RiscZero

The Signal: Ethereum

Security Assessment Report

Version: 2.0

July, 2025

Contents

Introduction

Sigma Prime was commercially engaged to perform a time-boxed security review of the RiscZero components in scope. The review focused solely on the security aspects of these components, though general recommendations and informational comments are also provided.

Disclaimer

Sigma Prime makes all effort but holds no responsibility for the findings of this security review. Sigma Prime does not provide any guarantees relating to the function of the components in scope. Sigma Prime makes no judgements on, or provides any security review, regarding the underlying business model or the individuals involved in the project.

Document Structure

The first section provides an overview of the functionality of the RiscZero components contained within the scope of the security review. A summary followed by a detailed review of the discovered vulnerabilities is then given which assigns each vulnerability a severity rating (see Vulnerability Severity Classification), an *open/closed/resolved* status and a recommendation. Additionally, findings which do not have direct security implications (but are potentially of interest) are marked as *informational*.

The appendix provides additional documentation, including the severity matrix used to classify vulnerabilities within the RiscZero components in scope.

Overview

The Signal: Ethereum (a.k.a. TSEth) leverages RiscZero's Zero Knowledge Virtual Machine (zkVM) to verify Ethereum's Casper FFG consensus and enabling provable execution of Rust programs using Beacon Chain state.

This uses two components: guest and host programs. The host program handles fetching the Beacon Chain state and processing this data for guest program use before undertaking the execution of the guest program. The guest program runs in the zkVM, providing trusted, isolated computation. Execution of the guest program produces a zk proof known as the receipt. The receipt can then be cryptographically verified by the host or a third party.

Third parties (smart contracts, servers) can confirm the guest program executed correctly using this receipt, allowing the trusted use of state changes for purposes such as cross-chain swaps.

Security Assessment Summary

Scope

The review was conducted on the files hosted on the RiscZero The Signal: Ethereum (TSEth) repository.

The scope of this time-boxed review was strictly limited to files at commit a0e54d5, particularly only the files contained in the following directories:

- `core/`
- `host/`
- `methods/`
- `ssz-multiproofs/`

Note: third party libraries and dependencies were excluded from the scope of this assessment.

Approach

The security assessment covered components written in Rust.

The manual review focused on identifying issues associated with the business logic implementation of the components in scope. This includes their internal interactions, intended functionality and correct implementation with respect to the underlying functionality of the Rust language.

Additionally, the manual review process focused on identifying vulnerabilities related to known Rust anti-patterns and attack vectors, such as unsafe code blocks, integer overflow, floating point underflow, deadlocking, error handling, memory and CPU exhaustion attacks, and various panic scenarios including index out of bounds, `panic!()`, `unwrap()`, and `unreachable!()` calls.

To support the Rust components of the review, the testing team may use the following automated testing tools:

- Clippy linting: `https://doc.rust-lang.org/stable/clippy/index.html`
- Cargo Audit: `https://github.com/RustSec/rustsec/tree/main/cargo-audit`
- Cargo Outdated: `https://github.com/kbknapp/cargo-outdated`
- Cargo Geiger: `https://github.com/rust-secure-code/cargo-geiger`
- Cargo Tarpaulin: `https://crates.io/crates/cargo-tarpaulin`

Output for these automated tools is available upon request.

Coverage Limitations

Due to the time-boxed nature of this review, all documented vulnerabilities reflect best effort within the allotted, limited engagement time. As such, Sigma Prime recommends to further investigate areas of the code, and any related functionality, where majority of critical and high risk vulnerabilities were identified.

Findings Summary

The testing team identified a total of 6 issues during this assessment. Categorised by their severity:

- Critical: 2 issues.
- Medium: 1 issue.
- Low: 1 issue.
- Informational: 2 issues.

Detailed Findings

This section provides a detailed description of the vulnerabilities identified within the RiscZero components in scope. Each vulnerability has a severity classification which is determined from the likelihood and impact of each issue by the matrix given in the Appendix: Vulnerability Severity Classification.

A number of additional properties of the components, including optimisations, are also described in this section and are labelled as "informational".

Each vulnerability is also assigned a **status**:

- ***Open:*** the issue has not been addressed by the project team.
- ***Resolved:*** the issue was acknowledged by the project team and updates to the affected components(s) have been made to mitigate the related risk.
- ***Closed:*** the issue was acknowledged by the project team but no further actions have been taken.

