hexens x RISC ZERO

Security Review Report for RISC Zero

July 2025

Table of Contents

1. About Hexens
2. Executive summary
3. Security Review Details
    - Security Review Lead
    - Scope
    - Changelog
4. Severity Structure
    - Severity characteristics
    - Issue symbolic codes
5. Findings Summary
6. Weaknesses
    - Division by zero DoS in mint calculator guest program
    - Unbounded loop in PovwMint contract can lead to DoS in epoch finalisation
    - Missing Binary Constraint on termA0Low in readReceiptClaimAndPovwNonce function
    - Preflight PoVW nonce load lacks lower-bound check
    - TODO comments in code

1. About Hexens

Hexens is a pioneering cybersecurity firm dedicated to establishing robust security standards for Web3 infrastructure, driving secure mass adoption through innovative protection technology and frameworks. As an industry elite experts in blockchain security, we deliver comprehensive audit solutions across specialized domains, including infrastructure security, Zero Knowledge Proof, novel cryptography, DeFi protocols, and NFTs.

Our methodology combines industry-standard security practices combined with unique methodology of two teams per audit, continuously advancing the field of Web3 security. This innovative approach has earned us recognition from industry leaders.

Since our founding in 2021, we have built an exceptional portfolio of enterprise clients, including major blockchain ecosystems and Web3 platforms.

2. Executive Summary

This report covers the security review for RISC0's Boundless network – it included the Proof of Verifiable Work update.

Our security assessment was a full review of the updates, spanning a total of 2 weeks.

During our review, we identified 2 high severity vulnerabilities, which could have resulted in permanent denial of service of the reward minting contracts, which would lead to significant network disruption.

We have also identified several minor severity vulnerabilities and code optimisations.

All of our reported issues were fixed or acknowledged by the development team and consequently validated by us.

We can confidently say that the overall security and code quality have increased after completion of our audit.

3. Security Review Details

Review Led by

SoonChan Hwang, Lead Security Researcher
Jahyun Koo, Lead Security Researcher

Scope

The analyzed resources are located on:

- https://github.com/risc0/zirgen/pull/250
- https://github.com/risc0/risc0/pull/3220
- https://github.com/boundless-xyz/boundless/pull/847

The issues described in this report were fixed in the following commits:

- https://github.com/risc0/zirgen/pull/267/
- https://github.com/boundless-xyz/boundless/commit/bc14c135cfeec418ab055179455a6a2000d3d644
- https://github.com/boundless-xyz/boundless/commit/c8e7c51d791b777e25ee63e35b1031b66487ba16

Changelog

28 July 2025	Audit start
12 August 2025	Initial report
21 August 2025	Revision received
27 August 2025	Final report

4. Severity Structure

The vulnerability severity is calculated based on two components:

1. Impact of the vulnerability
2. Probability of the vulnerability

Impact	Probability			
	Rare	Unlikely	Likely	Very likely
Low	Low	Low	Medium	Medium
Medium	Low	Medium	Medium	High
High	Medium	Medium	High	Critical
Critical	Medium	High	Critical	Critical

- ## Severity Characteristics

Smart contract vulnerabilities can range in severity and impact, and it's important to understand their level of severity in order to prioritize their resolution. Here are the different types of severity levels of smart contract vulnerabilities:

Critical

Vulnerabilities that are highly likely to be exploited and can lead to catastrophic outcomes, such as total loss of protocol funds, unauthorized governance control, or permanent disruption of contract functionality.

High

Vulnerabilities that are likely to be exploited and can cause significant financial losses or severe operational disruptions, such as partial fund theft or temporary asset freezing.

Medium Vulnerabilities that may be exploited under specific conditions and result in moderate harm, such as operational disruptions or limited financial impact without direct profit to the attacker.

Low Vulnerabilities with low exploitation likelihood or minimal impact, affecting usability or efficiency but posing no significant security risk.

Informational Issues that do not pose an immediate security risk but are relevant to best practices, code quality, or potential optimizations.

