Veridise

Auditing Report

Hardening Blockchain Security with Formal Methods

FOR

Boundless

Boundless Market: Fulfillment Data

Veridise Inc.

September 11, 2025

► **Prepared For:**

Boundless
https://boundless.network

► **Prepared By:**

Benjamin Sepanski
Petr Susil

► **Contact Us:**

contact@veridise.com

► **Version History:**

Sep. 30, 2025	V2 - Incorporated issue fixes
Sep. 23, 2025	V1
Sep. 19, 2025	Initial Draft

Contents

1 Executive Summary

From Sep. 11, 2025 to Sep. 18, 2025, Boundless engaged Veridise to conduct a security assessment of their Boundless Market. The security assessment covered on-chain and off-chain components of the Boundless Market related to introduction of BitVM support. Compared to the previous version, which Veridise has audited previously*, the new version covered an extension to the Boundless Market that introduces support for a Groth16 proof variant used in BitVM, which commits to a single digest BLAKE3 instead of multiple public values. Veridise conducted the assessment over 2 person-weeks, with 2 security analysts reviewing the project over 1 week on commit `63fc761b`. 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. As described in the prior audit report, Boundless Market is an on-chain prover marketplace with two primary roles. *Requesters* post `ProofRequests` to the market. These requests are priced using a dutch auction, in which `provers` bid to lock (and then fulfill) requests.

Each request encodes a concrete statement to be proven using a zkVM, as well as a deadline for request completion. In the original audit, this statement was represented in terms of an image ID and journal, specifying the program and program output, respectively. Provers bid on the requests by locking them at a price and providing some amount of collateral. In order to be paid, they must deliver a proof of the requested claim before the request deadline elapses. Otherwise, they are slashed, forfeiting their collateral.

The changes introduced in the extension lead to a generalization of fulfillment and verification logic. Overall, this change enhances processing assessor commitments within the ZK proof and simplifies handling of commitments within the final proof.

- **Fulfillment Model**: Fulfillments now deliver a claim digest, rather than an image ID and journal. A `ProofRequest` specifies the claim to be proven using a `Predicate`, which is evaluated within the ZK guest to check whether the delivered proof satisfies the requester's requirements. By using a `DigestMatch` or `PrefixMatch` `Predicate`, a requester may force the prover to provide an image ID and journal as `FulfillmentData`, included in the `Fulfillment`. For cases where no RISC Zero image ID and journal may be available (e.g. as in the BitVM case), the requester may instead specify a `ClaimDigestMatch` `Predicate`. Fulfillments satisfying these `Predicates` have no obligation to provide an image ID or journal.
- **Verification Responsibilities**: Verification is split between the zkVM guest and the on-chain verifier. The guest validates request signatures, evaluates `Predicate` conditions against supplied `FulfillmentData`, and packages results into `AssessorCommitment` leaves, but does not validate the proof seal itself. On chain, the `BoundlessMarket` contract invokes the `IRiscZeroVerifier` to check each fulfillment's `seal` against the claimed `claimDigest` (`verifyIntegrity`) and to validate the assessor batch proof against the Merkle root and journal digest (`verify`). This ensures that every `claimDigest` is derived from actual program execution and that assessor commitments are consistent with the verified proof, providing end-to-end soundness.

* The previous audit report, if it is publicly available, can be found on Veridise's website at `https://veridise.com/audits/`

Formerly, each `Fulfillment.seal` was verified *both* on-chain and within the assessor proof. In the upgraded version, the proof is only verified on-chain. More precisely, in the case that a proof is validated using the `RiscZeroSetVerifier`[†], the proof is verified within an aggregation ZK-circuit. Otherwise, the standard RISC Zero verifier verifies the ZK proof directly. In either case, the on-chain checks during fulfillment of a request ensure the proof has been verified.

