c1`, and `c2` (in that order) and that they were proposed by users `u1`, `u2`, and `u3` respectively. Let us also suppose two proofs have been submitted: one for `(c0, c1)` that successfully shows that `c1` wins and one for `(c1, c2)` that shows that `c2` wins. Let us suppose the first proof for `(c0, c1)` was proved by an attacker who set their own malicious attack contract `A` as the payout recipient for the proof.

Now, suppose we call `pruneChildren` with an `eliminationLimit` of `1`. Execution of the function starts by setting `u` to `0` and `v` to `1`. It then enters the loop, where it fetches the proof status for `(c0, c1)` and upon learning that `c0` lost, calls `eliminate` on `c0` specifying `A` as the payout recipient. Executing `eliminate` performs three checks and then calls `pay`:

```
// INVARIANT: Only the child's parent may call this
KailuaTournament parent = child.parentGame();
if (msg.sender != address(parent)) {
    revert Blacklisted(msg.sender, address(parent));
}
// INVARIANT: Only known proposals may be eliminated
address eliminated = proposerOf[address(child)];
if (eliminated == address(0x0)) {
    revert NotProposed();
}
// INVARIANT: Cannot double-eliminate players
if (eliminationRound[eliminated] > 0) {
    revert AlreadyEliminated();
}

// Record elimination round
eliminationRound[eliminated] = child.gameIndex();
// Transfer bond payment to the game's prover
pay(paidBonds[eliminated], prover);
```

Snippet 4.6: Snippet from `eliminate`

As indicated in the comments, we check that the caller is the parent (which it is when it is called in `pruneChildren`), the proposal is known (which is true of all proposals considered when pruning), and the proposer has not already been eliminated (which is true as `u0` has not yet been eliminated). In this case, all checks pass, `u0` is registered as eliminated, and `pay` is called with `A` as the recipient. Let us suppose `A` has the following simple fallback that calls `pruneChildren` again with an elimination limit of `2`.

```
fallback() external payable {
    KAILUA_GAME.pruneChildren(2);
}
```

Snippet 4.7: Example attack contract `fallback`.

When invoking `pruneChildren` from the fallback, the following logic is triggered to initialize the contender `u` and opponent `v`.

```
// Resume from last surviving contender
uint64 u = contenderIndex;
if (u == 0) {
    // Select the first possible contender
    for (; u < children.length && eliminationLimit > 0; (u++, eliminationLimit--)) {
```

```
        if (!isChildEliminated(children[u])) {
            break;
        }
    }
}
// Resume from last unprocessed opponent
uint64 v = opponentIndex;
if (v == 0) {
    // Select first possible opponent
    v = u + 1;
}
```

Snippet 4.8: Snippet from `pruneChildren`.

Again, `u` is initially set to `0`. However, The loop will now indicate that `isChildEliminated(children[0])` is `true` because `c0` was eliminated at the end of the `eliminate` function before `pay` was called in the first invocation of `pruneChildren`. Thus, `u` becomes `1` and `v` becomes `2` and the `eliminationLimit` is dropped to `1`. Now, the loop executes, the proof for `(c1, c2)` is fetched, and it is found that `c2` won so `c1` is eliminated. We suppose the payout recipient is normal for this one and note that elimination succeeds because `u1` (the proposer of `c1`) has not yet been eliminated. Thus elimination of `c1` succeeds. After this, `u` is set to `2` and the new contender to be the opponent. Now, because the `eliminationLimit` has dropped to `0`, the loop exits and the `contenderIndex` is set to `u` which is `2` and the `opponentIndex` is set to `v` which is `3`. This ends the function and thus ends execution of the fallback within `A`.

Back in the original execution, we have finished processing the elimination of `c0`. We set `u` to be `1` and the new contender to be the opponent (in this case, the opponent is `c1` so `c1` is now the new contender). Now, because the `eliminationLimit` has dropped to `0`, the loop exits and the `contenderIndex` is set to `u` which is `1` and the `opponentIndex` is set to `v` which is `2`. After this, the function finishes executing.

Now we are in the state where the `contenderIndex` is `1` which refers to the child `c1` whose proposer `u1` was eliminated via the reentrant call. `c1` is known to be invalid (as it was already disproven by `c2`). However, suppose we try to get `c1` eliminated. To do so, we call `pruneChildren(1)`. `u` is initialized to `1` (and the loop to update it is not invoked because `u` is not `0`). `v` is initialized to `2`. We enter the loop and again fetch the proof status for `(c1, c2)`. On learning `c1` lost, we attempt to eliminate its proposer `u1`. However, because the proposer for `c1` has already been eliminated, the call to `eliminate` fails and thus the transaction reverts.

In general, now that `c1` is the contender but is already eliminated, any future attempt to eliminate them will fail.

We have included a proof-of-concept foundry test for a simplified version of the contracts here for reference as well. Just run `forge test` to run the test case that shows the exploit.

reentrancy-poc.zip

Impact As shown through the example above, this can allow an invalid proposal to block progress indefinitely, leading to a denial of service.

On a less serious note, a very similar reentry can also be used to resolve the same claim twice. While this will not do much, it will result in potentially multiple emissions of the `Resolved` event which could mislead 3rd party applications.

Recommendation To disallow the particular reentrancy attack outlined, we suggest that reentrancy guards are used to disallow reentry into the `pruneChildren` function.

More generally, we suggest careful consideration of reentrancy as a possible concern. While this is the only specific attack we have discovered, the protocol makes numerous cross-contract calls with a variety of (often unstated) assumptions. We strongly suggest careful consideration of whether these are all necessary and, if so, abundant use of reentrancy guards including possible cross-contract protection when necessary.

Developer Response The developers do not send payments in the `eliminate()` function, and the new `claimEliminationBonds()` function is reentrancy protected. Thus, the immediate reentrancy concern is resolved. However, this change was made alongside a number of functional changes to the logic that could introduce other issues. Veridise strongly recommends to developers that the new logic be audited in its entirety to ensure no new or related issues were introduced and has thus marked the issue as "partially fixed".

