



Auditing Report

Hardening Blockchain Security with Formal Methods

FOR



Boundless Marketplace



Veridise Inc.
June 17, 2026

► **Prepared For:**

Boundless
www.boundless.network

► **Prepared By:**

Aayushman Magar
Evgeniy Shishkin
Mark Anthony

► **Contact Us:**

contact@veridise.com

► **Version History:**

Jul. 7, 2026	V1
Jul. 1, 2026	Initial Draft

Contents

Contents	iii
1 Executive Summary	1
2 Project Dashboard	4
3 Security Assessment Goals and Scope	5
3.1 Security Assessment Methodology	5
3.2 Identified Security Risks	5
3.3 Scope	6
3.4 Classification of Vulnerabilities	6
4 Security Review Assumptions	8
4.1 Operational Assumptions	8
5 Vulnerability Report	9
5.1 Detailed Description of Issues	10
5.1.1 V-BLMP-VUL-001: Fulfillment proofs can be copied and rebound to a different prover	10
5.1.2 V-BLMP-VUL-002: Class-based request selectors can accept attacker- registered fake verifiers	12
5.1.3 V-BLMP-VUL-003: Legacy fulfillment path may allow bypassing proof type restrictions	14
5.1.4 V-BLMP-VUL-004: Unpaid fulfillment can finalize never-locked requests	15
5.1.5 V-BLMP-VUL-005: Unused Code	16
A Appendix	17
A.1 Intended Behavior: Non-Issues of Note	17
A.1.1 V-BLMP-APP-VUL-001: _settleBatch executes callbacks despite non-empty payment errors	17
A.1.2 V-BLMP-APP-VUL-002: Reusable request IDs can cause ambiguous set- tlement against stale lock state	19
A.1.3 V-BLMP-APP-VUL-003: Multiple request IDs can alias to the same lifecycle flags	21

About Veridise

Veridise is an independent security firm founded by academics and security researchers with deep expertise in formal methods, programming languages, and applied cryptography. The firm focuses on high-assurance security for blockchain and cryptographic systems, combining rigorous theory with practical adversarial analysis.

Veridise provides full-stack blockchain security services spanning smart contracts, zero-knowledge circuits and proving systems, consensus and execution clients, cryptographic libraries, and supporting infrastructure. The team routinely analyzes systems at the boundary between protocol design, implementation, and cryptography, where failures are both subtle and high-impact. To date, Veridise has conducted hundreds of security assessments for protocols securing billions of dollars in TVL.

Dynamic Security Veridise treats security as an ongoing engineering discipline rather than a one-time validation exercise. Effective security requires iterative review, continuous feedback, and close collaboration between auditors and system designers. Throughout the engagement, Veridise analysts communicated regularly with the Boundless team using [AuditHub](#), enabling rapid clarification of design intent, prompt discussion of findings, and efficient validation of fixes.

In parallel with manual review, Veridise maintains active research programs focused on automated and formal analysis of blockchain systems. These efforts have produced tools such as *Vanguard* for static analysis and *Picus* for formal verification, which are available through [AuditHub](#). AuditHub integrates these techniques directly into CI/CD workflows, supporting continuous vulnerability detection and regression analysis beyond the scope of a single audit.

Additional details and case studies are available at <https://audithub.dev/case-studies/>.

Veridise Reports Veridise reports are published in the public audit archive linked below. Only reports obtained from this archive or confirmed directly by a representative of [Veridise](#) should be considered official Veridise reports.

<https://veridise.com/audits-archive/>



Between Jun. 17, 2026 and Jun. 24, 2026, Boundless engaged Veridise to conduct a security assessment of Boundless Marketplace - a decentralized proof marketplace. Veridise conducted the assessment over 1 week, with 3 security analysts contributing 3 person-weeks of review effort against commit e2bc0cc. The review strategy combined tool-assisted analysis of the source code with a thorough manual code review by Veridise security analysts. Compared to the previous version, which Veridise audited at commit f0e5fc4, the new version introduces an updated fulfillment flow that separates proof verification from request assessment and routes both through a router with pluggable verifier and assessor adapters.

Project Summary. Boundless Marketplace is a decentralized marketplace for zero-knowledge proofs. It allows requestors to ask for a computation to be proven, and allows provers to generate the corresponding proof in exchange for payment.