Code Assessment. The Boundless Market developers provided the source code of the Boundless Market contracts for the code review. The source code appears to be mostly original code written by the Boundless Market 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 developers shared a documentation page[‡] providing a high-level overview of the protocol, including an outline of the system architecture, principal design choices, and core abstractions. In addition, they supplied a supplementary description of recent modifications to the fulfillment data handling logic [§], detailing the updated data flow and its implications for protocol execution.

The source code contained a test suite, which the Veridise security analysts noted covered most critical routes of the protocol and achieved satisfactory coverage.

Summary of Issues Detected. The security assessment uncovered 7 issues, 1 of which is assessed to be of medium severity by the Veridise analysts. Specifically, V-BND-VUL-001 details how the prover can manipulate gas usage to deliberately cause callbacks to client contracts to fail, while still ensuring the overall transaction is successfully executed. The Veridise analysts also identified 1 low-severity issue, V-BND-VUL-002, which identified uncapped gas in ERC-1271 call, allowing unbounded execution of signature verification logic that can be exploited by clients to inflate relayer/prover costs. Additionally, the analysts identified 4 warnings, and 1 informational finding. The Boundless Market developers have fixed all the issues. The Veridise analysts confirmed that these changes resolved all raised concerns.

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

Document guidance on locking requests. Provers should fully validate a request before locking it. The changes to `Fulfillment` data introduce a new class of unfulfillable requests. If an image ID and journal are not provided, then the on-chain logic reverts when a callback is being executed. Consequently, if a callback is requested together with a `ClaimDigestMatch` for which there exists no corresponding `(imageId, journal)`, the request will not be fulfillable.

Consider using attributes on key functions. Apply the `#[must_use]` attribute to critical functions such as `Predicate::eval()}` and `Fulfillment::evaluate_requirements()`. This ensures that callers cannot inadvertently ignore their return values, thereby preventing subtle logic errors and enforcing correct usage at the type system level.

Callback considerations. Clients requesting proofs should refer to the project documentation for guidance on callback procedures associated with proof delivery, including gas limit

[†] https://github.com/risc0/risc0-ethereum/blob/4a0f35dd/contracts/src/RiscZeroSetVerifier.sol
[‡] Available at https://docs.beboundless.xyz
[§] Available at https://github.com/boundless-xyz/boundless/pull/1004#issue-3312017714

considerations, at-least-once delivery semantics, and the need for robust validation of image IDs and journals to ensure callbacks are handled correctly. [¶]

Atomic initialization/upgrade. As highlighted in V-BND-VUL-006, comprehensive validation and sanity checks must be performed during the upgrade process. This ensures a seamless transition and mitigates the risk of inadvertently restricting the contract owner's access or control. Most importantly, however, initialization should be performed atomically with the contract upgrade, ensuring no malicious actors can obtain control of the contract.

[¶] https://docs.boundless.network/developers/tutorials/callbacks#considerations

2 Project Dashboard

Table 2.1: Application Summary.

Name	Version	Type	Platform
Boundless Market	`63fc761b`	Solidity, Rust	Ethereum, RISC-Zero

Table 2.2: Engagement Summary.

Dates	Method	Consultants Engaged	Level of Effort
Sep. 11–Sep. 18, 2025	Manual & Tools	2	2 person-weeks

Table 2.3: Vulnerability Summary.

Name	Number	Acknowledged	Fixed
Critical-Severity Issues	0	0	0
High-Severity Issues	0	0	0
Medium-Severity Issues	1	1	1
Low-Severity Issues	1	1	1
Warning-Severity Issues	4	4	4
Informational-Severity Issues	1	1	1
TOTAL	7	7	7

Table 2.4: Category Breakdown.