4.1.4 V-KLA-VUL-004: Malicious payout recipient causes DoS

Severity	Critical	**Commit**	6e2ce8f
Type	Denial of Service	**Status**	Fixed
File(s)	`KailuaTournament.sol`, `KailuaTreasury.sol`		
Location(s)	pruneChildren, eliminate		
Confirmed Fix At	27dec82		

When submitting a proof, the creator of the proof can choose whichever address they want to receive the payout for the elimination. The payout recipient is sent the funds during calls to `eliminate` in the treasury via a call to the following function.

```
function pay(uint256 amount, address recipient) internal {
    (bool success,) = recipient.call{value: amount}(hex"");
    if (!success) revert BondTransferFailed();
}
```

Snippet 4.9: Implementation of `pay`

`pay` is called by `eliminate` which is called in the `pruneChildren` function in `KailuaTournament` when either a contender or an opponent is shown to be invalid and thus can be eliminated. Because the call to `pay` will revert if the call reverts, a malicious payout recipient that always reverts will block successful execution of `pruneChildren` and will block successful execution of the function.

Impact This can block the protocol indefinitely by making it impossible to resolve a correct proposal.

Recommendation There are a few ways to avoid the DoS risk here. One is to not check the success status of the call to the recipient. However, this could result in lost funds for a user. Another option is to make a separate function that allows users to claim rewards at a later time - this solution avoids ignoring the success status but could lead to other unexpected vulnerabilities if not implemented carefully.

Developer Response The developers have moved payout functionality to a function that requires a separate call, and therefore calls to the `msg.sender` account are not done in the `eliminate()` flow.

4.1.5 V-KLA-VUL-005: Early exit in proof generation enables fault proof against an honest proposal

Severity	Critical	**Commit**	6e2ce8f
Type	Logic Error	**Status**	Addressed and Partially Verified
File(s)	`crates/common/src/client.rs`		
Location(s)	run_client		
Confirmed Fix At	871eb34		

In the off-chain code, the following check is performed to early exit *before* doing validation if the claimed output root is equal to the agreed L2 output root.

```
if boot.agreed_l2_output_root == boot.claimed_l2_output_root {
    return Ok((precondition_hash, Some(boot.claimed_l2_output_root)));
}
```

Snippet 4.10: Snippet from `run_client()`

This allows anyone to offer a proof that simply copies the agreed root into the next index that will be able to disprove the actual valid proposal.

Impact This allows honest proposals to be disproved and invalid ones to take their place.

Recommendation Remove this check.

Disclosure This issue was found by developers during the early part of the audit and was shared with the audit team. The team verified that this is an issue and included it in the report for completeness.

Developer Response The developers have removed the check and thus the immediate issue has been resolved. However, this change was made alongside a number of functional changes to the logic that could introduce other issues. Veridise strongly recommends to developers that the new logic be audited in its entirety to ensure no new or related issues were introduced and has thus marked the issue as "addressed and partially verified".

4.1.6 V-KLA-VUL-006: Unchallengable proposal arises from out-of-order eliminations

Severity	Critical	**Commit**	6e2ce8f
Type	Logic Error	**Status**	Fixed
File(s)	`KailuaTournament.sol`		
Location(s)	pruneChildren		
Confirmed Fix At	871eb34		

When eliminating a proposal, its proposer will also be eliminated. This is done by storing an `eliminationRound` mapping containing an entry for every eliminated proposer. Each such entry stores the game index of the proposal responsible for a proposer's elimination. This is assumed to be the first dishonest proposal made by that proposer.

```
// Record elimination round
eliminationRound[eliminated] = child.gameIndex();
```

Snippet 4.11: Snippet from `KailuaTreasury.eliminate(...)`

When considering child proposals as potential contenders, a check is done to ensure that their proposer was not already eliminated prior to making that proposal. This behavior is reflected in the following function.

```
function isChildEliminated(KailuaTournament child) internal returns (bool) {
  address _proposer = KAILUA_TREASURY.proposerOf(address(child));
  uint256 eliminationRound = KAILUA_TREASURY.eliminationRound(_proposer);
  if (eliminationRound == 0 || eliminationRound > child.gameIndex()) {
      // This proposer has not been eliminated as of their proposal at gameIndex
      return false;
  }
  return true;
}
```

Snippet 4.12: Implementation of `isChildEliminated`

As shown, a child is considered "eliminated" if the "elimination round" for its proposer is less than or equal to the current child's index and is non-zero. Later on, if a child is eliminated, the `eliminate` function on the treasury is called, and only succeeds if the proposer of the child is not already eliminated per the following check.

```
// INVARIANT: Cannot double-eliminate players
if (eliminationRound[eliminated] > 0) {
  revert AlreadyEliminated();
}
```

Snippet 4.13: Check in `eliminate`

If `eliminate` fails this check and reverts, the whole call to `pruneChildren` will revert.

The key issue here is that it is possible for the treasury to eliminate a proposer `p` at an elimination round `r` while also having an incorrect proposal `c` made by `p` with index `i < r` as a child of another game. If proposal `c` later becomes a contender in the canonical chain, the protocol will become stuck.

Illustrative Example Consider the following example scenario showing how this can be exploited.