▪ Issue Symbolic Codes

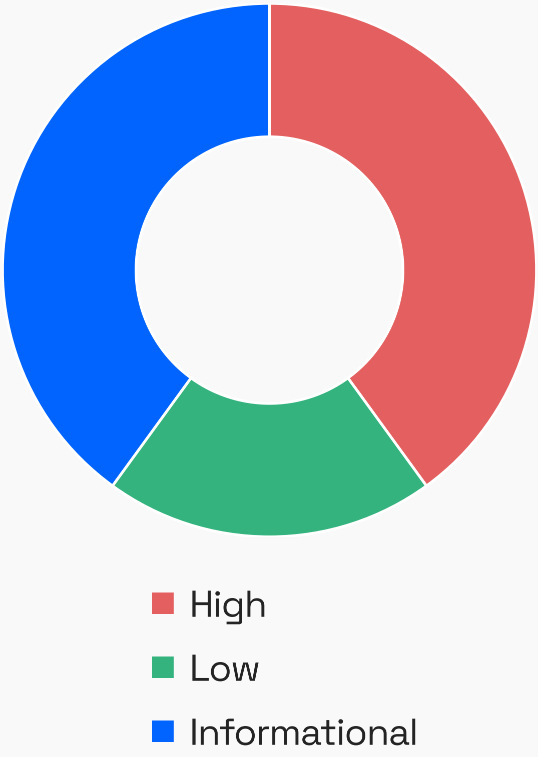
Each identified and validated issue is assigned a unique symbolic code during the security research stage.

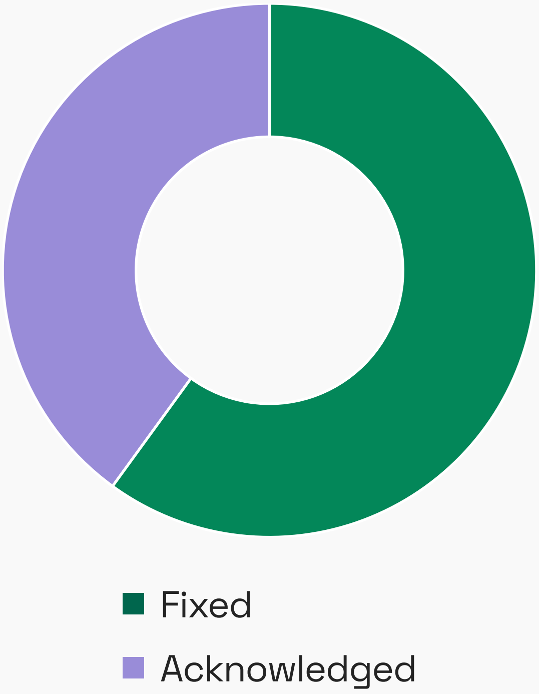
Due to the structure of the vulnerability reporting flow, some rejected issues may be missing.

5. Findings Summary

	Severity	Number of findings
🟥	Critical	0
🟥	High	2
🟧	Medium	0
🟩	Low	1
🟦	Informational	2

Total: **5**





6. Weaknesses

This section contains the list of discovered weaknesses.

RISC6-3 | Division by zero DoS in mint calculator guest program

Fixed ✓

Severity:	High	Probability:	Unlikely	Impact:	Critical

Path:

boundless/crates/guest/povw/mint-calculator/src/main.rs#L83

Description:

The mint calculator guest program is used to create a proof of events that happened in the PovwAccounting contract, to calculate rewards based on the value of the proving work done by the prover.

The accounting contract divides work into epochs and the mint calculator processes these by epoch. The rewards are calculator pro-rata of one update value against the total value of the epoch.

However, there is an edge case where the total value of an epoch (**totalWork**) is 0 but there is at least one update (**WorkLogUpdated**). This is possible to force if an epoch is empty, and one commits a work log update without any updates. The **log-builder** guest program accepts this and creates a commit with no value and the same root for both initial and updated commit. It could also be possible if it includes a request for 0 value, this would change the updated commit root.