Name	Number
Maintainability	3
Denial of Service	1
Usability Issue	1
Data Validation	1
Logic Error	1

3 Security Assessment Goals and Scope

3.1 Security Assessment Goals

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

- Is the protocol vulnerable to any common Solidity security issues, such as reentrancy, flash loan attacks, or front running?
- Is the protocol vulnerable to any common zkVM attacks, such as unchecked journal inputs, unchecked host inputs, uncaught errors, or missing checks on program metadata like exit codes?
- Can provers submit an image ID or proof that does not match the requested claim digest or journal prefix?
- Is there any program path along which a prover's submitted fulfillment seal is not verified?
- Can provers use the new `ClaimDigestMatch` to submit a callback which proves a failed execution of an image ID (i.e. one with non-zero exit code)?
- Can provers validate if a request may be fulfilled before locking it?
- Are fulfillments validated to satisfy all specified requirements?
- Are all requests properly signed and bound to the requester?
- Are the modified `Fulfillment` structs properly committed to by the assessor proof?
- Can requesters still ensure that, when required, they receive an image ID and journal rather than just a claim digest?
- Can a prover or requester abuse the `ClaimDigestMatch` by using arbitrary bytes, rather than the a valid commitment to some claim?
- Are all EIP-712 signatures properly adjusted to handle the new struct definitions?

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, as well as the open-source tool Slither. 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 changes to the following files since the prior Veridise audit, which resolved at commit `f0e5fc49`:

- `contracts/src/BoundlessMarket.sol`
- `contracts/src/IBoundlessMarket.sol`
- `contracts/src/types/AssessorCommitment.sol`
- `contracts/src/types/Fulfillment.sol`
- `contracts/src/types/FulfillmentData.sol`

- `contracts/src/types/Predicate.sol`
- `contracts/src/types/Requirements.sol`
- `crates/assessor/src/lib.rs`
- `crates/boundless-market/src/contracts/mod.rs`
- `crates/guest/assessor/assessor-guest/src/main.rs`

These contain the smart contract and zkVM implementation of the Boundless Market: Fulfillment Data.

Methodology. Veridise security analysts reviewed the reports of previous audits for Boundless Market, inspected the provided tests, and read the Boundless Market documentation. They then began a review of the code assisted by static analyzers. During the security assessment, the Veridise security analysts regularly met with the Boundless Market 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.

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

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

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.

- Provers are assumed to be actively assessing requests and tracking requester misbehavior. This includes executing requests off-chain to ensure the claim digests are satisfiable, validating client signatures, and ignoring requests which require callbacks, but have no valid image ID and journal.
- Out-of-scope code such as the zkVM itself and host-side logic is assumed to operate correctly.

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

The core role in the Boundless Market is the `BoundlessMarket.owner`. The`BoundlessMarket.owner` may upgrade the protocol, withdraw fees and collateral tokens allocated to the protocol from the treasury, and set the image URL. As part of upgrading the protocol, the owner is responsible for correctly configuring the protocol. This includes setting the verifier address, assessor image ID, and collateral token contract.

Operational Recommendations. The singular role comprises several important functions. Consider making a timelock contract the owner so that these privileges are not all granted to a single private key. This will ensure that highly sensitive operations, such as contract upgrades, receive more scrutiny than simple operations like changing the image URL.

Additionally, the Boundless team should make concrete preparations for emergency response. As there is no pause feature, a hasty upgrade in response to an incident may lead to further issues. It is important to create and practice plans for incident response.

As always, highly-privileged, non-emergency 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, emergency 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-BND-VUL-001	Expensive callbacks may be intentionally . . .	Medium	Fixed
V-BND-VUL-002	Uncapped ERC-1271 Call in . . .	Low	Fixed
V-BND-VUL-003	Off-chain signature checks do not enforce . . .	Warning	Fixed
V-BND-VUL-004	Unused/duplicate code	Warning	Fixed
V-BND-VUL-005	All-or-Nothing settlement prevents . . .	Warning	Fixed
V-BND-VUL-006	Missing checks on constructor	Warning	Fixed
V-BND-VUL-007	Out-of-date Solidity version	Info	Fixed