Summary of Findings

ID	Description	Severity	Status
ZKA-01	No Replay Protection In Attestation Verification Process	Critical	Resolved
ZKA-02	Attestation Manipulation Via Committee Bits And Aggregate Signature Modification	Critical	Resolved
ZKA-03	Missing Attestation Slot-Target Epoch Validation	Medium	Resolved
ZKA-04	Usage Of `unwrap()` and `expect()` In Host Code	Low	Closed
ZKA-05	Differing Methods Of Setting `CHUNK_SIZE`	Informational	Closed
ZKA-06	Miscellaneous General Comments	Informational	Closed

ZKA-01	No Replay Protection In Attestation Verification Process		
Asset	`core/src/verify.rs`		
Status	**Resolved:** See Resolution		
Rating	Severity: Critical	Impact: High	Likelihood: High

Description

The `verify()` function lacks a critical check for duplicate attestations, thereby allowing a malicious validator to inflate balance values by replaying attestations.

During the validation process, the `verify()` function `calls process_attestation()`, which after extracting attesting validators, calls `is_valid_indexed_attestation()` to verify the attestation.

The `is_valid_indexed_attestation()` function will validate that the validator set is not empty, extract the `signing root` for the `domain` and call `signature.fast_aggregate_verify()` to check the cryptographic validity.

This issue has been given a likelihood and severity of high, since the inflated balance could exploit the threshold verification process of a new justification, as seen in the following code snippet of the `verify()` function:

```
let mut target_balance = 0u64;
        for attestation in attestations {
            let attesting_balance =
                process_attestation(state_reader, &active_validators, &committees, attestation)?;
            target_balance.safe_add_assign(attesting_balance)?;
        }

        let total_active_balance = get_total_balance(spec, active_validators.values())?;
        debug!(
            target_balance,
            total_active_balance, "Attestations processed"
        );

        // the target balance must be sufficient for a new justification
        let lhs = target_balance as u128 * cfg.justification_threshold_quotient as u128;
        let rhs = total_active_balance as u128 * cfg.justification_threshold_factor as u128;
        ensure!(
            lhs >= rhs,
            VerifyError::ThresholdNotMet {
                attesting_balance: target_balance,
                // this is not exactly equivalent, but good enough for an error message
                threshold: (rhs / cfg.justification_threshold_quotient as u128) as u64,
            }
        );
```

Recommendations

Implement a check for duplicate attestations per validator for same `link`, as this could be a slashable offense.

Resolution

This issue has been resolved by the development team in PR 105, where each validators' balance is now only counted once towards the threshold, regardless of whether the attestation has been included multiple times.

ZKA-02	Attestation Manipulation Via Committee Bits And Aggregate Signature Modification		
Asset	`core/src/verify.rs`		
Status	**Resolved:** See Resolution		
Rating	Severity: Critical	Impact: High	Likelihood: High

Description

The attestation verification process is vulnerable to manipulation through coordinated modification of committee bits and aggregate signatures, potentially compromising the integrity of finalisation.

The `verify()` function computes a set of active validators into a committee of attesting validators and uses their public keys to validate the aggregate signature, as shown in the following code snippet:

```
// compute all committees for the target epoch
let active_validators: BTreeMap<_, _> = state_reader
    .active_validators(target_epoch)
    .map_err(|e| VerifyError::StateReaderError(e.to_string()))?
    .collect();
let committees = compute_committees(state_reader, &active_validators, target_epoch)?;

info!("Processing attestations for {}", link);
let mut target_balance = 0u64;
for attestation in attestations {
    let attesting_balance =
        process_attestation(state_reader, &active_validators, &committees, attestation)?;
    target_balance.safe_add_assign(attesting_balance)?;
}
```

The vulnerability stems from the lack of proper replay protection and validator deduplication across multiple submissions of similar attestations for the same `(source, target)` link. A malicious node could submit the same underlying attestation multiple times whilst strategically including or excluding validators to bypass basic replay detection mechanisms.

This double-counting attack works by submitting variations of the same attestation where:

- The core attestation data `(source, target)` remains identical
- Committee bits are modified to include or exclude specific validators
- Aggregate signatures are adjusted accordingly to maintain cryptographic validity
- Each variation appears as a distinct attestation to the verification process
- The system accumulates validator balances multiple times in the `target_balance` calculation

For example, an attacker could submit attestation A with validators {1,2,3,4} and then submit attestation B with validators {2,3,4,5} for the same link. Validators 2, 3, and 4 would have their stakes counted twice towards the threshold, artificially inflating the attesting balance and potentially enabling premature justification of checkpoints that lack genuine supermajority support.

