OpenZeppelin | security

# Taiko Shasta Protocol Re Audit

taiko

January 19, 2026

# Table of Contents

# Summary

| | |
|---|---|
| Type | Layer 2 & Rollups |
| Timeline | From 2025-12-15<br>To 2025-12-24 |
| Languages | Solidity |

| | |
|---|---|
| Total Issues | 6 (5 resolved, 1 partially resolved) |
| Critical Severity Issues | 0 (0 resolved) |
| High Severity Issues | 0 (0 resolved) |
| Medium Severity Issues | 0 (0 resolved) |
| Low Severity Issues | 3 (3 resolved) |
| Notes & Additional Information | 3 (2 resolved, 1 partially resolved) |

# Scope

OpenZeppelin performed a diff audit of the taikoxyz/taiko-mono repository, comparing the BASE commit 5034456 with the HEAD commit 430b332. During the audit, the diff comparing the BASE commit 430b332 and HEAD commit 2ad7378 was also added to the scope. The files below are annotated with full (commit 430b332), diff 1 (5034456...430b332), and diff 2 (430b332...2ad7378) depending on when they were reviewed.

```
packages/protocol/contracts
├── layer1
│   ├── core
│   │   ├── iface
│   │   │   ├── IBondManager.sol (diff 2)
│   │   │   ├── ICodec.sol (diff 1 & 2)
│   │   │   ├── IForcedInclusionStore.sol (diff 1)
│   │   │   ├── IInbox.sol (diff 1 & 2)
│   │   │   └── IProverWhitelist.sol (full)
│   │   ├── impl
│   │   │   ├── Codec.sol (full)
│   │   │   ├── Inbox.sol (full & diff 2)
│   │   │   ├── ProverWhitelist.sol (full)
│   │   └── libs
│   │       ├── LibBonds.sol (diff 2)
│   │       ├── LibBlobs.sol (diff 1)
│   │       ├── LibForcedInclusion.sol (diff 1)
│   │       ├── LibHashOptimized.sol (diff 1)
│   │       ├── LibInboxSetup.sol (full & diff 2)
│   │       ├── LibPackUnpack.sol (diff 1)
│   │       ├── LibProposeInputCodec.sol (diff 1)
│   │       ├── LibProposedEventCodec.sol (diff 1)
│   │       ├── LibProveInputCodec.sol (diff 1)
│   │       └── LibTransitionCodec.sol (full)
│   ├── mainnet
│   │   └── MainnetInbox.sol (diff 1 & 2)
│   ├── preconf
│   │   └── impl
│   │       └── PreconfWhitelist.sol (diff 1)
│   └── verifiers
│       ├── IProofVerifier.sol (diff 1)
│       ├── LibPublicInput.sol (diff 1)
│       ├── Risc0Verifier.sol (diff 1)
│       ├── SP1Verifier.sol (diff 1)
│       ├── SgxVerifier.sol (diff 1)
│       └── compose
│           └── ComposeVerifier.sol (diff 1)
├── layer2
│   └── core
│       ├── Anchor.sol (diff 1 & 2)
```

```
│       ├── AnchorForkRouter.sol (diff 1)
│       ├── BondManager.sol (full)
│       ├── IBondManager.sol (diff 1)
│       └── IBondProcessor.sol (full)
└── shared
    ├── fork-router
    │   └── ForkRouter.sol (diff 1)
    └── signal
        └── SignalService.sol (diff 1)
```

# System Overview

The Taiko Shasta protocol is a "Based Rollup" architecture that leverages Ethereum (L1) for sequencing and data availability while executing state transitions on a Layer 2 (L2) network. The system is designed to be permissionless and decentralized, utilizing a multi-proof mechanism that combines Trusted Execution Environments (SGX) and Zero-Knowledge Proofs (ZK-SNARKs/STARKs) to verify L2 state transitions on L1.

## L1 Core and Inbox

The `Inbox` contract serves as the central entry point for the protocol on Layer 1. It acts as the anchor for the L2 chain, managing the proposal of new blocks, the submission of validity proofs, and the finalization of the chain state.