5.1 Detailed Description of Issues

5.1.1 V-BND-VUL-001: Expensive callbacks may be intentionally DoS'ed by provers

Severity	Medium	**Commit**	63fc761
Type	Denial of Service	**Status**	Fixed
Location(s)	`contracts/src/BoundlessMarket.sol:621-633`		
Confirmed Fix At	`https://github.com/boundless-xyz/boundless/pull/1159`, 21f1b21		

When a prover delivers a proof, the contract executes an optional callback. This is done via the `_executeCallback()` function, shown below.

```
/// @notice Execute the callback for a fulfilled request if one is specified
/// @dev This function is called after payment is processed and handles any callback specified in the request
function _executeCallback(
    RequestId id,
    address callbackAddr,
    uint96 callbackGasLimit,
    bytes32 imageId,
    bytes calldata journal,
    bytes calldata seal
) internal {
    try IBoundlessMarketCallback(callbackAddr).handleProof{gas: callbackGasLimit}(imageId, journal, seal) {}
    catch (bytes memory err) {
        emit CallbackFailed(id, callbackAddr, err);
    }
}
```

Due to EIP-150, whenever a call is made, the callee receives at most 63/64 of the caller's remaining gas. Formally:

- Let `A` be the remaining gas available right before the `CALL`.
- Let `G_post` be the gas required to run the catch block and emit CallbackFailed.
- The call forwards at most `63A/64` gas.
- The caller retains at least `A/64` gas.

If the prover arranges `A >= 64 * G_post`, then the outer function *always* has enough gas to log `CallbackFailed`, even if the callback runs out of gas.

A small test (shown below) indicates that `G_post` is around 28k gas. That means the minimum gas in a DoS-able callback, i.e. `63 * G_post`, is around 1.7MGas.

Impact Any callback that requires more than 1.7MGas gas can be griefed by the prover, causing callbacks to fail even when the callback logic would normally succeed. The system will emit `CallbackFailed`, despite the fact that the callback's logic does not cause a revert. Importantly, provers are economically incentivized to provide as little gas as possible, i.e. to perform this gas-based DoS.

Fortunately, callbacks may be re-executed using the fulfilled data, so integrity is not broken. However, delivery by the prover is not guaranteed, and off-chain listeners should take care in their interpretation of the `CallbackFailed` event.

Recommendation One solution is to require a minimum amount of gas to be left, to ensure the prover does not intentionally DoS the callback. Unfortunately, future Ethereum upgrades such as EOF may remove gas introspection. In this case, documentation will be important, as well as considering additional prover incentives for successful callback execution.

Developer Response The developers added a check to ensure the forwarded gas is at least callbackGasLimit.

Gas Profiling The below test can be run to get a rough idea of the gas required. Note that this is likely an overestimate, as it includes some amount of gas from executing the callback.

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.19;

import "forge-std/Test.sol";

contract Callback{
    error Failed();
    function revertCallback() external {
        revert Failed();
    }
}

contract CallbackGasTest is Test {
    Callback cb;
    event CallbackFailed(uint256, address, bytes);

    function setUp() public {
        cb = new Callback();
    }

    function testMeasureCatchPathGasLeft() public {
        uint96 callbackGasLimit = 200_000;
        uint256 requestId = 0;
        uint256 gasAfterCatch;
        uint256 gasBefore = gasleft();
        try cb.revertCallback{gas: callbackGasLimit}() {
            require(false);
        } catch (bytes memory err) {
            emit CallbackFailed(requestId, address(cb), err);
            gasAfterCatch = gasleft();
        }
        console.log("Gas before", gasBefore);
        console.log("Gas after", gasAfterCatch);
        console.log("Gas spent", gasBefore - gasAfterCatch);
    }
}
```

5.1.2 V-BND-VUL-002: Uncapped ERC-1271 Call in lockRequestWithSignature enables gas griefing

Severity	Low	**Commit**	63fc761
Type	Usability Issue	**Status**	Fixed
Location(s)	`contracts/src/BoundlessMarket.sol:908-920`		
Confirmed Fix At	`https://github.com/boundless-xyz/boundless/pull/1160`, 2c44cef		

The `lockRequestWithSignature(...)` function validates client signatures via `_verifyClientSignatureAndExtractProverAddress(...)`, which issues an uncapped ERC-1271 `isValidSignature` call and bubbles any reverts. By contrast, `lockRequest(...)` calls `_verifyClientSignature(...)`, which caps the gas forwarded to the ERC-1271 contract (`ERC1271_MAX_GAS_FOR_CHECK`) and suppresses reverts.

This discrepancy allows a client marked as a smart contract signer to execute arbitrarily expensive `isValidSignature` logic, causing the caller of `lockRequestWithSignature` to consume excessive gas or revert unexpectedly. While this behavior is compliant with ERC-1271, it introduces a denial-of-service vector in contexts where third parties sponsor the transaction cost.

