Veridise

Auditing Report

Hardening Blockchain Security with Formal Methods

FOR

RISC ZERO

Boundless Market (Beta)

Veridise Inc.

April 4, 2025

► **Prepared For:**

RISC Zero
https://risczero.com/

► **Prepared By:**

Victor Faltings
Petr Susil
Kostas Ferles

► **Contact Us:**

contact@veridise.com

► **Version History:**

Apr. 4 2025	V2
Mar. 31 2025	V1

Contents

1 Executive Summary

From Mar. 10, 2025 to Mar. 26, 2025, RISC Zero engaged Veridise to conduct a security assessment of their Boundless Market (Beta). The security assessment covered the main on-chain and off-chain components of the Boundless Market (Beta). The scope of this security review spans across two different repositories, boundless and risc0-ethereum. The first repository contains the smart contracts and zkVM application of the Boundless Market (Beta). The second repository contains smart contracts and off-chain logic of a set verifier that is used by the Boundless Market (Beta). Veridise conducted the assessment over 9 person-weeks, with 3 security analysts reviewing the project over 3 weeks on commit `f0e5fc4` for boundless and commit `71de107` for risc0-ethereum. 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 Boundless Market (Beta) facilitates a decentralized process for generating and verifying zero-knowledge proofs through the following key stages:

- **Program Development**: An application developer (requestor) creates a program using the RISC Zero zkVM, which can generate zero-knowledge proofs of its execution.
- **Proof Request Submission**: The requestor submits a proof request to the Boundless Market (Beta), specifying the program, input data, and an offer detailing payment terms, including minimum and maximum prices, bidding start time, ramp-up period, timeout duration, and lock-in stake.
- **Prover Bidding**: Provers evaluate the proof request and participate in a reverse Dutch auction, where the price increases over time within the specified range. A prover can bid by accepting the current price, locking in the request, and committing the required stake, which may be forfeited if the proof is not delivered within the timeout period.
- **Proof Request Submission**: The requestor submits a proof request to the Boundless Market (Beta), specifying the program, input data, and an offer detailing payment terms, including minimum and maximum prices, bidding start time, ramp-up period, timeout duration, and lock-in stake.
- **Proof Generation**: The selected prover generates the proof using their computational resources. To optimize efficiency, provers may batch multiple proof requests and produce an aggregated proof, reducing on-chain verification costs.
- **Proof Settlement**: The prover submits the aggregated proof to the Boundless Market (Beta)'s smart contract. Upon successful verification, the contract releases the payment to the prover and emits an event signaling the requestor that the proof has been fulfilled.
- **Proof Utilization**: The requestor retrieves the proof, which includes the public output (journal) and the cryptographic proof (seal). The seal, often a Merkle inclusion proof within an aggregated proof, can be verified on-chain using the RiscZeroVerifierRouter, allowing seamless integration into the requestor's application.

This lifecycle enables efficient, decentralized proof generation and verification, allowing developers to leverage zero-knowledge proofs without significant hardware investments, while incentivizing provers to contribute computational resources.

Code Assessment. The Boundless Market (Beta) developers provided the source code of the Boundless Market (Beta) contracts for the code review. The source code appears to be mostly original code written by the Boundless Market (Beta) developers. It contains some documentation in the form of READMEs and documentation comments on functions and storage variables. To facilitate the Veridise security analysts understanding of the code, the Boundless Market (Beta) developers shared a documentation page* with a high-level overview of the protocol as well as a short description of their overall design and key concepts.

The source code contained a test suite, which the Veridise security analysts noted covered most critical routes of the protocol and achieved satisfactory coverage (more than 80% for critical components). However, the Veridise security analysts also noted a lack of negative tests.

Summary of Issues Detected. The security assessment uncovered 7 issues, 3 of which are assessed to be of high or critical severity by the Veridise analysts. Specifically, the review uncovered two critical issues that allowed malicious provers to charge an arbitrary client for any request they fulfilled (V-RISC0-VUL-001) and verify bogus statements using the set verifier (V-RISC0-VUL-004). It also uncovered a high-severity reentrancy vulnerability that allowed clients to withdraw funds before they were charged for a delivered proof (V-RISC0-VUL-003). The Veridise analysts also identified 2 medium-severity issues, which allowed provers to deliver bogus proofs as well as 1 low-severity issue, 0 warnings, and 1 informational finding. The Boundless Market (Beta) developers have acknowledged most of the reported issues and are currently working on resolving them.

Recommendations. After conducting the assessment of the protocol, the security analysts had a few suggestions to improve the Boundless Market (Beta).

Document incentives for participants. The Boundless Market (Beta) provides different incentives to clients and provers to participate in the protocol. However, the Veridise analysts notice that several of these incentives can only be understood if one reads the codebase. It will significantly simplify user onboarding if the incentive and reward structure is more clearly documented.

Improve negative testing. Several of the critical/high-severity issues that were discovered by this review could have been caught by writing negative tests. It is recommended to improve the existing set of negative tests with even more corner cases, especially for security-critical components of the architecture.