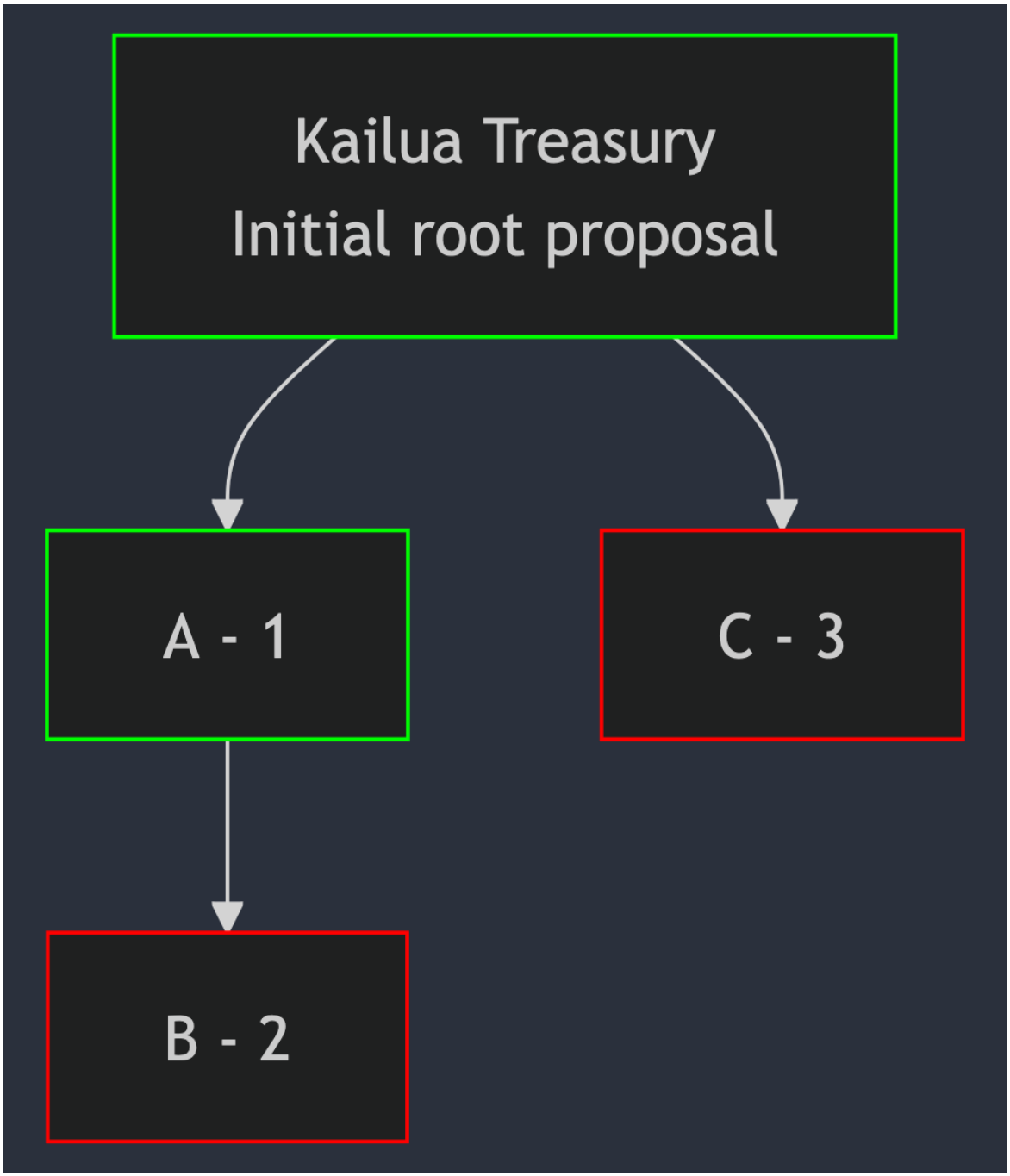
Actors

- Alice - an honest proposer
- Mallory - a dishonest proposer

Chain of Actions

1. The Kailua treasury is initialized and resolved trivially
2. Alice submits a proposal `A`, which is an honest proposal (it cannot be proven wrong, and will be accepted after the challenge period)
3. Mallory submits a proposal `B`, as a *child* of `A`. `B` is a dishonest proposal and can be proven wrong with the FPVM.
4. Mallory submits a proposal `C`, as a *sibling* of `A`. `C` is also a dishonest proposal that can be proven wrong.
5. `pruneChildren` is called on the treasury.

The resulting state of the chain is shown in the diagram below. Honest proposals are outlined in green, dishonest proposals are outlined in red. The numbers in the nodes correspond to that proposal's index in the dispute game factory (DGF).



Consequences When calling `pruneChildren` on the root, proposal `C` will be proven to be faulty and as a result its proposer Mallory will be eliminated. The treasury will note the `eliminationRound` to be the index of `C` in the DGF, i.e. 3.

However, since Mallory submitted proposal `B` before `C`, it will not be ignored from any tournaments, since its index in the DGF will be lower than the stored `eliminationRound` for Mallory.

But proposal `B` itself is dishonest, and therefore should be proven faulty when trying to prune the children of `A`.

This can however **never** happen, as its proposer has already been eliminated (`eliminationRound > 0`). As a result, the system can never progress.

Impact This can cause `pruneChildren` to become stuck indefinitely and progress of the protocol to become halted.

Recommendation A simple fix here is not obvious to us. One possible option is to remove the check in `eliminate` and thus allow "double elimination" of a user. This would avoid deadlock but would require people be willing to disprove dishonest proposals for no reward. In general, any fix here should avoid the possibility that an eliminated contender blocks progress of the protocol.

Developer Response The developers made it so that proposals must be submitted by increasing L2 block number which blocks this attack which relies on out-of-order proposal submissions.

4.1.7 V-KLA-VUL-007: Bonds cannot be recovered for honest actors

Severity	High	**Commit**	6e2ce8f
Type	Logic Error	**Status**	Fixed
File(s)	`KailuaTreasury.sol`		
Location(s)	propose		
Confirmed Fix At	bd76b76		

Users can submit a claim by calling `propose` on the `KailuaTreasury` contract. The logic shown below is used to ensure that proposals can only be made if the user includes enough bonds to exceed the threshold `participationBond`.

```
// Update proposer bond
if (msg.value > 0) {
  paidBonds[msg.sender] += msg.value;
}

// Check proposer bond
if (paidBonds[msg.sender] < participationBond) {
    revert IncorrectBondAmount();
}
```

Snippet 4.14: Snippet from `propose()`

When an invalid claim is eliminated via the `eliminate` function, the bonds associated with the user whose claim was invalid are sent to the user who proved the claim was invalid. However, in the event that the claim is successful and accepted, the user cannot reclaim their bond even if they have *no* remaining open claims. Furthermore, not only can the user not recover the funds, even of the owners of the contract cannot withdraw the funds, meaning they are simply stuck in the contract.

Impact Honest users will lose their bonds and eventually potentially large amounts of funds will become locked in the treasury.

Recommendation Allow users to withdraw their bonds if they have no outstanding games.

Developer Response The developers have implemented a `claimProposerBond()` function, which will allow a user to claim their bond once their last proposals is resolved. This allows an honest proposer to safely withdraw their bond if they have no additional pending proposals after that proposal and critically still allows them to withdraw even if some of the honest proposer's proposals were skipped. However, the logic does require an honest proposer to potentially continue to propose after they want to withdraw their bond until eventually their proposal is the one that is resolved as the winner for that round. This delay might not be ideal for honest proposers who want to quickly withdraw their funds.