```
function lockRequestWithSignature(...) {
    address prover = _verifyClientSignatureAndExtractProverAddress(...); // unbounded ERC-1271 call
    ...
}
```

By contrast:

```
function lockRequest(...) {
    _verifyClientSignature(...); // ERC-1271 call capped at ERC1271_MAX_GAS_FOR_CHECK
}
```

This creates two divergent gas safety profiles for signature verification, despite both serving similar purposes.

Impact Clients can run expensive signature logic and force relayers/provers, who sponsor gas, to spend more than expected or trigger out-of-gas reverts. Provers using `lockRequestWithSignature` cannot accurately estimate maximum gas usage, complicating integration with metatransaction or sponsored execution systems.

Recommendation Refactor the code to unify both signature validation paths through `_verifyClientSignature(...)`, which already implements capped gas forwarding for ERC-1271 checks. This avoids code duplication and ensures uniform gas usage and revert behavior regardless of the entry point.

Developer Response The developers followed the recommendation and capped the gas forwarding.

5.1.3 V-BND-VUL-003: Off-chain signature checks do not enforce non-malleability

Severity	Warning	**Commit**	63fc761
Type	Data Validation	**Status**	Fixed
Location(s)	▶ `contracts/src/BoundlessMarket.sol:894`		
Confirmed Fix At	`https://github.com/boundless-xyz/boundless/pull/1161`, 7face92		

The system's ECDSA signature verification process is inconsistently applied across off-chain and on-chain components, introducing a subtle form of malleability-related DoS risk.

Off-chain (Rust), in `assessor/src/lib.rs::Fulfillment::verify_signature`, the following code accepts any ECDSA signature that successfully recovers to the client address:

```
let signature = Signature::try_from(self.signature.as_slice())?;
let recovered = signature.recover_address_from_prehash(&hash)?;
```

However, `Signature::try_from` from the Alloy library accepts high-s signatures (i.e., where `s > n/2`, with `n` the curve order) and internally normalizes them via `normalize_s()`. This allows malleable signatures to pass off-chain checks. In contrast, on-chain (Solidity), both `lockRequest(...) -> _verifyClientSignature(...)` and `priceRequest(...) -> _verifyClientSignature(...)` use OpenZeppelin's `ECDSA.recover`, which enforces signature canonicality by rejecting high-s signatures.

Impact This inconsistency creates a griefing vector: if a prover precomputes a ZK proof using a high-s signature (believing it valid due to off-chain success), and later encounters an on-chain `InvalidSignature` revert during`priceRequest`, the prover's effort is wasted - potentially for all fulfillments close to deadline. The prover can debug the issue, normalize the signature and retry within time-sensitive context. However, this action may be problematic for automated proving service.

Recommendation Add an explicit check immediately after `Signature::try_from(...)` in Rust to reject non-canonical signatures before recovery. Use `if !signature.s_is_canonical()` or similar.

Developer Response The developers implemented checks to ensure the signature is normalized.

5.1.4 V-BND-VUL-004: Unused/duplicate code

Severity	Warning	**Commit**	63fc761
Type	Maintainability	**Status**	Fixed
Location(s)	▶ `contracts/src/types/Predicate.sol` ▶ `crates/boundless-market/src/[...]/mod.rs:444-482,`		
Confirmed Fix At	`https://github.com/boundless-xyz/boundless/pull/1165/`, 3894ec4		

The codebase contains some instances of duplicated or unused functionality that introduce unnecessary maintenance burden and potential for divergence between related methods:

1. `contracts/src/types/Predicate.sol`:
   a) The `eval()` function is unused outside of test code.
2. `crates/boundless-market/src/contracts/mod.rs`:
   a) The function `ProofRequest::validate()` checks internal consistency of a ProofRequest (e.g., ensuring non-empty URLs, correct price ranges, and valid predicates). However, it is not invoked anywhere in the assessor code, only in tests. This performs additional checks on top of the ones performed on-chain when pricing or locking a request.
   b) Both `sign_request()` and `verify_signature()` in ProofRequest recompute the EIP-712 domain and signing hash manually.
   This logic also appears in a new method, `signing_hash()`, which could be invoked directly instead of copying the logic.