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.

* Available at `https://docs.beboundless.xyz`

2 Project Dashboard

Table 2.1: Application Summary.

Name	Version	Type	Platform
Boundless Market (Beta)	`f0e5fc4,71de107`	Solidity, Rust	Ethereum, RISC Zero

Table 2.2: Engagement Summary.

Dates	Method	Consultants Engaged	Level of Effort
Mar. 10–Mar. 26, 2025	Manual & Tools	3	9 person-weeks

Table 2.3: Vulnerability Summary.

Name	Number	Acknowledged	Fixed
Critical-Severity Issues	1	1	1
High-Severity Issues	2	2	2
Medium-Severity Issues	2	2	2
Low-Severity Issues	1	1	0
Warning-Severity Issues	0	0	0
Informational-Severity Issues	1	1	1
TOTAL	7	7	6

Table 2.4: Category Breakdown.

Name	Number
Data Validation	3
Reentrancy	1
Logic Error	1
Usability Issue	1
Maintainability	1

3 Security Assessment Goals and Scope

3.1 Security Assessment Goals

The engagement was scoped to provide a security assessment of Boundless Market (Beta)'s code. During the assessment, the security analysts aimed to answer questions such as:

- Can honest provers always bid for valid requests?
- Can users always submit requests?
- Can honest provers always submit proofs?
- Can users always be refunded for proof requests that cannot be fulfilled?
- Will dishonest provers (or ones that miss a deadline) always be slashed?
- Can malicious provers get paid more than they were supposed to?
- Does the Boundless market perform proper accounting for all entities?
- Does the on-chain logic perform proper data validation?
- Do the off-chain components correctly build and maintain batches of proofs?
- Is the protocol vulnerable to any common vulnerabilities.

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 techniques:

- *Static analysis.* To identify potential common vulnerabilities, security analysts leveraged Veridise's custom smart contract analysis tool Vanguard. 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 files from each repository.

Repository boundless:

- `contracts/src/BoundlessMarket.sol`
- `contracts/src/BoundlessMarketCallback.sol`
- `contracts/src/HitPoints.sol`
- `contracts/src/IBoundlessMarket.sol`
- `contracts/src/IBoundlessMarketCallback.sol`
- `contracts/src/IHitPoints.sol`
- `contracts/src/libraries/BoundlessMarketLib.sol`
- `contracts/src/libraries/MerkleProofish.sol`
- `contracts/src/libraries/types/Account.sol`
- `contracts/src/libraries/types/AssessorCallback.sol`
- `contracts/src/libraries/types/AssessorJournal.sol`

- `contracts/src/libraries/types/AssessorReceipt.sol`
- `contracts/src/libraries/types/Callback.sol`
- `contracts/src/libraries/types/Fulfillment.sol`
- `contracts/src/libraries/types/FulfillmentContext.sol`
- `contracts/src/libraries/types/Input.sol`
- `contracts/src/libraries/types/Offer.sol`
- `contracts/src/libraries/types/Predicate.sol`
- `contracts/src/libraries/types/ProofRequest.sol`
- `contracts/src/libraries/types/RequestId.sol`
- `contracts/src/libraries/types/RequestLock.sol`
- `contracts/src/libraries/types/Requirements.sol`
- `contracts/src/libraries/types/Selector.sol`
- `crates/assessor/src/lib.rs`
- `crates/guest/assessor/assessor-guest/src/main.rs`

Repository risc0-ethereum:

- `contracts/src/RiscZeroSetVerifier.sol`
- `crates/aggregation/guest/set-builder/src/main.rs`
- `crates/aggregation/guest/src/lib.rs`
- `crates/aggregation/src/lib.rs`
- `crates/aggregation/src/receipt.rs`

Methodology. Veridise security analysts reviewed the reports of previous audits for Boundless Market (Beta), inspected the provided tests, and read the Boundless Market (Beta) documentation. They then began a review of the code assisted by the Vanguard static analyzer.

During the security assessment, the Veridise security analysts regularly met with the Boundless Market (Beta) 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.

The impact of a vulnerability is evaluated according to the Table 3.3:

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

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 Trust Model

4.1 Operational Assumptions.

In addition to assuming that any out-of-scope components behave correctly, Veridise analysts assumed the following properties held when modeling security for Boundless Market (Beta)

- Consumers of events generated by the Boundless Market (Beta) properly verify client signatures before taking actions.
- Provers and clients have a basic understanding of the incentives and reward structure of the protocol.

4.2 Privileged Roles.

Roles. This section describes in detail the specific roles present in the system, and the actions each role is trusted to perform. The roles are grouped based on two characteristics: privilege-level and time-sensitivity. *Highly-privileged* roles may have a critical impact on the protocol if compromised, while *limited-authority* roles have a negative, but manageable impact if compromised. Time-sensitive *immediate-action* roles may be required to perform actions quickly based on real-time monitoring, while *delayed-action* roles perform actions like deployments and configurations which can be planned several hours or days in advance. Additionally, some roles may be required to perform different actions with different time-sensitivities.