This vulnerability has been classified as high impact and high likelihood because it directly compromises the threshold verification process used for block justification and finalisation.

Recommendations

Implement robust replay protection and validator deduplication to ensure each validator's stake is counted only once per `(source, target)` link. This could be achieved by:

- Tracking which validators have already been counted for each specific link across all attestation submissions
- Implementing a validator set union approach that prevents the same validator from contributing to the threshold multiple times
- Adding attestation fingerprinting that considers validator overlap, not just exact attestation equality

Resolution

This issue has been resolved by the development team in PR 105, where each validators' balance is now only counted once towards the threshold.

ZKA-03	Missing Attestation Slot-Target Epoch Validation		
Asset	`core/src/verify.rs`		
Status	**Resolved:** See Resolution		
Rating	Severity: Medium	Impact: Medium	Likelihood: Medium

Description

The `zkasper` attestation processing implementation does not validate that an attestation's slot corresponds to its target epoch, violating a requirement of the Ethereum consensus specification and potentially allowing malicious attestations to manipulate the finalisation process.

In Ethereum's Casper FFG consensus mechanism, attestations serve as votes for checkpoint finalisation. Each attestation contains a `data.slot` field indicating when the attestation was created and a `data.target` field specifying the checkpoint being voted for. The Ethereum consensus specification mandates that `attestation.data.target.epoch == compute_epoch_at_slot(attestation.data.slot)` meaning an attestation's slot must belong to the same epoch as its target checkpoint.

The `zkasper` implementation's `process_attestation()` function processes attestations without performing this validation:

```rust
fn process_attestation(
    state: &S,
    active_validators: &BTreeMap,
    committees: &CommitteeCache,
    attestation: &Attestation,
) -> Result {
    let attesting_indices = get_attesting_indices(attestation, committees)?;
    // @audit Missing validation: attestation.data.target.epoch == attestation.data.slot.epoch()
    let attesting_validators = attesting_indices
        .iter()
        .map(|i| {
            active_validators
                .get(i)
                .copied()
                .ok_or(VerifyError::MissingValidatorInfo(*i))
        })
        .collect::, _>>()?;

    // verify signature
    ensure!(
        is_valid_indexed_attestation(
            state,
            &attesting_validators,
            attestation.data(),
            attestation.signature(),
        )?,
        VerifyError::InvalidAttestation("Invalid signature")
    );

    // sum up the effective balance of all validators who have not been slashed
    let target_balance = get_total_balance(
        state.chain_spec(),
        attesting_validators.iter().filter(|v| !v.slashed),
    )?;

    Ok(target_balance)
}
```

This contrasts with Lighthouse's implementation, which correctly validates this requirement in `verify_casper_ffg_vote()`:

```
fn verify_casper_ffg_vote(
    attestation: AttestationRef,
    state: &BeaconState,
) -> Result<()> {
    let data = attestation.data();
    verify!(
        data.target.epoch == data.slot.epoch(E::slots_per_epoch()),
        Invalid::TargetEpochSlotMismatch {
            target_epoch: data.target.epoch,
            slot_epoch: data.slot.epoch(E::slots_per_epoch()),
        }
    );
    // ...
}
```

Without this validation, an attacker could potentially craft attestations where the slot belongs to one epoch but targets a checkpoint from a different epoch. While such attestations would still require valid signatures from actual validators, this missing check could enable several attack vectors:

- Attestations from validators in epoch N could be used to vote for targets in epoch `N+1` or `N-1`.
- Malformed attestations could potentially affect the timing or validity of checkpoint finalisation.
- The zkVM would accept attestations that violate fundamental Ethereum consensus rules

The `verify()` function processes attestations in batches grouped by their `(source, target)` links and uses the accumulated attesting balance to determine if sufficient stake exists for justification. If invalid attestations are accepted due to this missing validation, it could lead to incorrect finalisation decisions, warranting a medium impact rating.

The likelihood is classified as medium as an attacker may obtain signed attestations from neighbouring epochs which would be rejected by consensus clients but accepted in TSEth. However, to exploit this a situation must occur where the header chain has provided less than the threshold number of attestation votes for in a given epoch, but the votes from neighbouring epochs are sufficient to create finality.

