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
    - Duplicate Callback Execution on Re-submitted Request IDs
    - Consider Hardening GitHub Workflows
    - Missing Payment Guarantee for Unlocked or Post-Lock Fulfillments
    - Minor Consistency & Code-Hygiene Observations

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 audit for updates to RISC0 Boundless smart contracts and guest programs. Boundless is a universal execution protocol that brings a ZK proof market on-chain.

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

During our audit, we identified 1 medium severity vulnerability, in addition to several minor severity vulnerabilities and code optimisations.

All of our reported issues were acknowledged by the development team.

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/boundless-xyz/boundless/tree/a7b3fc42c8ccce828f1d46c176bebdcd6d5b7af3
  - contracts/src/*
  - crates/assessor/*
  - crates/guest/assessor/*

The issues described in this report were Acknowledged.

- **Changelog**

7 July 2025	Audit start
21 July 2025	Initial report
28 July 2025	Revision received
30 July 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	0
Medium	1
Low	2
Informational	1
Total:	**4**

6. Weaknesses

This section contains the list of discovered weaknesses.

RISC5-7 | Duplicate Callback Execution on Re-submitted Request IDs

Acknowledged

Severity:	Medium	Probability:	Likely	Impact:	Medium

Path:

contracts/src/BoundlessMarket.sol

Description:

BoundlessMarket currently skips the callback for an expired request, but it does execute the callback every time the same requestId appears in the fills[] array. As a result, a prover can present two distinct proofs for the same request (each with a different journal + seal) and cause the callback to run multiple times even though the request is already marked as fulfilled and no additional payment is made.

Example: in the sample Counter contract the callback increments a counter whenever a new journal digest is verified. If a prover supplies two valid proofs that satisfy the same request, increment() is invoked twice and the counter increases by 2, breaking the implicit "1 request → 1 count" expectation.

```
    function fulfill(Fulfillment[] calldata fills, AssessorReceipt calldata
assessorReceipt)
        public
        returns (bytes[] memory paymentError)
    {
        verifyDelivery(fills, assessorReceipt);

        paymentError = new bytes[](fills.length);

        uint256[] memory fillToCallbackIndexPlusOne = new uint256[]
(fills.length);
        uint256 callbacksLength = assessorReceipt.callbacks.length;
        for (uint256 i = 0; i < callbacksLength; i++) {
```

```
            AssessorCallback calldata callback = assessorReceipt.callbacks[i];
            fillToCallbackIndexPlusOne[callback.index] = i + 1;
        }

        for (uint256 i = 0; i < fills.length; i++) {
            Fulfillment calldata fill = fills[i];
            bool expired;
            (paymentError[i], expired) = _fulfillAndPay(fill,
assessorReceipt.prover);

            if (expired) {
                continue;
            }

            uint256 callbackIndexPlusOne = fillToCallbackIndexPlusOne[i];
            if (callbackIndexPlusOne > 0) {
                AssessorCallback calldata callback =
assessorReceipt.callbacks[callbackIndexPlusOne - 1];
                _executeCallback(fill.id, callback.addr, callback.gasLimit,
fill.imageId, fill.journal, fill.seal);
            }
        }
    }
```

Remediation:

- Add a check in fulfill() (or _executeCallback) to suppress the callback when _fulfillAndPay indicates the request is already fulfilled.
- Alternatively, extend handleProof to include **requestId** so callback contracts can enforce their own idempotency.
- Document in developer guides that callbacks must be idempotent if the interface is left unchanged.

Commentary from the client:

"We may consider including requestId in handleProof in a future release, but for now no plans to change. We will refactor the docs to make this property clearer."

RISC5-6 | Consider Hardening GitHub Workflows

Acknowledged

Severity:	Low	Probability:	Rare	Impact:	Medium

Path:

boundless/.github/workflows/main.yml
boundless/.github/workflows/release.yml
boundless/.github/workflows/contracts.yml

Description:

The affected GitHub workflows exhibit the following risks. Note that due to the public nature of PRs, live exploitation was not attempted, however it is recommended to consider hardening the workflows against potential abuse.

Using self-hosted runners

The affected workflows use self-hosted, non-ephemeral runners. Self-hosted runners can cause access tokens to be persisted between workflow runs, even if they are cleared from the immediate runner workspace and environment variables. This is because such credentials are often additionally stored in the home directory of the account used to execute the self-hosted runner.