During the review, Veridise analysts assume that the role operators perform their responsiblities as intended. Protocol exploits relying on the below roles acting outside of their privileged scope are considered outside of scope.

- Highly-privileged, immediate-action roles:
  - The owner of the `HitPoints` contract can revoke the `HitPoints.MINTER` and `HitPoints.AUTHORIZED_TRANSFER` roles from a user.
  - The `HitPoints.MINTER` role can mint ERC-20 staking tokens.
- Highly-privileged, delayed-action roles:
  - The owner of the `BoundlessMarket` contract can upgrade the implementation of the market to a new version.
  - The owner of the `HitPoints` contract can grant the `HitPoints.MINTER` and `HitPoints.AUTHORIZED_TRANSFER` roles to other users. They can also transfer ownership of the `HitPoints` contract to another address.
- Limited-authority, immediate-action roles:
  - Users with the `HitPoints.AUTHORIZED_TRANSFER` role can transfer tokens to one another.

Operational Recommendations. Highly-privileged, delayed-action operations should be operated by a multi-sig contract or decentralized governance system. These operations should be guarded by a timelock to ensure there is enough time for incident response. Highly-privileged, immediate-action operations should be tested in example scenarios to ensure the role operators are available and ready to respond when necessary.

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:

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

5 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 5.1 summarizes the issues discovered:

Table 5.1: Summary of Discovered Vulnerabilities.

ID	Description	Severity	Status
V-RISC0-VUL-001	The BoundlessMarket does not validate . . .	Critical	Fixed
V-RISC0-VUL-002	BoundlessMarket allows delivery of bogus . . .	High	Fixed
V-RISC0-VUL-003	Reentrancy vulnerability for fulfillments . . .	High	Fixed
V-RISC0-VUL-004	The set verifier accepts non-leaf nodes	Medium	Fixed
V-RISC0-VUL-005	No domain separation between client and . . .	Medium	Fixed
V-RISC0-VUL-006	Provers may not be compensated for . . .	Low	Acknowledged
V-RISC0-VUL-007	Code quality improvements	Info	Fixed

5.1 Detailed Description of Issues

5.1.1 V-RISC0-VUL-001: The BoundlessMarket does not validate RequestIds during fulfillment

Severity	Critical	**Commit**	f0e5fc4
Type	Data Validation	**Status**	Fixed
File(s)	`BoundlessMarket.sol`		
Location(s)	functions fulfill, fulfillBatch, and _fulfillAndPay		
Confirmed Fix At	`https://github.com/boundless-xyz/boundless/pull/441`, b9d7c9d		

In order to fulfill proof requests, a prover must provide a set of fulfillments as represented by the following struct:

```
struct Fulfillment {
    /// @notice ID of the request that is being fulfilled.
    RequestId id;
    /// @notice EIP-712 digest of request struct.
    bytes32 requestDigest;
    // [VERIDISE]: ommiting rest of fields
```

Snippet 5.1: Snippet from `struct Fulfillment`

The intended invariant that every object of this struct must satisfy is that the `requestDigest` corresponds to a proof request whose `RequestId` is `id`. However, none of the fulfillment functions (i.e., `fulfill` and `fulfillBatch`) validate that the provided id corresponds to a proof request with digest requestDigest.

Impact This missing check gives attackers the liberty to choose arbitrary `RequestIds` when fulfilling requests. Since the provided `RequestId` is used to determine which account to be charged (see attached snippet below), an attacker can steal funds from users by manipulating the value of the fulfillment's `id`.

```
// [VERIDISE]: Fetching the client account
RequestId id = fill.id;
(address client, uint32 idx) = id.clientAndIndex();
Account storage clientAccount = accounts[client];
(bool locked, bool fulfilled) = clientAccount.requestFlags(idx);
// [VERIDISE]: Charging the account depending on the value of locked.
if (locked) {
    RequestLock memory lock = requestLocks[id];
    if (lock.lockDeadline >= block.timestamp) {
        paymentError = _fulfillAndPayLocked(lock, id, client, idx, fill.requestDigest
    , fulfilled, prover);
    } else {
        paymentError = _fulfillAndPayWasLocked(lock, id, client, idx, fill.
    requestDigest, fulfilled, prover);
    }
} else {
```

```
    paymentError = _fulfillAndPayNeverLocked(id, client, idx, fill.requestDigest,
    fulfilled, prover);
}
```