In this context, a proof is the cryptographic material submitted by a prover to show that a requested program was executed correctly and produced the claimed output. This lets requestors rely on verified computation without running the computation themselves.

Boundless Marketplace is built around three main conceptual parts.

The first part is the request layer. Requestors define what they want proven by submitting signed proof requests. A request includes the program to run, the input, the requirements for the result, and the payment offered for a valid proof.

The second part is the prover layer. Provers evaluate available requests, run the requested programs, and submit fulfillments containing the proof material and claimed result.

The third part is the marketplace layer. The marketplace connects requestors and provers. It checks submitted fulfillments, records successful results, pays provers, and executes any configured callback.

Compared to the previously reviewed architecture, the redesigned version separates market settlement from fulfillment validation more clearly. The marketplace remains responsible for request binding, state updates, rewards, and callbacks, while proof verification and request assessment are handled through a dedicated validation layer.

This makes the fulfillment flow more modular. Different proof verification and assessment mechanisms can be supported without changing the core marketplace settlement logic. It also gives governance clearer control over which validation paths are available.

The redesign preserves compatibility with the previously deployed marketplace implementation by forwarding unsupported function selectors through a legacy fallback path. As a result, the new architecture introduces a more flexible validation model while maintaining continuity with the earlier system.

Code Assessment. The Boundless Marketplace developers provided the contracts' source code for review. The codebase 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 codebase, the Boundless Marketplace developers explained the main workflows during the kickoff call and provided several accompanying documents.

The source code included a comprehensive test suite that covered many expected user flows and a substantial portion of the protocol's functionality, including the newly introduced features.

Summary of Issues Detected. The security assessment uncovered 5 issues, including 1 high-severity issue. The most severe issue, [V-BLMP-VUL-001](#), allows fulfillment proofs to be copied and rebound to a different prover, potentially enabling a front-runner to steal fulfillment credit or payment. The analysts also identified [V-BLMP-VUL-002](#), where class-based request selectors may accept attacker-registered verifier or assessor implementations under unsafe permissionless class configurations. Both issues were fixed by the developers.

The assessment also identified medium-severity findings related to legacy fulfillment compatibility and unpaid fulfillment. Specifically, [V-BLMP-VUL-003](#) may allow disabled proof types to remain usable through the legacy fulfillment path, while [V-BLMP-VUL-004](#) can finalize never-locked requests before confirming that the prover can be paid. The developers acknowledged both issues, but decided not to fix this behavior.

In addition, the assessment identified 1 informational finding related to general maintainability. Specifically, [V-BLMP-VUL-005](#) covers unused or misleading code constructs, which the developers fixed.

The report also documents several non-issues of note that were determined to be intended behavior. These include [V-BLMP-APP-VUL-001](#), where callbacks may execute despite non-empty payment errors, [V-BLMP-APP-VUL-002](#), where reusable request IDs can interact with stale lock state as part of the protocol's stake rollover design, and [V-BLMP-APP-VUL-003](#), where multiple request variants may intentionally share lifecycle flags for the same client-controlled request slot.

The Boundless Marketplace developers have fixed 3 issues, and have acknowledged 2 issues.

Recommendations. After conducting the assessment of the protocol, the security analysts identified several suggestions for improving Boundless Marketplace.

Document router, adapter, and legacy fallback trust boundaries. The redesigned architecture splits fulfillment validation across the market, router, verifier adapters, and assessor adapters. The security analysts recommend documenting the responsibilities of each component, including which component enforces proof validity, request binding, prover binding, callback-relevant data binding, and routing policy.

The documentation should also explain the security expectations around router configuration and route selection. In particular, it should describe what assumptions a requestor makes when selecting or accepting a class, which choices a prover can make when choosing an entry within that class, and which parts of the route are curated by governance, driven by the market, or selected by the prover. This is especially relevant for permissionless verifier classes, where requestors should be clearly informed that opting in depends on their own due diligence and that interface conformance alone does not imply that a verifier implementation is honest or

appropriate. The temporary legacy fallback path should also be documented, including its role during the broker migration window and when it is expected to be removed.