4.1.8 V-KLA-VUL-008: Elimination inconsistency off-chain leads to blocked proposer

Severity	High	**Commit**	6e2ce8f
Type	Denial of Service	**Status**	Fixed
File(s)	`bin/cli/src/db/mod.rs`		
Location(s)	determine_tournament_participation		
Confirmed Fix At	871eb34		

The code in the Kailua database is meant to simulate the progress of proofs on-chain off-chain so that the proposer logic can track the correct "canonical" proposal and suggest new proposals as descendants of canonical proposals. There is an assumption built into this logic that the "canonical" proposal off-chain should be the same proposal that (eventually) wins the corresponding on-chain tournament. The code attempts to ensure this by simulating the challenges that will occur on-chain. While this simulation is mostly accurate, it does miss a key behavior that is present on-chain that can cause inconsistencies. In particular, consider the following check from `pruneChildren` on the on-chain contracts.

```
// If the contender hasn't been challenged for as long as the timeout, declare them
    winner
if (contender.getChallengerDuration(opponent.createdAt().raw()).raw() == 0) {
  // Note: This implies eliminationLimit > 0
  break;
}
```

Snippet 4.15: Snippet from `pruneChildren()`

This check happens *before* any player elimination happens and early exits out of the tournament loop if enough time has passed in between opponents to simply declare the contender the winner with no proof necessary. Off-chain, no such check exists.

Impact This can result in the `eliminations` off-chain being out-of-sync with the on-chain `eliminationRound` mapping. Because proposals are filtered based on their elimination status, this can cause a proposal that is "canonical" on-chain to be ignored off-chain, ultimately leading to a canonical proposal off-chain that is unresolvable.

Illustrative Example To better understand this attack, consider the following steps that happen (in the order described):

1. The Kailua treasury is initialized and resolved.
2. Alice submits an honest proposal `A` as a child of the treasury.
3. `MAX_CLOCK_DURATION` amount of time passes.
4. Bob submits a dishonest proposal `B` as a child of the treasury.
5. Bob submits an honest proposal `C` as a child of `A`.
6. Alice submits an honest proposal `D` as a child of `A`.

On-chain, `A` is chosen as the winner of the treasury's tournament. `A` is also considered the canonical proposal for this tournament off-chain. On-chain, because `B` was submitted *after*

`MAX_CLOCK_DURATION` passed, `B` is *not* eliminated. However, off-chain, `B`'s proposer is eliminated (i.e., Bob is added to `eliminations`) because there is no such check.

Now, for the tournament under `A`, on-chain, `C` is the contender and (because it is honest) will not be able to be defeated. Off-chain however will be different. Because Bob is the sender of `C` and Bob is eliminated, `C` is skipped and `D` becomes the canonical proposal. The proposer will now continually try to resolve `D` as the canonical winner of the tournament under `A` but will never be able to succeed.

Recommendation Add the same check for time between challenges to the off-chain logic.

Developer Response The developers have added the timeout check to the off-chain database.

4.1.9 V-KLA-VUL-009: Matches between duplicate proposals off-chain leads to crashed validator

Severity	High	**Commit**	6e2ce8f
Type	Logic Error	**Status**	Fixed
File(s)	`bin/cli/src/db/mod.rs`		
Location(s)	determine_tournament_participation		
Confirmed Fix At	f402ec3		

When loading proposals from the Dispute Game Factory (DGF), the database (DB) internally tracks which conflicting pairs of proposals will eventually need to be settled via a dispute game on-chain. However, there are some inconsistencies in the off-chain logic compared to the smart contracts. Notably, there is no check to avoid matching proposals against duplicate versions of them.

In the `KailuaTournament.pruneChildren(...)` function, a number of checks are done in order to avoid playing out dispute games between proposals in certain scenarios. For example, two proposals from the same proposer should never challenge each other. Additionally, a proposal should never challenge a duplicate of itself.

```
// If the opponent proposal is an identical twin, skip it
if (contender.rootClaim().raw() == opponent.rootClaim().raw()) {
    uint64 common;
  for (common = 0; common < PROPOSAL_BLOBS; common++) {
    if (contender.proposalBlobHashes(common).raw() != opponent.proposalBloHashes(
    common).raw()) {
      break;
    }
  }
  if (common == PROPOSAL_BLOBS) {
    // The opponent is an unjustified duplicate proposal. Ignore it.
    continue;
  }
}
```

Snippet 4.16: Snippet from `pruneChildren(...)` function in the `KailuaTournament` contract.

In the off-chain component similar checks are performed when loading proposals from the DGF into the DB (such as checking for self-conflicts of a given proposer). These checks are performed in the `determine_tournament_participation` function. The result of this function will determine whether or not a given proposal will be processed by the proposer and validator agents. However, one notable omission from this function is a check for duplicate proposals. These will therefore **not** be omitted from the tournament.

The validator's primary function is to generate proofs for proposals that are participating in the tournament against their `contender` field. This `contender` field mirrors the state of the `KailuaTournament`'s `contenderIndex` at the time of a proposal's consideration in `pruneChildren`. A proof will be requested between `proposal` and `contender` whenever `proposal` is determined to participate in the tournament.

However, before requesting a proof the validator performs a number of checks on the state of `proposal` and `contender`. One such check is to find a divergence point between the two.

Crucially if a divergence point is not found, the function will **panic** due to the use of the `expect()` function.

```
let challenge_point = contender
    .divergence_point(proposal)
    .expect("Contender does not diverge from proposal.") as u64;
```

Snippet 4.17: Snippet from the `request_proof(...)` function in the validator logic.

If two proposers submit proposals `A` and `B`, where `A` and `B` are both identical copies of the "correct" proposal. They can be sure that when loading in proposal `B`, its `contender` field will point to a duplicate of itself (either `A`, or another version of the "correct" proposal submitted before `A`). The validator will then request a proof for `B` and its duplicate, leading to a crash when it cannot find a divergence point between the two proposals.

Impact As shown above, the vulnerability can crash any validator instance.

Recommendation Patch the off-chain component to correctly mirror the logic of the on-chain component.

Developer Response The developers now return `Ok(false)` from `determine_tournament_participation` when the proposers are the same, or there is no divergence point.