Snippet 5.2: Snippet from `_fulfillAndPay`

Proof of Concept The following foundry test demonstrates the issue by shuffling the request ids of a request batch. It is recommended to add this test (or a variant of it) as a negative test in the test suite.

```
function testFulfillBatchUnLockedRequestsPOC() public {
    // Provide a batch definition as an array of clients and how many requests each
    submits.
    uint256[5] memory batch = [uint256(1), 2, 1, 3, 1];
    uint256 batchSize = 0;
    for (uint256 i = 0; i < batch.length; i++) {
        batchSize += batch[i];
    }
    ProofRequest[] memory requests = new ProofRequest[](batchSize);
    bytes[] memory journals = new bytes[](batchSize);
    bytes[] memory signatures = new bytes[](batchSize);
    uint256 idx = 0;
    for (uint256 i = 0; i < batch.length; i++) {
        Client client = getClient(i);

        for (uint256 j = 0; j < batch[i]; j++) {
            ProofRequest memory request = client.request(uint32(j));

            requests[idx] = request;
            journals[idx] = APP_JOURNAL;
            signatures[idx] = client.sign(request);
            idx++;
        }
    }

    (Fulfillment[] memory fills, AssessorReceipt memory assessorReceipt) =
        createFillsAndSubmitRoot(requests, journals, testProverAddress);

        // Swap first two IDs
    RequestId id0 = fills[0].id;
    fills[0].id = fills[1].id;
    fills[1].id = id0;

    vm.warp(requests[0].offer.timeAtPrice(uint256(1.5 ether)));
    for (uint256 i = 0; i < fills.length; i++) {
        vm.expectEmit(true, true, true, true);
        emit IBoundlessMarket.RequestFulfilled(fills[i].id);
        vm.expectEmit(true, true, true, false);
        emit IBoundlessMarket.ProofDelivered(fills[i].id);
    }

    boundlessMarket.priceAndFulfillBatch(requests, signatures, fills, assessorReceipt
    );
```

```
    vm.snapshotGasLastCall(string.concat("fulfillBatch: a batch of ", vm.toString(
    batchSize)));

    for (uint256 i = 0; i < fills.length; i++) {
        // Check that the proof was submitted
        expectRequestFulfilled(fills[i].id);
    }

    expectMarketBalanceUnchanged();
}
```

Snippet 5.3: Proof of concept.

Recommendation There are two ways to resolve the issue:

1. Include the fulfillment IDs in the `AssessorJournal`. This will prevent malicious provers from tampering with the request ids, since they will be committed in the accompanied ZK proof.
2. Modify the `Fulfillment` struct to contain the entire `ProofRequest` instead of the `id` and the `requestDigest` (see snippet below). The modified struct has all the information to generate the now removed `requestDigest` and provers will not be able to tamper with the `id` since this will result into an invalid `requestDigest`.

```
struct Fulfillment {
    ProofRequest proofRequest;
    bytes32 imageId;
    bytes journal;
    bytes seal;
}
```

Snippet 5.4: Recommended new version of `Fulfillment`

Developer Response The developers have changed the logic so that the leaves commit to the request ID, the request digest and the claim digest.

5.1.2 V-RISC0-VUL-002: BoundlessMarket allows delivery of bogus fulfillments

Severity	High	**Commit**	f0e5fc4
Type	Data Validation	**Status**	Fixed
File(s)	`BoundlessMarket.sol`		
Location(s)	functions verifyBatchDelivery		
Confirmed Fix At	`https://github.com/boundless-xyz/boundless/pull/441`, b9d7c9d		

When verifying a batch of fulfillments, the `BoundlessMarket` eventually calls `verifyBatchDelivery` that performs the following:

1. Verifies all application receipts.
2. Creates the `batchRoot` from all `claimDigests`.
3. Verifies the `assessorReceipt`.

The set of claimDigests are connected to the `assessorReceipt` via the `batchRoot` which is part of the `AssessorJournal` (see snippet below). However, because the tree is constructed with a commutative hash function (see `MerkleProofish.sol`), its leaves can be permuted in multiple ways while resulting in the same `batchRoot`.

```
// Verify the application receipts.
for (uint256 i = 0; i < fills.length; i++) {
    Fulfillment calldata fill = fills[i];

    requestDigests[i] = fill.requestDigest;
    claimDigests[i] = ReceiptClaimLib.ok(fill.imageId, sha256(fill.journal)).digest()
    ;

    // [VERIDISE]: verifying application receipts
    // If the requestor did not specify a selector, we verify with
    DEFAULT_MAX_GAS_FOR_VERIFY gas limit.
    // This ensures that by default, client receive proofs that can be verified
    cheaply as part of their applications.
    if (!hasSelector[i]) {
        VERIFIER.verifyIntegrity{gas: DEFAULT_MAX_GAS_FOR_VERIFY}(Receipt(fill.seal,
    claimDigests[i]));
    } else {
        VERIFIER.verifyIntegrity(Receipt(fill.seal, claimDigests[i]));
    }
}

// [VERIDISE]: creating batchRoot from claimDigests
bytes32 batchRoot = MerkleProofish.processTree(claimDigests);

// Verify the assessor, which ensures the application proof fulfills a valid request
    with the given ID.
// NOTE: Signature checks and recursive verification happen inside the assessor.
bytes32 assessorJournalDigest = sha256(
    abi.encode(
        AssessorJournal({
            requestDigests: requestDigests,
```

```
            root: batchRoot,
            callbacks: assessorReceipt.callbacks,
            selectors: assessorReceipt.selectors,
            prover: assessorReceipt.prover
        })
    )
);
// [VERIDISE]: verifying the assessorReceipt
// Verification of the assessor seal does not need to comply with
    DEFAULT_MAX_GAS_FOR_VERIFY.
VERIFIER.verify(assessorReceipt.seal, ASSESSOR_ID, assessorJournalDigest);
```

Snippet 5.5: Snippet from `verifyBatchDelivery`

Impact A malicious prover can deliver bogus batches of fulfillments by permuting their `imageId` and `journal` fields. This can also affect clients who make use of the callbacks feature.