- **Block Proposal**: `Inbox` accepts L2 block proposals, which include transaction data blobs. It enforces sequencing rules and interacts with the `PreconfWhitelist` (via the `IProposerChecker` interface) to validate authorized proposers during specific windows.
- **Proof Verification**: The contract coordinates the verification of state transitions. It supports a modular verification architecture where different proof types (SGX, Risc0, SP1) are routed to their respective verifiers.
- **Bond Management**: Instead of a standalone manager, `Inbox` integrates `LibBonds` to handle the economic security of the protocol. It manages Taiko Token (TKO) deposits, withdrawals, and the slashing of liveness bonds for provers who fail to submit proofs within the designated window.
- **Forced Inclusions**: The system implements a censorship-resistance mechanism allowing users to force transaction inclusion directly via L1 if L2 proposers are unresponsive. `Inbox` queues these requests and enforces their processing.

# Proof Verification Architecture

The verification layer is modular, allowing for flexible security configurations. It consists of specific verifier implementations and a composition layer.

- `ComposeVerifier`: An abstract verifier that aggregates multiple sub-verifiers (e.g., requiring both an SGX proof and a ZK proof). It ensures that a state transition is considered valid only if all required sub-verifiers approve it.
- Specific Verifiers:
  - `SgxVerifier`: Verifies signatures from attested SGX enclaves. It maintains a registry of valid instances and handles the expiration of attestations to prevent side-channel attacks.
  - `Risc0Verifier` & `SP1Verifier`: These contracts verify ZK proofs generated by the RISC Zero and SP1 zkVMs, respectively. They validate that the execution trace matches the public inputs derived from `Inbox`'s state.

# L2 Anchor and Signal Service

On L2, the protocol maintains synchronization with L1 and facilitates cross-chain communication.

- `Anchor`: This contract is responsible for anchoring L1 block data onto L2. It validates and stores L1 checkpoints (block hash, state root), providing the L2 chain with a trusted view of the L1 state. It utilizes a "Golden Touch" address to ensure that only protocol-authorized transactions can update the anchor state.
- `SignalService`: A general-purpose cross-chain messaging protocol. It allows contracts on one layer to "send signals" (store data) and prove on the other layer that a signal was sent. This underpins the bridge and other cross-chain applications.
- `ForkRouter`: A utility framework used to manage protocol upgrades. It routes calls between legacy and new implementations using `delegatecall`, allowing specific functionalities (like the Anchor) to be upgraded while preserving storage layout compatibility.

# Preconfirmation Whitelist

The `PreconfWhitelist` contract acts as a gatekeeper for block proposers. It manages a set of active operators who are authorized to provide pre-confirmations. This contract determines

who is allowed to propose blocks to the Inbox during the exclusive submission window, enabling a pre-confirmation layer before L1 sequencing.

# Security Model and Trust Assumptions

The Taiko Shasta protocol relies on the security of the underlying Ethereum L1 for data availability and transaction ordering. The system's security model combines economic incentives (bonds) with cryptographic and hardware-based proofs to ensure state validity. While striving for decentralization, the current iteration relies on specific governance controls and trusted hardware assumptions.

The critical trust assumptions and security model components are as follows:

- **L1 Security**: The system assumes that Ethereum L1 is secure, censorship-resistant, and that its history is immutable. L1 is the ultimate source of truth for the canonical L2 chain.
- **Governance and Privileged Roles**:
  - **Owner/DAO**: The protocol includes an `Owner` role (a DAO or multi-sig) with the power to upgrade contracts and update verifier configurations (e.g., adding trusted SGX instances or ZK verification keys).
  - **Prover Whitelist**: If enabled, the system relies on a whitelist of authorized provers. There is a fallback mechanism to allow permissionless proving if the whitelisted provers colluding to censor or halt the chain.
  - **Preconf Operators**: The entities managed by the `PreconfWhitelist` are trusted to provide fair pre-confirmations and not to abuse their exclusive proposal windows.
- **Verifier and Circuit Correctness**:
  - **Trusted Execution Environments (TEE)**: The `SgxVerifier` contract assumes that Intel SGX hardware is secure and that the remote attestation process is not compromised. It assumes the enclave code correctly implements the protocol rules.
  - **Zero-Knowledge Circuits**: The `Risc0Verifier` and `SP1Verifier` contracts assume that the underlying ZK circuits correctly constrain the state transition function and that the verifier contracts correctly implement the verifier logic for the respective proof systems.