4.1.10 V-KLA-VUL-010: Malicious duplicate game can block proposer progress

Severity	High	Commit	6e2ce8f
Type	Logic Error	Status	Fixed
File(s)	bin/cli/src/propose.rs		
Location(s)	propose		
Confirmed Fix At	871eb34		

When constructing a new dispute game, the following special logic is needed to compute a "duplication counter" that acts as a nonce to avoid conflict with previously deployed games with the same information.

```
let unique_extra_data = loop {
    // compute extra data with block number, parent factory index, and blob hash
    let extra_data = [
      proposed_block_number.abi_encode_packed(),
      canonical_tip.index.abi_encode_packed(),
      dupe_counter.abi_encode_packed(),
    ]
    .concat();
    // check if proposal exists
    let dupe_game_address = dispute_game_factory
      .games(
        KAILUA_GAME_TYPE,
        proposed_output_root,
        Bytes::from(extra_data.clone()),
      )
      .stall()
      .await
      .proxy_;
    if dupe_game_address.is_zero() {
      // proposal was not made before using this dupe counter
      break Some(extra_data);
    }
    // fetch proposal from local data
    let dupe_game_index: u64 = KailuaTournament::new(dupe_game_address, &
    proposer_provider)
      .gameIndex()
      .stall()
      .await
      ._0
      .to();
    let Some(dupe_proposal) = kailua_db.get_local_proposal(&dupe_game_index) else {
      // we need to fetch this proposal's data
      break None;
    };
    // check if proposal was made incorrectly or by an already eliminated player
    if dupe_proposal.is_correct().unwrap_or_default()
      && !kailua_db.was_proposer_eliminated_before(&dupe_proposal)
    {
      break None;
    }
    // increment counter
```

```
41     dupe_counter += 1;
42 }
```

Snippet 4.18: Snippet from `propose()`

This logic works by iteratively incrementing the `dupe_counter` starting from `0` until no duplicate is found. However, you will notice two cases with `break None` that can stop this iteration before finding such a duplication counter. These capture the case when the correct proposal was already submitted (by a non-eliminated player) so we simply need to wait to process that correct proposal. However, these cases (in particular, the first case) can lead to trouble.

The problem here is that not all proposals are saved to the local database and thus the call `kailua_db.get_local_proposal(&dupe_game_index)` may always return `None` even after the relevant duplicate proposal is processed.

Illuminating Example To better understand, consider the following example. Suppose Alice is malicious and wants to break the proposer. She first submits an incorrect proposal and then immediately thereafter submits the correct proposal. Because the "contender" for the second proposal has the same proposer, the function `determine_tournament_participation` in the database logic will not save the second proposal to the database. However, because it is correct and thus uses the same arguments to the dispute game factory as the proposer will try, as long as they used the duplication counter `0`, it will block the correct proposal from being submitted by the proposer.

Impact This allows users to block proposer progress and could allow them to corrupt the chain without manual intervention.

Recommendation Ensure that the proposer can always make progress, even in the presence of duplicate proposals.

Developer Response The developers have added an extra check that will break if the game index has not been processed yet in the database. Otherwise, we only break if the duplicate proposal is stored in the database, is correct, and has a previously un-eliminated proposer. This handles the issue as proposals which were skipped no longer block progress to considering new duplication counters.

4.1.11 V-KLA-VUL-011: Proposal with skipped parent crashes proposer

Severity	High	**Commit**	6e2ce8f
Type	Logic Error	**Status**	Fixed
File(s)	bin/cli/src/db/mod.rs		
Location(s)	determine_correctness		
Confirmed Fix At	f1c7e42		

In `determine_correctness`, when determining the correctness of a given proposal, the code first fetches information about the proposal's parent and checks if the parent is correct via the following code.

```
let is_parent_correct = self
  .get_local_proposal(&proposal.parent)
  .expect("Attempted to process child before registering parent.")
  .is_correct()
  .expect("Attempted to process child before deciding parent correctness");
```

Snippet 4.19: Snippet from `determine_correctness()`

This logic first attempts to fetch the parent from the local database and then checks if that parent is correct via the `is_correct()` call. The issue here is the first call to `get_local_proposal` and the subsequent `expect` which will panic if the proposal's parent is not found. It is possible for this to happen as some proposals are not added to the local database, such as proposals whose contender has the same proposer. For these proposals, they will not be saved to the local database but there can be a subsequent proposal that chooses one of these "ignored" proposals as a parent.

Impact If this case occurs, proposers trying to process that proposal will crash.

Recommendation Appropriately account for the possibility of a proposal with a parent that was ignored and is thus not in the local database.

Developer Response The developers now interpret missing parent's as a `false` value in both `determine_correctness()` and `determine_tournament_participation()`.

4.1.12 V-KLA-VUL-012: Validators blocked from submitting valid proofs due to inconsistency in child index accounting

Severity	High	**Commit**	6e2ce8f
Type	Logic Error	**Status**	Fixed
File(s)	`bin/cli/src/db/mod.rs`		
Location(s)	determine_tournament_participation		
Confirmed Fix At	f1c7e42		

In `determine_tournament_participation`, a child is appended to a proposal's parent only if they pass certain checks, such as that they are not proposed by the current contender and that their proposer has not already been eliminated. However, this is not consistent with on-chain logic, where these proposals are still appended as children but are just skipped in the proof. This inconsistency means that the following code in the validator can return the wrong child indices in cases when proposals are skipped.

```
let u_index = proposal_parent
  .child_index(contender_index)
  .expect("Could not look up contender's index in parent tournament");
let v_index = proposal_parent
  .child_index(proposal.index)
  .expect("Could not look up contender's index in parent tournament");
```

Snippet 4.20: Snippet from `handle_proposals()`

These indices are passed directly to the `prove` function on-chain which is used to know which two proposals to compare. Because the indices are inconsistent with those on-chain, the call will fail despite having an otherwise valid proof.

Impact This can cause validators to be unable to submit valid proofs in relatively common scenarios. Someone can even maliciously abuse this to ensure that validators always fail.

Recommendation Ensure on and off chain accounting of child indices are consistent.

Developer Response The developers now append the child in the database, as long as it has a parent.

4.1.13 V-KLA-VUL-013: Off-chain conversion to field elements is incorrect

Severity	High	**Commit**	6e2ce8f
Type	Logic Error	**Status**	Fixed
File(s)	`crates/common/src/blobs.rs`		
Location(s)	hash_to_fe		
Confirmed Fix At	5586e1d		

V-KLA-VUL-002 reported an issue with the on-chain contracts where the conversion of values to field elements was incorrect and could lead to exploitable conflicts. The same issue occurs in the off-chain component of the protocol, where the same logic is duplicated in the following function

```
pub fn hash_to_fe(mut hash: B256) -> B256 {
    hash.0[0] &= u8::MAX >> 2;
    hash
}
```

Snippet 4.21: Implementation of `hash_to_fe`

Impact This function is used in both the proposer and validator, which means at times both of these may incorrectly encode values as field elements. This incorrect encoding can lead to crashes and possible conflicts.