Proof of Concept The following foundry test demonstrates the issue by shuffling the fields of a request batch. It is recommended to add this test (or a variant of it) as a negative test in the test suite.

```
function testShuffledFulfillBatchRequestsPOC() public {
    // Provide a batch definition as an array of clients and how many requests each
    submits.
    uint256[5] memory batch = [uint256(1), 2, 1, 3, 1];
    uint256 batchSize = 0;
    for (uint256 i = 0; i < batch.length; i++) {
        batchSize += batch[i];
    }
    ProofRequest[] memory requests = new ProofRequest[](batchSize);
    bytes[] memory journals = new bytes[](batchSize);
    uint256 expectedRevenue = 0;
    uint256 idx = 0;
    for (uint256 i = 0; i < batch.length; i++) {
        Client client = getClient(i);

        for (uint256 j = 0; j < batch[i]; j++) {
            ProofRequest memory request = client.request(uint32(j));

            // TODO: This is a fragile part of this test. It should be improved.
            uint256 desiredPrice = uint256(1.5 ether);
            vm.warp(request.offer.timeAtPrice(desiredPrice));
            expectedRevenue += desiredPrice;

            boundlessMarket.lockRequestWithSignature(request, client.sign(request),
    testProver.sign(request));

            requests[idx] = request;
            journals[idx] = APP_JOURNAL;
            idx++;
        }
    }
```

```
30
31      (Fulfillment[] memory fills, AssessorReceipt memory assessorReceipt) =
32          createFillsAndSubmitRoot(requests, journals, testProverAddress);
33
34      bytes32 imageId0 = fills[0].imageId;
35      bytes memory journal0 = fills[0].journal;
36
37      fills[0].imageId = fills[1].imageId;
38      fills[1].imageId = imageId0;
39
40      fills[0].journal = fills[1].journal;
41      fills[1].journal = journal0;
42
43      for (uint256 i = 0; i < fills.length; i++) {
44          vm.expectEmit(true, true, true, true);
45          emit IBoundlessMarket.RequestFulfilled(fills[i].id);
46          vm.expectEmit(true, true, true, false);
47          emit IBoundlessMarket.ProofDelivered(fills[i].id);
48      }
49      boundlessMarket.fulfillBatch(fills, assessorReceipt);
50      vm.snapshotGasLastCall(string.concat("fulfillBatch: a batch of ", vm.toString(
        batchSize)));
51
52      for (uint256 i = 0; i < fills.length; i++) {
53          // Check that the proof was submitted
54          expectRequestFulfilled(fills[i].id);
55      }
56
57      testProver.expectBalanceChange(int256(uint256(expectedRevenue)));
58      expectMarketBalanceUnchanged();
59  }
```

Recommendation Ensure the journal prevents reordering of proofs. This can be achieved by one of the following

1. include array `claimDigests` into journal, and as part of `verifyBatchDelivery` ensure `(claimDigests[i] == claimDigest)`
2. compute digest over elements `claimDigests[i]` and include in the journal
3. avoid using "commutative" hash function

Developer Response The developers have fixed the issue by modifying the assessor's journal in a way that prevents reorderings.

5.1.3 V-RISC0-VUL-003: Reentrancy vulnerability for fulfillments with callbacks

Severity	High	**Commit**	f0e5fc4
Type	Reentrancy	**Status**	Fixed
File(s)	`BoundlessMarket.sol`		
Location(s)	functions fulfill and fulfillBatch		
Confirmed Fix At	`https://github.com/boundless-xyz/boundless/pull/442`, 1e77e57		

The `fulfill` and `fulfillBatch` functions are vulnerable to a **reentrancy attack** due to an unsafe external call made via `_executeCallback` before executing `_fulfillAndPay`. This order of execution allows a malicious client to exploit the callback mechanism to re-enter the contract and manipulate their funds, allowing them to withdraw their balance before the prover gets paid for currently unlocked requests.