Recommendations

Add the missing slot-target epoch validation to the `process_attestation()` function before any other processing.

Resolution

This issue has been resolved by the development team in PR 120.

ZKA-04	Usage Of `unwrap()` and `expect()` In Host Code		
Asset	`host/src/state_patch_builder.rs, host/src/conversions.rs`		
Status	**Closed:** See Resolution		
Rating	Severity: Low	Impact: Medium	Likelihood: Low

Description

The host code contains multiple instances of `unwrap()` and `expect()` calls that can trigger panics when encountering unexpected `None` values or `Err` results. When these panics occur, they cause the entire host process to terminate, potentially stalling the system and disrupting normal operation until manual intervention or automatic restart mechanisms recover the service.

Usage of `unwrap()` occurs in:

- `conversions.rs::to_validator_info()` on line [`74`]
- `input_builder.rs::collect_attestations_for_links()` on lines [`177, 178`]
- `state_patch_builder.rs::randao_mix()` on line [`40`]
- `host_state_reader.rs::randao_mix()` on line [`150`]

Usage of `expect()` occurs in:

- `conversions.rs::conv_attestation()` on lines [`33,37,40,42,45,48`]

This issue has been given low severity, as these `unwrap()` and `expect()` statements are not expected to be reachable except in rare cases such as a malformed response from a consensus client.

Recommendations

Modify implementation to return errors instead of using `unwrap()`.

Resolution

This issue has been closed by the development team with a comment that the issue will be addressed without a need for retesting.

ZKA-05	Differing Methods Of Setting `CHUNK_SIZE`
Asset	`ssz_multiproof/src/multiproof.rs`
Status	**Closed:** See Resolution
Rating	Informational

Description

The constant `CHUNK_SIZE` is used to set the length of various fields when calculating Merkle tree roots. However, when calling the helper function `calculate_max_stack_depth()` in order to prepare the `max_stack_depth_hint` used in preparing Merkle trees a hardcoded value of 32 is used for `CHUNK_SIZE`.

```
pub(crate) fn calculate_max_stack_depth(descriptor: &Descriptor) -> usize {
    let mut stack = Vec::new();
    let mut max_stack_depth = 0;
    for bit in descriptor.iter() {
        if *bit {
            stack.push(TreeNode::Computed([0; 32])); // @audit Hardcoded value of 32
            while stack.len() > 2
                && stack[stack.len() - 1].has_value()
                && stack[stack.len() - 2].has_value()
                && stack[stack.len() - 3].is_internal()
            {
                stack.pop();
                stack.pop();
                stack.pop();
                stack.push(TreeNode::Computed([0; 32])); // @audit Hardcoded value of 32
                max_stack_depth = max_stack_depth.max(stack.len());
            }
        } else {
            stack.push(TreeNode::Internal);
            max_stack_depth = max_stack_depth.max(stack.len());
        }
    }
    assert_eq!(stack.len(), 1);
    max_stack_depth
}
```

While this hardcoded value is identical to the current value of `CHUNK_SIZE` and so currently this is an informational issue only.

Recommendations

It is advised to use the same source for the config value `CHUNK_SIZE` throughout the code base to avoid discrepancies appearing if this value is changed.

Resolution

This issue has been closed by the development team with a comment that the issue will be addressed without a need for retesting.

ZKA-06	Miscellaneous General Comments
Asset	All assets
Status	**Closed:** See Resolution
Rating	Informational

Description

This section details miscellaneous findings discovered by the testing team that do not have direct security implications:

1. **Typo In Comments**
   Related Asset(s): core/src/verify.rs, core/src/lib.rs
   In `verify.rs` there is a typo in the comment on line [**65**] and in `lib.rs` there is a typo in the comment on line [**82**]. Instead of *"superiority link"* it should be *"supermajority link"*.

Recommendations

Ensure that the comments are understood and acknowledged, and consider implementing the suggestions above.

Resolution

This issue has been closed by the development team with a comment that the issue will be addressed without a need for retesting.

Appendix A Vulnerability Severity Classification

This security review classifies vulnerabilities based on their potential impact and likelihood of occurance. The total severity of a vulnerability is derived from these two metrics based on the following matrix.

Impact	High	Medium	High	Critical
	Medium	Low	Medium	High
	Low	Low	Low	Medium
		Low	Medium	High
			Likelihood	

Table 1: Severity Matrix - How the severity of a vulnerability is given based on the *impact* and the *likelihood* of a vulnerability.

References