Recommendation Fix the implementation of `hash_to_fe` to appropriately compute the value mod the prime.

Developer Response The developers added the appropriate computation.

4.1.14 V-KLA-VUL-014: Validators skip submitting proofs on error

Severity	Medium	**Commit**	6e2ce8f
Type	Logic Error	**Status**	Addressed and Partially Verified
File(s)	`bin/cli/src/validate.rs`		
Location(s)	handle_proposals		
Confirmed Fix At	871eb34		

The function `handle_proposals` is intended to process submitted proposals, create proofs of contender/proposal pairs, and submit those proofs on-chain. This loop is important in that it allows tournaments to progress and ultimately proposals of blocks to be resolved. The function works by processing new proposals as they are submitted on-chain, detecting when proofs are needed, constructing those proofs, and submitting them. One important aspect that is not considered are failed proof submissions. In particular, each proof submission is only attempted once - if anything goes wrong, the validator simply skips that proof and continues on. For example, a proof may be correct but a connection issue with the on-chain Kailua contract may fail; in this case, the validator would simply skip this proof. As a result, to make progress on-chain, another validator would need to process the proof *or* a proof would need to be submitted manually.

Impact This can result in the chain getting stuck until either another validator or a manual user provide a proof to progress a tournament.

Recommendation Add some retrying logic for failure cases in the validator.

Developer Response The developers have introduced a queue where proposals-to-be-submitted are stored. Proposals are added back to this queue on failure enabling retrying of failed proposal submissions.

This fix solves the direct issue raise by the auditors. However, the Veridise team also notes that this change was made alongside a number of functional changes to the logic that could introduce other issues. Veridise strongly recommends to developers that the new logic be audited in its entirety to ensure no new or related issues were introduced and has thus marked the issue as "addressed and partially verified".

4.1.15 V-KLA-VUL-015: Insecure key management

Severity	Medium	**Commit**	6e2ce8f
Type	Authorization	**Status**	Addressed and Partially Verified
File(s)	`bin/cli/src/proposer.rs`, `bin/cli/src/validator.rs`		
Location(s)	N/A		
Confirmed Fix At	`https://github.com/risc0/kailua/pull/21`		

Both the proposer and validator store a private key in plaintext as part of their respective argument structs. Storing private keys in plaintext is dangerous and can potentially risk stolen keys.

Impact Stolen private keys can potentially be very bad depending on what those keys are used for. In this case, it appears the private keys are intended to have access to funds - thus stolen keys could lead to lost funds.

Recommendation We recommend not storing keys in plaintext. Instead, rely on safer industrial solutions for key management.

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:

1. Never storing a protocol key in plaintext, on a regularly used phone, laptop, or device, or relying on a custom solution for key management.
2. Using separate keys for each separate function.
3. Storing multi-sig keys in a diverse set of key management software/hardware services and geographic locations.
4. 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.
5. Validating that no party has control over multiple multi-sig keys.
6. Performing regularly scheduled key rotations for high-frequency operations.
7. Securely storing physical, non-digital backups for critical keys.
8. Actively monitoring for unexpected invocation of critical operations and/or deployed attack contracts.
9. Regularly drilling responses to situations requiring emergency response such as pausing/unpausing.

Developer Response The developers have added additional options for using AWS or GCP key management. They did still keep the option to provide keys as plaintext. The Veridise team suggests users *do not* use the plaintext option and instead always opt for using either AWS or GCP.

The Veridise team also notes that this change was made alongside a number of functional changes to the logic that could introduce other issues. Veridise strongly recommends to developers that the new logic be audited in its entirety to ensure no new or related issues were introduced and has thus marked the issue as "addressed and partially verified".

4.1.16 V-KLA-VUL-016: Not all fields included in rollup hash

Severity	Medium	**Commit**	6e2ce8f
Type	Logic Error	**Status**	Fixed
File(s)	`crates/common/src/client.rs`		
Location(s)	config_hash		
Confirmed Fix At	bd76b76		

The function `config_hash` is used to hash the configuration options chosen for the rollup that is included in the journal and is thus used to ensure consistency with on-chain contracts. The function essentially takes all of the configuration options, converts them to bytes, concatenates them together, and hashes them. However, the following fields are missing from the computation:

- From the system config, the fields `eip1559_denominator` and `eip1559_elasticity`
- From the rollup config, the fields `genesis.l2_time` and `isthmus_time`

Impact If these fields can change the rollup behavior in meaningful ways, this could mean two different rollups would have the same hash. This could lead to proofs for one rollup being incorrectly used for a different rollup.

Recommendation Add all relevant fields to the computation of the hash.

Additionally, we also recommend the use of a domain separator between entries in the hash to avoid conflicts. We do not believe this is currently an issue as all elements included in the hash are of fixed length but suggest this to avoid possible issues in the future if this changes.

Developer Response The developers have added the missing fields to the calculation of the hash.

4.1.17 V-KLA-VUL-017: Missing conversion to field element on output comparison

Severity	Low	**Commit**	6e2ce8f
Type	Logic Error	**Status**	Fixed
File(s)	`KailuaTournament.sol`		
Location(s)	prove		
Confirmed Fix At	5586e1d		

In the `prove` function, the following check is used to ensure that proofs are only submitted where the opponent is asserting some difference with the contender.

```
1 if (proposedOutput[0] == proposedOutput[1]) {
2     revert NoConflict();
3 }
```

Snippet 4.22: Snippet from `prove`

This check fails to first convert the proposed outputs to field elements using the `hashToFe` function as is used later in the function.

Impact This can allow this check to be passed if the proposed outputs are different *before* conversion to a field element but the same after (for instance the values `0` and `p`). At the moment, this is a low severity issue because the contender is always assumed to be the first output, their output is checked first, and if it matches, they are simply chosen as the winner. Thus, a challenger *cannot* use this to choose the correct output plus `p` and get a valid proposal rejected. However, this (1) allows what should be invalid proofs to be processed and posted and (2) could introduce serious bugs if the logic changes even slightly (e.g., if the contender could ever be the *second* competitor in the proof).

Recommendation Convert both outputs to field elements before comparing.

It should be noted that fixing V-KLA-VUL-002 will *not* resolve this issue.

Developer Response The developers added a fix which enforces that the proposed output values are within the field *if* the proposals differ on the last element. Otherwise, this is enforced in the call to `verifyIntermediateOutput` which uses the KZG precompile to perform this check.