- **Economic Security**:
  - **Bond Sufficiency**: It is assumed that the `minBond` and `livenessBond` values are set sufficiently high to make spamming or griefing attacks economically irrational for proposers and provers.
  - **Rational Actors**: The protocol assumes that provers and proposers are rational actors motivated by profit (fees and rewards) and deterred by slashing penalties.
- **Data Availability**: The system relies on Ethereum's EIP-4844 (blobs) for data availability. It is assumed that blob data is available for a sufficient period to allow provers to derive the chain state and generate proofs.
- **Cross-Chain Integrity**:
  - The `Anchor` contract assumes the integrity of the data provided by the "Golden Touch" transaction, which is generated by the protocol's node software.
  - The `SignalService` relies on the correctness of Merkle proofs to verify cross-chain messages, assuming that the root hash anchored on the destination chain is correct.

# Low Severity

## L-01 Misleading documentation

Throughout the codebase, multiple instances of misleading documentation were identified. For instance, the `Derivation.md` description:

- incorrectly states that `isForcedInclusion` should be `false` when discussing forced inclusion sources
- still refers to the obsolete "bond instruction" concept. It correctly describes the L1 behavior, but it is listed as a subsection under "Metadata Validation and Computation"
- references a non-existent `ShastaAnchor` contract
- omits the `parentProposalHash` and `endOfSubmissionWindowTimestamp` fields from its proposal-level metadata tables, despite these being present in the on-chain `Proposal` struct and `Proposed` event.

In addition, the `Checkpoint` struct fields are described as L2 data, but it is also used for L1 checkpoints.

Lastly, the `minCheckpointDelay` is described as less than a "finalization grace period", but it is configured to be larger than `maxProofSubmissionDelay` and it is not clear which other variable the grace period refers to. Any such requirement should be enforced when validating the configuration.

Consider updating the documentation accordingly.

***Update:*** *Resolved in pull request #21161.*

## L-02 Imprecise Error

The `ProverWhitelistedAlready` error suggests the owner is attempting to whitelist a whitelisted address. However, the error is also used when the owner attempts to remove an address that is not yet whitelisted.

Consider updating the `ProverWhitelistedAlready` error to account for both cases.

***Update:*** *Resolved in pull request #21162. The team stated:*

> *Implemented 2 different errors.*

## L-03 Incorrect Storage Gap

The `Anchor` contract's state variables occupy 7 storage slots. However, the `__gap` variable only pads this to 48 (instead of the standard 50).

Consider increasing the gap variable accordingly.

***Update:*** *Resolved in pull request #21163. The team stated:*

> *Updated the contract to use the remaining slots.*

# Notes & Additional Information

## N-01 Code Simplification

The `_getAvailableCapacity` function is only ever used to check if there is any available capacity.

Consider replacing the `_getAvailableCapacity` function with an `_isSpaceAvailable` check that simply returns `_ringBufferSize > _nextProposalId - _lastFinalizedProposalId`.

***Update:*** *Resolved in pull request #21166.*

## N-02 Code Consistency

The `forcedInclusionDelay` field of the `Inbox` config struct is set to `384` instead of `384 seconds`. This is inconsistent with how other duration fields are set.

For consistency, consider specifying the forced inclusion delay as `384 seconds` instead of just `384`.

***Update:*** *Resolved in pull request #21164. The team stated:*

> *Used explicit seconds as a unit.*

## N-03 Naming Suggestions

Throughout the codebase, multiple opportunities for improve naming were identified:

- `lastProposalBlockId` could be named `lastProposalL1BlockNumber`.
- Some fields in the `CoreState` struct refer to proposal finalization. However, finalization is achieved with proving and is no longer an independent phase. Thus, the naming can be adjusted to "proven" or similar.
- `PROPOSAL_MAX_BLOCKS` could be renamed to `DERIVATION_SOURCE_MAX_BLOCK` in line with its actual usage.