Self-hosted runners can also allow for malicious code to achieve persistence on the runner across workflow runs. This can allow attackers to install persistent watcher routines, which monitor specific locations or environment variables for sensitive information that is only populated after a repository member triggers a privileged workflow step.

Workflows initiated by PRs from forks in public repositories

By default, for public repositories, external accounts can only cause workflows to be executed after a single valid PR or commit from them has been merged. After this, there are no subsequent restrictions on how workflows are executed from that account, regardless of whether they are initiated by PRs from forks. Note that after having the single PR merged, the external account does not need to be granted write access to the repository, so this shouldn't be considered a significant mitigation.

A review of previous workflow runs implies that external contributors must at least first have a previous PR merged to trigger workflow execution from a fork PR. However, it appears that the majority of workflow executors are not direct members of the boundless GitHub organization, which may increase the risks of providing a seemingly legitimate contributor the permissions to directly execute workflows via subsequent PRs. External contributors with prior merged

contributions appear to have the ongoing permissions to trigger workflows such as **main.yaml** workflow, without requiring validation from **boundless** organization members.

This can allow for situations where threat actors make legitimate PRs (e.g., to fix minor documentation details) to public repositories specifically to bypass the initial restriction on workflow execution. This is partially why GitHub documentation strongly recommends against using self-hosted runners in public workflows, particularly those triggered by the **pull_request** event.

Checking out and executing code from PRs

The **main.yaml** workflow's **rust** and **foundry** jobs both execute checked-out code in the context of the self-hosted runner. In the **rust** job's case, this is done directly via the **python license-check.py** step. The **foundry** job does not use a preconfigured **FOUNDRY_PROFILE**, allowing a malicious checked-out **foundry.toml** file which includes **ffi=true**, therefore allowing for command execution via **forge test**. The default foundry profile also includes **ffi=true** currently.

Workflows are configured to assume AWS roles via OIDC

Since the workflows appear to be configured to assume the **gha_oidc_risc0_cache_shared_access** AWS role via OIDC, code execution would provide external threat actors access into the AWS account specified via the assumed role, allowing for lateral movement. The risks of this may be elevated, since the resulting AWS credentials are temporarily cached to text files within the workflow's working directory.