The root cause of the vulnerability is that `_executeCallback` is executed before `_fulfillAndPay` in both functions. Additionally, `_fulfillAndPay` does not revert when the client lacks sufficient funds, which guarantees the feasibility of the exploit for unlocked requests.

```
1  // Execute the callback with the associated fulfillment information.
2  // Note that if any of the following fulfillment logic fails, the entire transaction
       will
3  // revert including this callback.
4  if (assessorReceipt.callbacks.length > 0) {
5      AssessorCallback memory callback = assessorReceipt.callbacks[0];
6      _executeCallback(fill.id, callback.addr, callback.gasLimit, fill.imageId, fill.
       journal, fill.seal);
7  }
8
9  paymentError = _fulfillAndPay(fill, assessorReceipt.prover);
10 emit ProofDelivered(fill.id);
```

Snippet 5.6: Snippet from `fulfill`

Impact A malicious client can use reentrancy to withdraw their funds before `_fulfillAndPay` executes, preventing the prover from being compensated.

Recommendation To mitigate this attack, `_fulfillAndPay` should be executed **before** `_executeCallback`. This ensures that the prover is compensated before the external call is made.

Developer Response The developers have changed the order of operations so that `_fulfillAndPay` is called before `_executeCallback` when delivering proofs.

5.1.4 V-RISC0-VUL-004: The set verifier accepts non-leaf nodes

Severity	Medium	**Commit**	71de107
Type	Data Validation	**Status**	Fixed
File(s)	`RiscZeroSetVerifier.sol`		
Location(s)	functions _verifyIntegerity		
Confirmed Fix At	`https://github.com/risc0/risc0-ethereum/pull/526`, a123a34		

In the `RiscZeroSetVerifier`, the `_verifyIntegrity` method checks the validity of a submitted `seal` against a `claimDigest` object. The intended behavior of the verifier is to execute successfully when called with a valid `claimDigest` and revert otherwise. A `claimDigest` is valid if it was included in a finalized Merkle Mountain Range (MMR), created by the `set-builder` zkVM guest application.

This verifier expects set inclusion proofs as the cryptographic `seal`. These proofs contain a Merkle proof `path` field as well as an optional `rootSeal`.

- The `path` field is used to compute a Merkle root from the provided `claimDigest`
- If provided, the `rootSeal` field is used to prove the validity of the computed root. If not, the verifier checks its internal mapping to see if the root was submitted beforehand.

```
function _verifyIntegrity(bytes calldata seal, bytes32 claimDigest) internal view {
        // ...
    if (seal.length > 4) {
            setVerifierSeal = abi.decode(seal[4:], (Seal));
    }
        // [VERIDISE]: computing Merkle root with claimDigest as the leaf
        bytes32 root = MerkleProof.processProof(setVerifierSeal.path, claimDigest);
        // ...
}
```

Snippet 5.7: Snippet from `_verifyIntegrity()`

However, there is currently no differentiation between leaf nodes and intermediate nodes in the MMR. That is, a user can provide an intermediate node as a `claimDigest` and a partial path and still obtain a valid root. This means that any intermediate node, as well as the root node itself, are seen as valid `claimDigests` from the point of view of the verifier.

```
function processProof(bytes32[] memory proof, bytes32 leaf) internal pure returns (
    bytes32) {
      bytes32 computedHash = leaf;
      for (uint256 i = 0; i < proof.length; i++) {
              // [VERIDISE]: the leaf nodes are not differentiated from intermediate
    nodes in any way
            computedHash = _hashPair(computedHash, proof[i]);
      }
      return computedHash;
}
```

Snippet 5.8: Snippet from `MerkleProof.processProof()`

Impact As a result of this, an attacker is able to verify invalid `claimDigests` as valid proofs.

Proof of Concept The following test fails when added to the tests of file `contracts/test/RiscZeroSetVerifier.t.sol`. This test is providing an empty proof and the root as a digest, but any intermediate node (with the appropriate path) would result in similar behavior.

```
function test_VerifyIntegrityFailsOnNonLeaf() public {
    bytes32[] memory claimDigests = new bytes32[](3);
    claimDigests[0] = hex"1eaf00";
    claimDigests[1] = hex"1eaf01";
    claimDigests[2] = hex"1eaf02";
    (bytes32 root, ) = TestUtils.computeMerkleTree(claimDigests);
    submitRoot(root);

    TestUtils.Proof memory proof;
    vm.expectRevert();
    setVerifier.verifyIntegrity(
            RiscZeroReceipt({seal: setVerifier.encodeSeal(proof), claimDigest: root})
        );
}
```

Recommendation We recommend two possible fixes for the vulnerability:

- The API of the verifier could be modified to accept the raw claim directly, this way when computing the digest for it it will be differentiated from intermediate nodes
- Keep using the digests as inputs but use a separate hash function (for example with a tag) to differentiate leaf nodes from intermediate nodes (see here for an example). This hashing function would be used both in the `set-builder` guest before a digest is added to the MMR and in the verifier contract before computing the root.

Developer Response The developers have changed the logic to hash the leaves of the MMRs with a tagged hash function.