4.1.18 V-KLA-VUL-018: Network issues can crash validator

Severity	Low	**Commit**	6e2ce8f
Type	Denial of Service	**Status**	Addressed and Partially Verified
File(s)	`bin/cli/src/validate.rs`		
Location(s)	handle_proposals		
Confirmed Fix At	`https://github.com/risc0/kailua/pull/21`		

In the validator, if the `handle_proposals` function ever returns an error, this will cause the validator to crash. In the main loop, errors can be returned in two different cases: whenever loading proposals fails and whenever requesting a proof fails. These can fail due to simple network failures when communication with a node fails, meaning any network disruption could cause a validator to crash.

Impact As mentioned above, network failures can crash validators.

Recommendation Handle network failures without crashing the validator.

Developer Response The developers have added in retry logic to avoid crashing for a number of the possible errors that can arise during normal functioning of a validator.

While these fixes address that heart of the issue raised by auditors, the Veridise team also notes that this change was made alongside a number of functional changes to the logic that could introduce other issues. Veridise strongly recommends to developers that the new logic be audited in its entirety to ensure no new or related issues were introduced and has thus marked the issue as "addressed and partially verified".

4.1.19 V-KLA-VUL-019: Proofs cannot be submitted for identical proposals with differing last elements

Severity	Warning	**Commit**	6e2ce8f
Type	Logic Error	**Status**	Addressed and Partially Verified
File(s)	`KailuaTournament.sol`		
Location(s)	prove		
Confirmed Fix At	5586e1d		

The final blob element for a given proposal is simply presumed to be the root claim. As such, for the on-chain `prove` logic, the final blob element for competing proposals are not checked, with the exception of the following logic, which is used to check that, in the case that the differing blob element is at the last possible index, all the blobs are simply identical (with presumably the only difference being the root claim).

```
// Find the divergent blob index
uint256 divergentBlobIndex = KailuaLib.blobIndex(uvo[2]);
if (uvo[2] == PROPOSAL_BLOCK_COUNT - 1) {
    // If the only difference is the root claim, require all blobs to be equal.
    divergentBlobIndex = PROPOSAL_BLOBS;
}
// Ensure blob hashes are equal until divergence
for (uint256 i = 0; i < divergentBlobIndex; i++) {
    if (childContracts[0].proposalBlobHashes(i).raw() != childContracts[1].
    proposalBlobHashes(i).raw()) {
        revert BlobHashMismatch(
            childContracts[0].proposalBlobHashes(i).raw(), childContracts[1].
    proposalBlobHashes(i).raw()
        );
    }
}
```

Snippet 4.23: Snippet from `prove()`

This leaves some strange behavior for the final blob element. Someone could submit a challenge that is entirely valid (modulo the last blob elements up until the end of the last blob which are unused) and be unable to submit their challenge.

Impact The impact is relatively minimal as the challenger need only update their last blob elements to match the contender's last blob element values. However, this is an annoyance for users who likely do not want to do this.

Recommendation Remove the special handling for checking when the difference is on the final root and instead enforce a canonical padding that must be used for all unused blob elements.

Disclosure This issue was found by developers during the early part of the audit and was shared with the audit team. The team verified that this is an issue and included it in the report for completeness.

Developer Response The developers no longer change the `divergentBlobIndex` in the special case when `uvo[2]` is equal to the root claim, and thus the special case of differing final elements is no longer relevant for this particular check.

However, the Veridise team also notes that this change was made alongside a number of functional changes to the logic that could introduce other issues. For instance, there is now an additional function `proveTrailFault` which allows disproving of proposals that *agree* on the root claim but *disagree* on the final intermediate output - introduction of this function could introduce other vulnerabilities, as it is an entirely new interface for providing proofs. Veridise strongly recommends to developers that the new logic be audited in its entirety to ensure no new or related issues were introduced and has thus marked the issue as "addressed and partially verified".

4.1.20 V-KLA-VUL-020: Elimination round calculation off-chain can become inconsistent with on-chain

Severity	Warning	**Commit**	6e2ce8f
Type	Logic Error	**Status**	Fixed
File(s)	`bin/cli/src/db/treasury.rs`		
Location(s)	fetch_elimination_round		
Confirmed Fix At	`https://github.com/risc0/kailua/pull/21`		

The function `fetch_elimination_round` is intended to get the elimination round for a given address. The implementation is given below.

```
let instance = self.treasury_contract_instance(provider);
let round = match self.elimination_round.entry(address) {
    Entry::Vacant(entry) => {
        let round = instance.eliminationRound(address).stall().await._0.to();
        *entry.insert(round)
    }
    Entry::Occupied(entry) => *entry.get(),
};
Ok(round)
```

Snippet 4.24: Implementation of `fetch_elimination_round`

As shown, the first time the elimination round is fetched, it fetches the information from on-chain and saves it into the the local `elimination_round` mapping. However, thereafter any time the elimination round for an address is fetched, it uses the stored value instead of querying the on-chain instance again.

This will not always return a consistent elimination round for an address as on-chain because the elimination round of an address can change over time. As a result, the stored local version in `elimination_round` can become out-of-date.

Impact This function is currently not used anywhere, so the impact of this is minimal. However, if one did use this function, substantial bugs could potentially occur due to the inconsistency between on-and-off-chain tracking of eliminations.

Recommendation Fix the logic to fetch information from chain or simply remove the function as it is unused.

Developer Response The developers deleted the function.

4.1.21 V-KLA-VUL-021: KZG precompile field modulus return not checked

Severity	Warning	**Commit**	6e2ce8f
Type	Logic Error	**Status**	Fixed
File(s)	`crates/contracts/src/KailuaLib.sol`		
Location(s)	verifyKZGBlobProof		
Confirmed Fix At	f1c7e42		

The following precompile call is made in the code.

```
bytes memory kzgCallData = abi.encodePacked(
    versionedHash, // proposalBlobHash().raw(),
    rootOfUnity,
    hashToFe(value),
    blobCommitment,
    proof
);
(success,) = KZG.call(kzgCallData);
```

Snippet 4.25: Snippet from `verifyKZGBlobProof()`

The return value of the call which is ignored returns the field modulus. It is possibly this could change in the future, so it is best practice to check the value.

Impact If the modulus changes, the computation could be incorrect unexpectedly and without clear explanation.

Recommendation Add a check on the field modulus returning by the call.