Also note that the AWS role may be exploitable irrespective of the code execution occurring in a self-hosted runner, as OIDC authentication is usually configured on a per-repository/branch basis.

```yaml
// https://github.com/boundless-xyz/boundless/.github/workflows/main.yml
name: Main workflow

on:
  pull_request:
    branches:
      - main
      - 'release-*'
  workflow_dispatch:
...
permissions:
  id-token: write
  contents: read
  pull-requests: read
...
  rust:
    runs-on: [ self-hosted, prod, "${{ matrix.os }}", "${{ matrix.device }}" ]
    strategy:
      # Run only on Linux with CPU.
      matrix:
        include:
          - os: Linux
            feature: default
            device: cpu
    needs: files-changed
    if: needs.files-changed.outputs.main == 'true' || needs.files-
changed.outputs.docs == 'true'
    services:
      postgres:
        image: postgres:latest
        ports:
          - 5432:5432
        env:
          POSTGRES_PASSWORD: password
    steps:
      - name: checkout code
        uses: actions/checkout@v4
        with:
          # Full history is required by license-check.py
          fetch-depth: 0
          submodules: recursive
...
      - name: license check
        run: python license-check.py
```

```
...
  foundry:
    runs-on: [ self-hosted, prod, "${{ matrix.os }}", "${{ matrix.device }}" ]
    strategy:
      # Run only on Linux with CPU.
      matrix:
        include:
          - os: Linux
            feature: default
            device: cpu
    needs: files-changed
    if: needs.files-changed.outputs.main == 'true'
    steps:
      - name: checkout code
        uses: actions/checkout@v4
        with:
          submodules: recursive
...
      - name: forge test
        run: forge test -vvv
        working-directory: contracts
...
  docker:
    runs-on: [ self-hosted, prod, "${{ matrix.os }}", "${{ matrix.device }}" ]
    strategy:
      # Run only on Linux with CPU.
      matrix:
        include:
          - os: Linux
            feature: default
            device: cpu
    needs: files-changed
    if: needs.files-changed.outputs.docker == 'true'
    steps:
      - name: checkout code
        uses: actions/checkout@v4
        with:
          submodules: recursive

       // NOTE: this may only prevent the docker login step, not the subsequent
steps in the Docker job
      - name: Login to Docker Hub
        if: github.event.pull_request.head.repo.full_name ==
github.event.pull_request.base.repo.full_name
```

```
        uses: docker/login-action@v3
...
      - name: Fetch CI AWS Creds
        id: aws-creds
        uses: aws-actions/configure-aws-credentials@v4
        with:
          aws-region: 'us-west-2'
          role-to-assume: 'arn:aws:iam::083632199359:role/
gha_oidc_risc0_cache_shared_access'
          output-credentials: true

      - name: create ci creds file
        run: |
          echo "[default]" > ./dockerfiles/ci-cache-creds.txt
          echo "aws_access_key_id=${{ steps.aws-creds.outputs.aws-access-key-id
}}" >> ./dockerfiles/ci-cache-creds.txt && \
          echo "aws_secret_access_key=${{ steps.aws-creds.outputs.aws-secret-
access-key }}" >> ./dockerfiles/ci-cache-creds.txt && \
          echo "aws_session_token=${{ steps.aws-creds.outputs.aws-session-token
}}" >> ./dockerfiles/ci-cache-creds.txt
...
```

```yaml
//https://github.com/boundless-xyz/boundless/blob/
a7b3fc42c8ccce828f1d46c176bebdcd6d5b7af3/.github/workflows/release.yml
name: Contracts


on:
  pull_request:
    branches: [main]
...
permissions:
  contents: read
  pull-requests: write

env:
  GITHUB_TOKEN: ${{ secrets.GITHUB_TOKEN }}
  FOUNDRY_VERSION: v1.2.2

jobs:
  contracts-changed:
    runs-on: ubuntu-latest
    outputs:
      src: ${{ steps.changes.outputs.src }}
      foundry: ${{ steps.changes.outputs.foundry }}
    steps:
      - name: checkout code
        uses: actions/checkout@v4

      - uses: dorny/paths-filter@v3.0.0
        id: changes
        with:
          filters: |
            src:
              - 'contracts/src/**'
            foundry:
              - 'foundry.toml'
            test:
              - 'contracts/test/**'

  gas-snapshots-check:
    runs-on: [self-hosted, Linux, X64, prod, cpu]
    needs: contracts-changed
    if: ${{ needs.contracts-changed.outputs.src == 'true' || needs.contracts-
changed.outputs.foundry == 'true' }}
    steps:
```

```
...
      - name: checkout code
        uses: actions/checkout@v4
        with:
          submodules: recursive
...
      - name: Ensure gas snapshots have been updated. Reminder to run `forge
test --isolate`.
        run: FORGE_SNAPSHOT_CHECK=true forge test --isolate
```

```
//https://github.com/boundless-xyz/boundless/blob/
a7b3fc42c8ccce828f1d46c176bebdcd6d5b7af3/.github/workflows/release.yml#L1-L72
name: Release



on:
  push:
    tags:
      - "v*.*.*"
  pull_request:
    branches: ["release-*"]
  workflow_dispatch:
    inputs:
      chain_key:
        description: "chain key to use for testing"
        required: false
        default: 'ethereum-sepolia-prod'
...
jobs:
  deployment:
    runs-on: [ self-hosted, prod, "${{ matrix.os }}", "${{ matrix.device }}" ]
    strategy:
      # Run only on Linux with CPU.
      matrix:
        include:
          - os: Linux
            feature: default
            device: cpu
...
```

Remediation:

Limit usage of self-hosted runners

The primary recommendation is to use GitHub-hosted, ephemeral runners in the affected workflows. This would both limit access to internal systems, while guaranteeing a fresh execution environment for each workflow. It is recommended to restrict self-hosted runners to the bare minimum required usage in all other workflows, even in private repositories.

Enable the "Require approval for all external contributors" setting for public repositories

If not already configured, it is strongly recommended to enable "Require approval for all external contributors" under Settings → Actions → General for the affected repository. This will require all workflow initiations from fork pull requests to be manually approved for non-member accounts.