In such a case, the e**poch_total_work** would be 0 and the **mint-calculator** guest program would run into a panic from division by zero on lines 82-83:

```
*mints.entry(update_event.valueRecipient).or_default() +=
    FixedPoint::fraction(update_event.updateValue, epoch_total_work);

[..]

pub fn fraction(num: U256, dem: U256) -> Self {
    let fraction = num.checked_mul(Self::BASE).unwrap() / dem;
    assert!(fraction <= Self::BASE, "expected fractional value is greater than
one");
```

```
    Self { value: fraction }
}
```

This would make proving the epoch impossible.

Because the commit roots form a chain and proving the update from this commit becomes impossible, it would also block the proving of any future commits proven by this **workLogId**. Such a future commit might be part of an epoch that is used and filled with normal user logs, in which case that processing would also become impossible, blocking normal users.

```
    for env in envs.0.values() {
        // Query all `WorkLogUpdated` events of the PoVW accounting contract.
        // NOTE: If it is a bottleneck, this can be optimized by taking a hint
from the host as to
        // which blocks contain these events.
        let update_events = Event::new::<IPovwAccounting::WorkLogUpdated>(env)
            .address(input.povw_contract_address)
            .query();

        for update_event in update_events {
            match updates.entry(update_event.workLogId) {
                btree_map::Entry::Vacant(entry) => {
                    entry.insert((update_event.initialCommit,
update_event.updatedCommit));
                }
                btree_map::Entry::Occupied(mut entry) => {
                    assert_eq!(
                        entry.get().1,
                        update_event.initialCommit,
                        "multiple update events for {:x} that do not form a
chain",
                        update_event.workLogId
                    );
                    entry.get_mut().1 = update_event.updatedCommit;
                }
            }

            // TODO: Consider minting to an address that is not necessarily the
log id.
            let epoch_number = update_event.epochNumber.to::<u32>();
            let epoch_total_work = *epochs.get(&epoch_number).unwrap_or_else(||
{
                panic!("no epoch finalized event processed for epoch number
{epoch_number}")
```

```
               });
               *mints.entry(update_event.valueRecipient).or_default() +=
                   FixedPoint::fraction(update_event.updateValue,
  epoch_total_work);
           }
       }
```

Remediation:

The **mint-calculator** guest program should handle the case where the total value is 0, so no division by zero happens, and the mint value simply becomes 0.

RISC6-4 | Unbounded loop in PovwMint contract can lead to DoS in epoch finalisation

Fixed ✓

Severity:	High	Probability:	Likely	Impact:	High

Path:

`boundless/contracts/src/povw/PovwMint.sol:mint#L105-L138`

Description:

The PovwMint contract takes a consolidated journal from the **mint-calculator** guest program to distribute rewards for work update logs from the PovwAccounting contract.

In order to do so, the **mint-calculator** guest program loops over every **WorkLogUpdated** event and constructs an array with the latest updated commit per work log ID. These are submitted in the call to **PovwMint:mint** in which it loops over each update:

```
for (uint256 i = 0; i < journal.updates.length; i++) {
    MintCalculatorUpdate memory update = journal.updates[i];

    // On the first mint for a journal, the initialCommit should be equal to
the empty root.
    bytes32 expectedCommit = lastCommit[update.workLogId];
    if (expectedCommit == bytes32(0)) {
        expectedCommit = EMPTY_LOG_ROOT;
    }

    if (update.initialCommit != expectedCommit) {
        revert IncorrectInitialUpdateCommit({expected: expectedCommit,
received: update.initialCommit});
    }
    lastCommit[update.workLogId] = update.updatedCommit;
}
```

The issue here is that one can create as many work logs for an arbitrary number of different work log IDs as you want. The work logs could be real requests, but they could also be empty, in which case it would not change the commit root, but it would create an entry in the **journal.updates** array.

As such, an attacker can forcefully fill the epoch with work logs of different work log IDs and create a journal update that will cost more gas than the block limit allows and cause it to always revert due to an out of gas error.