```
let domain = eip712_domain(contract_addr, chain_id);
let hash = self.eip712_signing_hash(&domain.alloy_struct());
```

Impact The codebase may become more difficult to maintain, and unused functions may become inconsistent with the rest of the code as future upgrades are performed.

Additionally, `ProofRequests` may not be fully validated in a provable environment.

Recommendation

1. Remove the unused `eval()` function.
2. Refactor `sign_request()` and `verify_signature()` to reuse the existing `signing_hash()` method rather than recomputing the domain and hash inline.
3. `validate()` proof requests within the assessor.

Developer Response The developers implemented recommendations (2.) and (3.), using the existing `signing_hash()` method and `validate()`ing proof requests within the assessor.

Veridise Response Our main concern with `eval()` is that its semantics are now slightly out of date. On-chain, the claim-digest-match requires an image-ID+journal be provided, whereas off-chain it is allowed to provide no image-ID+journal for a `ClaimDigestMatch`.

It would be good to either document the difference or use a different name for the methods

Updated Developer Response This commit addresses the out of date semantics of the eval function by providing options for evaluating with either imageID and journal or with just the claim digest.

5.1.5 V-BND-VUL-005: All-or-Nothing settlement prevents minimum prover payment

Severity	Warning	**Commit**	63fc761
Type	Logic Error	**Status**	Fixed
Location(s)	`contracts/src/BoundlessMarket.sol:551`		
Confirmed Fix At	`https://github.com/boundless-xyz/boundless/pull/1171`, 83f31a6		

In the settlement logic, the function short-circuits when `clientAccount.balance < clientOwes`, returning an `InsufficientBalance` error and skipping all payout logic:

```
if (clientAccount.balance < clientOwes) {
    return abi.encodeWithSelector(InsufficientBalance.selector, client);
}
```

However, this contradicts the intended incentive mechanism: the prover should receive at least the `lockPrice` if the client's balance is sufficient to cover that amount. The current logic enforces an all-or-nothing condition, which prevents partial payment of `lockPrice` when `client.balance < clientOwes` but `client.balance >= lockPrice`.

Impact A client can underfund just below `clientOwes` to block any payout, even if the minimum `lockPrice` is covered.

Recommendation Instead of returning early when`clientAccount.balance < clientOwes`, pay out `lockPrice + clientAccount.balance`.

Developer Response The developers followed the recommendations and implemented partial payment when client funds are insufficient.

5.1.6 V-BND-VUL-006: Missing checks on constructor

Severity	Warning	**Commit**	63fc761
Type	Maintainability	**Status**	Fixed
Location(s)	`contracts/src/BoundlessMarket.sol:111-132`		
Confirmed Fix At	`https://github.com/boundless-xyz/boundless/pull/1166`, 064e9e9		

The contract `constructor()` and `initialize()` functions set several important configurations, but do not perform any basic checks on the values.

Impact Zero-ed out addresses or image IDs may make the contract non-upgradeable or permit incorrect proofs to be submitted.

Recommendation Ensure key addresses and image IDs are non-zero at initialization.

Developer Response The developers followed recommendations and added basic checks in the constructor and initializer.

5.1.7 V-BND-VUL-007: Out-of-date Solidity version

Severity	Info	**Commit**	63fc761
Type	Maintainability	**Status**	Fixed
Location(s)	`contracts/src/` ▶ `HitPoints.sol` ▶ `libraries/` • `BoundlessMarketLib.sol` • `MerkleProofish.sol` ▶ `types/` • `Account.sol` • `AssessorCallback.sol` • `AssessorCommitment.sol` • `AssessorJournal.sol` • `AssessorReceipt.sol` • `Fulfillment.sol` • `Input.sol` • `LockRequest.sol` • `ProofRequest.sol` • `RequestId.sol`		
Confirmed Fix At	`https://github.com/boundless-xyz/boundless/pull/1167`, 37bb9ce		

The project currently specifies Solidity 0.8.20 in some contracts. According to the official Solidity bug tracker, this version has known issues, including potential problems with code generation, optimizer behavior, and language semantics. These bugs can impact contract safety depending on usage of specific features (e.g., inline assembly, calldata tuples, optimizer edge cases).

While the specific code patterns affected vary, relying on a compiler version with unresolved bugs increases the attack surface.

Impact Continuing to use Solidity 0.8.20 may expose contracts to:

- Incorrect execution semantics due to compiler miscompilation.
- Optimizer bugs that alter logic during deployment.
- Subtle issues in calldata handling that attackers could exploit.

Even if current code is not affected by the known issues, future modifications or different compiler flags could unintentionally trigger them.

Recommendation Upgrade all contracts to a more recent Solidity 0.8.x release (preferably the latest stable version) where the known issues have been patched.

- Update pragma solidity directives across the repository.
- Recompile and rerun all test suites under the upgraded version.
- Audit deployment scripts and CI pipelines to ensure the new compiler is consistently enforced.

Developer Response The developers implemented the recommendation, raising the Solidity version to 0.8.26.

Glossary

BitVM A computing paradigm enabling Turing-complete Bitcoin contracts without changing consensus rules. Computations are not executed on-chain but verified, akin to optimistic rollups. A prover claims that a function evaluates to a given output for specific inputs; if false, fraud proofs allow others to penalize the prover. This mechanism allows any computable function to be verified on Bitcoin, serving as a foundation for bridging BTC to second layers such as sidechains, rollups, and zkCoins. 1

BLAKE3 A cryptographic hash function combining an ARX-based compression function with a binary Merkle tree structure. Provides high throughput, SIMD/GPU parallelism, and native extendable-output (XOF). Supports keyed hashing (PRF, MAC, KDF), domain separation via flags, and resists length-extension attacks. To learn more, visit `https://github.com/BLAKE3-team/BLAKE3-specs/blob/master/blake3.pdf` . 1

EIP Ethereum Improvement Proposal. 19

EIP-712 An Ethereum Improvement Proposal (EIP) defining a standard for typed structured data hashing and signing, improving security and usability of off-chain message signatures. See `https://eips.ethereum.org/EIPS/eip-712` for the full EIP. 5

Ethereum Improvement Proposal Peer-reviewed proposals for the Ethereum platform. Visit `https://eips.ethereum.org` to learn more. 19

Groth16 A pairing-based zkSNARK protocol introduced by Jens Groth in 2016. It produces extremely succinct proofs and is widely used in blockchain systems due to its fast verification and small proof size. Groth16 is zero-knowledge, allowing proofs without revealing witnesses, but it requires a relation-specific trusted setup and has relatively slower proving time compared to newer schemes. 1

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. 19

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. 5

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. 19

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, 5