Consider implementing the above renaming suggestions to improve code clarity and maintainability.

***Update:*** *Partially Resolved in pull request #21165. The Taiko team changed the wording of the documentation, while the code remains the same. The team stated:*

> *Updated, but avoid doing ABI breaking changes.*

# Conclusion

This audit covered the Shasta protocol version of the Taiko Based Rollup in a 12-day engagement. Overall, the codebase appears to be in good shape and demonstrates a robust security posture. The findings are limited to low- and note-severity issues.

The Taiko team is commended for being very helpful and responsive throughout the engagement, providing clear insights into the system's design and architecture.

# Appendix

## Rollup Stages

L2Beat includes a set of criteria for classifying rollups into stages of maturity. The design of the system naturally satisfies all Stage 0 criteria.

With respect to the Stage 1 criteria, the Taiko team intends to:

- deploy the new inbox
- configure it with a working Verifier contract
- activate it
- transfer ownership to the `TaikoDAOController`

Once this occurs, the only ways to make invalid L2 withdrawals will be to:

- maliciously upgrade the inbox contract, or
- change the verifier contract (or function image it verifies against).

This will require action from the `TaikoDAOController` contract, which meets all the Stage 1 requirements:

- 6 of 8 (75%) security council members are required to make an emergency proposal.
- Regular proposals will take effect 7 days after they are passed.

The codebase currently utilizes a prover whitelist. However, pull request #21146 **introduces a permissionless fallback mechanism**. Thus, if whitelisted provers fail to prove a block within the configured `permissionlessProvingDelay` (set to 72 hours), proving becomes open to anyone. This ensures that whitelisted provers cannot halt the chain indefinitely. Since this 3-day delay is strictly shorter than the 7-day governance timelock, users retain the ability to exit even in a censorship scenario, making the system compatible with Stage 1 requirements.

## Issue Classification

OpenZeppelin classifies smart contract vulnerabilities on a 5-level scale:

- Critical
- High

- Medium
- Low
- Note/Information

### Critical Severity

This classification is applied when the issue's impact is catastrophic, threatening extensive damage to the client's reputation and/or causing severe financial loss to the client or users. The likelihood of exploitation can be high, warranting a swift response. Critical issues typically involve significant risks such as the permanent loss or locking of a large volume of users' sensitive assets or the failure of core system functionalities without viable mitigations. These issues demand immediate attention due to their potential to compromise system integrity or user trust significantly.

### High Severity

These issues are characterized by the potential to substantially impact the client's reputation and/or result in considerable financial losses. The likelihood of exploitation is significant, warranting a swift response. Such issues might include temporary loss or locking of a significant number of users' sensitive assets or disruptions to critical system functionalities, albeit with potential, yet limited, mitigations available. The emphasis is on the significant but not always catastrophic effects on system operation or asset security, necessitating prompt and effective remediation.

### Medium Severity

Issues classified as being of medium severity can lead to a noticeable negative impact on the client's reputation and/or moderate financial losses. Such issues, if left unattended, have a moderate likelihood of being exploited or may cause unwanted side effects in the system. These issues are typically confined to a smaller subset of users' sensitive assets or might involve deviations from the specified system design that, while not directly financial in nature, compromise system integrity or user experience. The focus here is on issues that pose a real but contained risk, warranting timely attention to prevent escalation.

### Low Severity

Low-severity issues are those that have a low impact on the client's operations and/or reputation. These issues may represent minor risks or inefficiencies to the client's specific business model. They are identified as areas for improvement that, while not urgent, could enhance the security and quality of the codebase if addressed.

### Notes & Additional Information Severity

This category is reserved for issues that, despite having a minimal impact, are still important to resolve. Addressing these issues contributes to the overall security posture and code quality improvement but does not require immediate action. It reflects a commitment to maintaining high standards and continuous improvement, even in areas that do not pose immediate risks.