This would block all rewards from the entire epoch from being distributed.

Remediation:

A remediation for this issue is not arbitrary, so we recommend to consider:

1. Changing the mint function to instead take a journal with a Merkle root of the updates and mints that is stored in storage, so it is verified against the mint-calculator guest program and the Steel commitment.
2. Implement a second function to allow users to mint the actual rewards by providing a proof against the stored Merkle root.

This would make it a pull-based approach instead of push, getting rid of the need for unbounded loops.

RISC6-1 | Missing Binary Constraint on termA0Low in readReceiptClaimAndPovwNonce function

Fixed ✓

Severity:	Low	Probability:	Unlikely	Impact:	Low

Path:

zirgen/circuit/predicates/predicates.cpp

Description:

The **readReceiptClaimAndPovwNonce** function generates a receipt claim and a nonce from the provided input. The **claim.sysExit** value is determined based on the **isTerminate** flag and the **termA0Low** parameter. According to the specification, if **isTerminate** is **0**, then **sysExit** must be set to **2**. Otherwise, it should take the value of **termA0Low**, which is expected to be either **0** or **1**.

However, since there is no constraint enforced on the value of **termA0Low**, an attacker can exploit this by setting **termA0Low = 2**, generating an invalid provable state.

```
eqz(isTerminate * (1 - isTerminate));
...

// isTerminate:
// 0 -> 2
// 1 -> termA0Low (0, 1)
claim.sysExit = (2 - 2 * isTerminate) + (isTerminate * termA0Low);
```

Remediation:

Add a binary constraint on **termA0Low** value.

RISC6-5 | Preflight PoVW nonce load lacks lower-bound check

Acknowledged

Severity:	Informational	Probability:	Rare	Impact:	Informational

Path:

zirgen/zirgen/circuit/rv32im/v2/emu/preflight.cpp

Description:

In **preflight.cpp**, addresses in the range **[MEMORY_END_WORD, POVW_NONCE_END_WORD)** are treated as the PoVW nonce without checking the lower bound **POVW_NONCE_START_WORD**. For addresses below **POVW_NONCE_START_WORD**, the unsigned subtraction used to compute the nonce index underflows, leading to an out-of-bounds access and a runtime exception.

Example: **word = 0x43fffffe** satisfies **word >= 0x40000000** and **word < 0x44000008**; the computed index is **0x43fffffe - 0x44000000 = 0xfffffffe**, which is outside the 8-word nonce buffer.

```
constexpr uint32_t MEMORY_END_WORD        = 0x40000000;
constexpr uint32_t POVW_NONCE_START_WORD = 0x44000000;
constexpr uint32_t POVW_NONCE_END_WORD   = 0x44000008;
```

This affects preflight error handling only and does not impact proof soundness or on-chain verification.

```
if (word >= MEMORY_END_WORD) {
  if (pageMemory.count(word)) {
    val = pageMemory.at(word);
  } else if (word < POVW_NONCE_END_WORD) {
    val = segment.povwNonce.at(word - POVW_NONCE_START_WORD);
  } else {
    throw std::runtime_error("Invalid load from page memory");
  }
} else {
  val = pager.load(word);
}
```

Remediation:

- Constrain the nonce window with both bounds: **if (word >= POVW_NONCE_START_WORD && word < POVW_NONCE_END_WORD)**
- Optionally check **idx < 8** after **idx = word - POVW_NONCE_START_WORD**

RISC6-2 | TODO comments in code

Acknowledged

Severity:	Informational	Probability:	Likely	Impact:	Informational

Description:

In various places in the code, there are TODO comments that are still unresolved, where some are more important than others.

For example, in **PovwMint.sol** on line 76:

```
// TODO(povw): Extract to a shared library along with EPOCH_LENGTH.
// NOTE: Example value of 100 tokens per epoch, assuming 18 decimals.
uint256 public constant EPOCH_REWARD = 100 * 10 ** 18;
```

Remediation:

It is best practice to resolve these before deployment.

hexens x RISC ZERO