Prevent workflows from PRs from executing checked out code

If possible, consider adding additional checks to prevent workflow steps which execute checked-out code from external PRs, with a check similar to below.

```
// require the following to hold if the step is to be executed
github.event.pull_request.head.repo.full_name ==
github.event.pull_request.base.repo.full_name
```

Use restricted FOUNDRY_PROFILE when running forge test

Where possible, consider using **FOUNDRY_PROFILE** settings which explicitly disables **ffi** in all **forge test** routines. This profile can function as a sandbox to prevent command execution when it is necessary to run forge tests on checked out test suites.

Commentary from the client:

"Approval already required before executing workflows from external PRs, changed severity to low."

RISC5-8 | Missing Payment Guarantee for Unlocked or Post-Lock Fulfillments

Acknowledged

Severity:	Low	Probability:	Rare	Impact:	Low

Path:

contracts/src/BoundlessMarket.sol

Description:

The functions **_fulfillAndPayWasLocked** and **_fulfillAndPayNeverLocked** attempt to transfer the client's payment only at fulfillment time. If the client's balance is lower than the required amount (**clientOwes** or **price**), the function returns an **InsufficientBalance** error and no funds are sent to the prover. The request is still marked as **fulfilled**, preventing any future collection attempts. An uncooperative client can withdraw funds after the request is posted (or after the lock expires) but before the proof is delivered, causing the prover to complete the work without compensation. This behavior reduces economic incentives and may discourage provers from serving unlocked or expired-lock requests.

```
    function _fulfillAndPayWasLocked(
        ...
    ) internal returns (bytes memory paymentError) {
        ...
        if (!fulfilled) {
            accounts[client].setRequestFulfilled(idx);
            emit RequestFulfilled(id, assessorProver, fill);
        }
        ...
        uint96 lockPrice = lock.price;
        if (price > lockPrice) {
            uint96 clientOwes = price - lockPrice;
            if (clientAccount.balance < clientOwes) {
                return abi.encodeWithSelector(InsufficientBalance.selector,
client);
            }
            unchecked {
                clientAccount.balance -= clientOwes;
            }
        }
        ...
```

```
    }

    function _fulfillAndPayNeverLocked(
        ...
    ) internal returns (bytes memory paymentError) {
        ...
        if (fulfilled) {
            return abi.encodeWithSelector(RequestIsFulfilled.selector,
RequestId.unwrap(id));
        }
        ...
        if (clientAccount.balance < price) {
            return abi.encodeWithSelector(InsufficientBalance.selector,
client);
        }
        ...
    }
```

Remediation:

Consider adding a payment guarantee for these code paths. Possible approaches include locking the full payment earlier (e.g., during **priceRequest**), reserving the amount in escrow, or allowing fulfillment to revert entirely when the client's balance is insufficient. Any solution should ensure that a valid proof either results in payment or does not finalize the request state.

Commentary from the client:

"Our recommendation to provers is that they consider this risk when fulfilling orders without locking. A conservative approach is to always lock orders before trying to fulfill them. This is the default behavior of the prover node (broker). Another option is to wrap the fulfill call in a smart contract proxy that checks for payment errors, and reverts on payment failure. When combined with a private RPC service that ensures reverting transactions do not get included in a block, this can prevent spending gas on transactions that do not result in payment, and conceal the proof from the client."

RISC5-9 | Minor Consistency & Code-Hygiene Observations

Acknowledged

Severity:	Informational	Probability:	Rare	Impact:	Informational

Description:

The audit found several small inconsistencies and leftovers that don't affect security but reduce clarity and monitoring.

- **setImageUrl** changes storage but emits no event, so front-ends cannot detect the update.
- **(burnValue);** in **slash** is an unnecessary no-op and can simply be removed.
- The contract accepts up to 65 535 fills (**> uint16.max** check) whereas the assessor guest rejects 65 535 exactly (**>= u16::MAX**).
- On-chain signature checks use OpenZeppelin's high-s enforcement; the guest still accepts high-s values.
- Several read-only/payment helpers are declared **public** yet never called internally: **requestIsFulfilled**, **withdrawFromStakeTreasury**, **balanceOfStake**, **withdrawStake**, **withdrawFromTreasury**, **balanceOf**, **withdraw**.

hexens x RISC ZERO