5.1.5 V-RISC0-VUL-005: No domain separation between client and prover signatures

Severity	Medium	**Commit**	f0e5fc4
Type	Logic Error	**Status**	Fixed
File(s)	`BoundlessMarket.sol`		
Location(s)	functions lockRequestWithSignature		
Confirmed Fix At	`https://github.com/boundless-xyz/boundless/pull/464`, db0d114		

The function `lockRequestWithSignature` accepts as parameters a `request`, `clientSignature`, and `proverSignature`. The validity of a request is verified using the `clientSignature`. Whereas the `proverSignature` is used to retrieve the prover address and lock the request on behalf of the signing prover.

However, in both cases, the same `requestHash` is used to recover the address as there is no domain separation for a client and a prover:
`ECDSA.recover(requestHash, "client/prover"Signature)`

```
function _extractProverAddress(bytes32 requestHash, bytes calldata proverSignature)
    ... returns (address)
{
   return **ECDSA.recover(requestHash, proverSignature)**;
}

function _verifyClientSignature(ProofRequest calldata request, address addr, bytes
    calldata clientSignature) ... returns (bytes32)
{
   bytes32 requestHash = _hashTypedDataV4(request.eip712Digest());
   if (request.id.isSmartContractSigned()) {
      if (IERC1271(addr).isValidSignature(requestHash, clientSignature) != IERC1271.
    isValidSignature.selector) {
          revert IBoundlessMarket.InvalidSignature();
      }
   } else {
      if (**ECDSA.recover(requestHash, clientSignature)** != addr) {
         revert IBoundlessMarket.InvalidSignature();
      }
   }
   return requestHash;
}
```

Snippet 5.9: Snippet from functions `_extractProverAddress` and `_verifyClientSignature`

Impact An attacker can call `lockRequestWithSingature` as follows: `lockRequestWithSignature(request, clientSignature, clientSignature)`, that is providing the `clientSignature` as a `provingSignature`. This will affect accounts that behave as both a client and a prover in the protocol, because they are likely to have positive `balance` and `stakeBalances`. Consequently, the aforementioned call to `lockRequestWithSignature` would lead to a case where the requesting client would also lock their proof request. This could be abused to grieve the client through slashing and delays in fulfillment.

```
function lockRequestWithSignature(
        ProofRequest calldata request,
        bytes calldata clientSignature,
        bytes calldata proverSignature
    ) external
{
   (address client, uint32 idx) = request.id.clientAndIndex();
   bytes32 requestHash = _verifyClientSignature(request, client, clientSignature);
   address prover = _extractProverAddress(requestHash, proverSignature);
   (uint64 lockDeadline, uint64 deadline) = request.validate();

   _lockRequest(request, requestHash, client, idx, prover, lockDeadline, deadline);
}
```

Snippet 5.10: Function `lockRequestWithSignature`

Recommendation

1. Prevent self-locking, put `require(client != prover)` as part of either `_lockRequest` or `lockRequestWithSignature`.
2. Ensure domain separation for client and prover signatures.

Developer Response The developers have adapted the code so that the client and prover are no longer signing the same message.

5.1.6 V-RISC0-VUL-006: Provers may not be compensated for delivered proofs

Severity	Low	Commit	f0e5fc4
Type	Usability Issue	Status	Acknowledged
File(s)	BoundlessMarket.sol		
Location(s)	functions _fulfillAndPayNeverLocked, _fulfillAndPayWasLocked		
Confirmed Fix At	N/A		

Functions `_fulfillAndPayNeverLocked` and `_fulfillAndPayWasLocked` will only pay a prover if the client has sufficient balance to cover the current price (see snippet below). So, provers may not get compensated for delivered proofs even if the client has some unlocked balance.

```
1    function _fulfillAndPayNeverLocked(
2        RequestId id,
3        address client,
4        uint32 idx,
5        bytes32 requestDigest,
6        bool fulfilled,
7        address assessorProver
8    ) internal returns (bytes memory paymentError) {
9        // When never locked, the fulfilled flag _does_ indicate that payment has
     already been transferred,
10       // so we return early here.
11       if (fulfilled) {
12           return abi.encodeWithSelector(RequestIsFulfilled.selector, RequestId.
     unwrap(id));
13       }
14
15       // If no fulfillment context was stored for this request digest (via
     priceRequest),
16       // then payment cannot be processed. This check also serves as an expiration
     check since
17       // fulfillment contexts cannot be created for expired requests.
18       FulfillmentContext memory context = FulfillmentContextLibrary.load(
     requestDigest);
19       if (!context.valid) {
20           return abi.encodeWithSelector(RequestIsExpiredOrNotPriced.selector,
     RequestId.unwrap(id));
21       }
22       uint96 price = context.price;
23
24       Account storage clientAccount = accounts[client];
25       if (!fulfilled) {
26           clientAccount.setRequestFulfilled(idx);
27           emit RequestFulfilled(id);
28       }
29
30       // Deduct the funds from client account.
31       if (clientAccount.balance < price) {
32           return abi.encodeWithSelector(InsufficientBalance.selector, client);
33       }
34       unchecked {
35           clientAccount.balance -= price;
```

```
36          }
37
38          if (MARKET_FEE_BPS > 0) {
39              price = _applyMarketFee(price);
40          }
41          accounts[assessorProver].balance += price;
42      }
```

Snippet 5.11: Snippet from `_fulfillAndPayNeverLocked()`

Recommendation Compensate the prover with `clientAccount.balance` and raise a different payment error for this occasion.