Developer Response The developers now compare the precompile response to the stored `FIELD_ELEMENTS_PER_BLOB` and `BLS_MODULUS` values.

4.1.22 V-KLA-VUL-022: Fetching blob returns default in the case of no match

Severity	Warning	**Commit**	6e2ce8f
Type	Logic Error	**Status**	Fixed
File(s)	bin/host/src/lib.rs		
Location(s)	get_blob_fetch_request		
Confirmed Fix At	f1c7e42		

The function `get_blob_fetch_request` is used to retrieve information about a blob and block matching the provided hashes. Below is the implementation of the function.

```
let block = l1_provider
    .get_block_by_hash(block_hash, BlockTransactionsKind::Full)
    .await?
    .expect("Failed to fetch block {block_hash}.");
let mut blob_index = 0;
for blob in block.transactions.into_transactions().flat_map(|tx| {
    tx.blob_versioned_hashes()
        .map(|h| h.to_vec())
        .unwrap_or_default()
    }) {
        if blob == blob_hash {
            break;
      }
      blob_index += 1;
}

Ok(BlobFetchRequest {
    block_ref: BlockInfo {
        hash: block.header.hash,
        number: block.header.number,
        parent_hash: block.header.parent_hash,
      timestamp: block.header.timestamp,
  },
    blob_hash: IndexedBlobHash {
        index: blob_index,
        hash: blob_hash,
    },
})
```

Snippet 4.26: Implementation of `get_blob_fetch_request`

As shown, the function iterates through transactions in the block matching the provided hash, and breaks when a matching hash is found. However, if the hash is not found, information about the header is still returned (even though it didn't contain a transaction with the desired blob) and it returns an incorrect blob hash.

Impact Currently this is only used to construct precondition validation data for the host. If incorrect data is provided, this could cause proof generation to fail. However, if in the future this logic is used elsewhere, it could lead to security vulnerabilities.

Recommendation Error in the case that no match is found.

Developer Response The developers now return an `Err` when a blob is not found.

4.1.23 V-KLA-VUL-023: Unused program constructs

Severity	Info	**Commit**	6e2ce8f
Type	Maintainability	**Status**	Fixed
File(s)	See issue description		
Location(s)	See issue description		
Confirmed Fix At	851e099		

Description The following program constructs are unused:

- `crates/contracts/foundry/src/KailuaLib.sol`
  - `error ClockExpired`
  - `error NotProven`
  - `error UnchallengedGame`
  - `error InvalidAnchoredGame`
  - `error BlockNumberMismatch`
- `bin/cli/src/`
  - `db/`
    - `config.rs`
      - `Config.game` field
      - `Config.verifier` field
      - `Config.image_id` field
      - `Config.game_type` field
      - `Config.factory` field
    - `mod.rs`
      - `pub enum ProofStatus`
      - `pub fn is_proposer_eliminated`
    - `treasury.rs`
      - `pub async fn fetch_elimination_round`
  - `propose.rs`
    - function `propose`
      - The variable `proposal` is declared twice.
- `bin/client/src/`
  - function `run_boundless_client`
    - `_journal` return from call `boundless_client.wait_for_request_fulfillment` is never used.

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 incorporated most suggestions, and for those that they did not, they have provided reasonable justification for not incorporating.

4.1.24 V-KLA-VUL-024: Small code suggestions

Severity	Info	**Commit**	6e2ce8f
Type	Maintainability	**Status**	Fixed
File(s)	See issue description		
Location(s)	See issue description		
Confirmed Fix At	27dec82		

Description In the following locations, the auditors identified the following places in the code where small changes are suggested:

- `crates/contracts/foundry/src`:
  - `KailuaLib.sol`:
    - `blobPosition()`:
      - This function computes a number that is guaranteed to be in the range `[0, 4095]` but returns a `uint256`. This function can be updated to return a `uint32` instead.
    - `modExp()`:
      - This computes the value `root ** x mod p` where `x` is the input to the function and both `root` and `p` are constants defined in the file - I have changed the names of these for clarity. However, in the actual function, the input is instead named `base` which is misleading given that it is used as the exponent in the calculation. We suggest renaming this variable to avoid confusion.
      - It is technically possible that calls to the `MOD_EXP` precompile could revert. It is likely the only cause of a revert is running of out of gas. However, to avoid all risk, we would suggest adding in a check that the call did not revert.
    - interface `IKailuaTreasury`:
      - The functions `eliminationRound` and `proposerOf` can be defined as `view` functions.
  - `KailuaTournament.sol`:
    - Most of the immutable variables are made `internal` and provided separate getters. These would have gettters provided automatically by making them `public`.
    - `prove()`:
      - There is a remaining `TODO` open for the case where both the contender and opponent lose.
- `bin/cli/src`:
  - `validate.rs`:
    - `handle_proofs()`:
      - This function contains the following logic applicable only to dev mode. In most places, the feature `#[cfg(feature = "devnet")]` is used for these, but not here.

```
if is_dev_mode() {
    kailua_host_command.env("RISC0_DEV_MODE", "1");
}
```

  - db/:
    * treasury.rs:
      · fetch_elimination_round():
        • Calculation of instance can be moved inside of the Vacant entry case as it is only used there.
      · fetch_proposes():
        • Same note as for fetch_elimination_round.
    * proposal.rs:
      · fetch_parent_tournament_survivor_status():
        • The expression survivor.map(|survivor| survivor == self.contract) is computed twice unnecessarily.

- crates/common/src:
  - blobs.rs:
    * get_blobs():
      · This continues to add elements to the vector after a match is found. This means there could be duplicate elements for a single blob hash, if the blob_hashes passed in contained duplicates. This is not a security issue as the only place it is used simply fetches a single hash and fetches the first element of the vector returned. However, this could lead to bugs in the future if used differently. We suggest continuing onto the next hash after a match is found to avoid duplicates.
  - client.rs:
    * validate_precondition():
      · The computation of agreed_l2_output_root can move out of the loop.

Impact These minor errors may lead to future developer confusion.

Developer Response The developers have incorporated most suggestions, and for those that they did not, they have provided reasonable justification for not incorporating.

Glossary

Denial-of-Service An attack in which the liveliness or ability to use a service is hindered . 5

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

reentrancy A vulnerability in which a smart contract hands off control flow to an unknown party while in an intermediate state, allowing the external party to take advantage of the situation. 5

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, 45

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, 45

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. 45

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