Developer Response The developers acknowledged the behavioral and they will adjust documentation with guidance for clients and provers. Their response can be found below:

> "This is one of a few different conditions under which a prover will not be paid when fulfilling a proof for which they do not hold a lock. We considered providing partial payment to the prover in the case that the client balance is greater than 0 but less than the offer price. However, this raises a challenge of what to do about the remaining unpaid balance (e.g. do we record the unpaid balance in storage and have the prover submit a second transaction?). We decided not to do this as it would add complexity to the number of states an order can exist in, in particular by adding a "partially paid" state in addition to "paid" and "unpaid".
>
> Our recommendation to provers is that they take this into account along with other conditions that might result in a payment error during fulfillment. 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.
>
> We will be updating documentation to more clearly highlight the risks that exist when fulfilling orders without a lock, and include this as one of them."

5.1.7 V-RISC0-VUL-007: Code quality improvements

Severity	Info	**Commit**	f0e5fc4
Type	Maintainability	**Status**	Fixed
File(s)	See issue description		
Location(s)	See issue description		
Confirmed Fix At	`https://github.com/boundless-xyz/boundless/pull/465`, 125ea35		

There are a number of minor code quality issues that were uncovered in the audit. They are listed per file below:

Boundless repo

- `contracts/src/libraries/BoundlessMarketLib.sol`:
  - The `encodeConstructorArgs` function seems to have fallen out of sync with the constructor of the `BoundlessMarket` contract as it is missing the `address stakeTokenContract` argument.
- `contracts/src/BoundlessMarket.sol`:
  - The check on line 610 in `_fulfillAndPayNeverLocked` is not required, since it is already checked on line 596.
  - The `revertWith` function is never used.
  - In the `depositStakeWithPermit` function, the call to the ERC-20 `permit` function is wrapped in a try-catch block. This could lead to confusing error reporting when the transaction later reverts from calling `safeTransferFrom`.
  - In the `slash` function, a portion of ERC-20 tokens are burned on line 714. It is recommended to move this interaction to the end of the function to avoid any reentrancy risks from the underlying token.
  - The `withdrawFromTreasury` function can be simplified with a call to `_withdraw`.
  - In the `slash` function, the `stakeRecipient` variable can be used instead of `lock.prover` and `address(this)` when accessing accounts on lines 725 and and 728.
  - The computation on lines 571-572 could be simplified by computing `uint96 clientOwed = lockPrice - price`.
- `contracts/src/BoundlessMarketCallback.sol`:
  - The documentation of `handleProof` in `IBoundlessMarketCallback` contains the following note:

    > If not called by BoundlessMarket, the function MUST verify the proof before proceeding.

    However, in the implementation of the function, the function reverts if not called by the `BoundlessMarket` and verifies the proof in all cases.
- `contracts/src/HitPoints.sol`:
  - The balance check on line 72 of the `mint` function is redundant with the one performed on line 87 of `_update`
- `crates/assessor/src/lib.rs`:

- The `require_payment` field in the `Fulfillment` struct is never used.
- There is a typo in the type of the `domain` field in `AssessorInput` (`EIP721DomainSaltless` instead of `EIP712DomainSaltless`).
- On line 54 of the `Fulfillment` implementation block, `self.request.client_address()?` can be replaced with the `client_addr` variable.

Risc0-ethereum repo

- `crates/aggregation/src/lib.rs`:
  - There is a typo in the documentation on line 158 ("rooted" is written twice).

Impact There is no security impact, however it is recommended to fix them in order to help future maintainability of the codebase.

Developer Response The developers have fixed most of the reported issues. There are two recommendations which were not implemented, for which appropriate justification was provided:

> "Note there are two recommendations not implemented:
>
> 1. **In depositStakeWithPermit(), the call to the ERC-20 permit() function is wrapped in a try-catch block. This could lead to confusing error reporting when the transaction later reverts from calling safeTransferFrom():** try/catch for permit, we are following the recommendations from OZ here.
> 2. **withdrawFromTreasury() can be simplified with a call to _withdraw():** For withdrawFromTreasury, we withdraw from address(this) but send to msg.sender, so _withdraw can not be used."

Glossary

ERC-20 The famous Ethereum fungible token standard. See `https://eips.ethereum.org/EIPS/eip-20` to learn more. 7

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, 25

Solidity The standard high-level language used to develop smart contracts on the Ethereum blockchain. See `https://docs.soliditylang.org/en/v0.8.19/` to learn more. 25

Vanguard A static analysis tool for Solidity by Veridise. See `https://docs.veridise.com/vanguard/` for more information . 4

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. 25

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