Document request identity and lifecycle semantics. Some request lifecycle behaviors in the market are intentional but not obvious from the external interface. In particular, the protocol treats a `RequestId` as a client-controlled slot rather than as a globally unique identifier for a single immutable request. As reflected in [V-BLMP-APP-VUL-002](#) and [V-BLMP-APP-VUL-003](#), a client may reuse the same request index for later requests, and multiple request variants that share the same client and index may also share lifecycle flags such as lock or fulfillment status.

The security analysts note that the code includes some comments about reusing the same request ID for different request variants. However, the intended lifecycle behavior is not clearly documented for users and integrators. The documentation should explicitly describe that lifecycle flags are keyed by the client and index rather than by the full encoded `RequestId`, that the smart contract signature flag is not part of the canonical lifecycle key, and that this aliasing behavior is intended rather than an accidental consequence of request ID decoding.

Disclaimer. We hope that this report is informative, but we 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.



Table 2.1: Application Summary.

Name	Version	Type	Platform
Boundless Marketplace	e2bc0cc	Solidity	Ethereum

Table 2.2: Engagement Summary.

Dates	Method	Consultants Engaged	Level of Effort
Jun. 17–Jun. 24, 2026	Manual & Tools	3	3 person-weeks

Table 2.3: Vulnerability Summary.

Name	Number	Acknowledged	Fixed
Critical-Severity Issues	0	0	0
High-Severity Issues	1	1	1
Medium-Severity Issues	3	3	1
Low-Severity Issues	0	0	0
Warning-Severity Issues	0	0	0
Informational-Severity Issues	1	1	1
TOTAL	5	5	3

Table 2.4: Category Breakdown.

Name	Number
Authorization	2
Frontrunning	1
Logic Error	1
Maintainability	1



3.1 Security Assessment Methodology

The security assessment process consists of the following steps:

1. Gather an initial understanding of the protocol's business logic, users, and workflows by reviewing the provided documentation and consulting with the developers.
2. Identify all valuable assets in the protocol.
3. Identify the main workflows for managing these assets.
4. Identify the most significant security risks associated with these assets.
5. Systematically review the codebase for execution paths that could trigger the identified security risks, considering different assumptions.
6. Prioritize one finding over another by assigning a severity level to each.

To improve the depth and efficiency of the review, analysts used the following tools:

- ▶ *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.
- ▶ *LLM-assisted review.* Security analysts performed a comprehensive manual review of all in-scope components, including detailed examination of the source code and system design. Large language models (LLMs) were used as an auxiliary aid to support analyst workflows, including structured reasoning, protocol summarization, trust-boundary mapping, hypothesis generation, proof-of-concept development, and report drafting. All findings, conclusions, and supporting evidence were independently reviewed and validated by analysts against the source code before inclusion in the final report.

3.2 Identified Security Risks

After the initial phase of the security assessment was completed, the security analysts generated a list of potential security risks. They used this list during the code review as a starting point for identifying potential attack vectors. A few of these risks include the following:

- ▶ Can fulfillment proof material be copied, replayed, or rebound so that a different prover receives fulfillment credit or payment?
- ▶ Can router class, entry, or selector configuration allow a prover-selected verifier or assessor implementation to bypass the validation intended by the requestor?
- ▶ Can the legacy fulfillment fallback path remain usable after newer router-based proof type restrictions are expected to disable a proof path?
- ▶ Can settlement, payment, and callback execution diverge, causing callbacks or fulfillment events to occur when the prover is not actually paid?
- ▶ Can reusable or aliased request identifiers cause stale lock, fulfillment, or slashing state to affect a later request?

- ▶ Are there typical Solidity implementation vulnerabilities, such as arithmetic overflows or underflows, reentrancy, unsafe external calls, uninitialized state, or incorrect access control?
- ▶ Can malformed calldata, mismatched array lengths, invalid selectors, or unexpected batch contents cause incorrect routing, denial of service, or inconsistent settlement state?
- ▶ Can user-controlled callback targets, verifier adapters, assessor adapters, or legacy fallback calls introduce unsafe cross-contract interactions?
- ▶ Are administrative or governance-controlled configuration changes adequately constrained so that verifier, assessor, router, and market behavior remains consistent with protocol expectations?
- ▶ Can edge cases around request expiration, locking deadlines, payment balances, and partial settlement produce unfair outcomes for requestors or provers?

3.3 Scope

The scope of this security assessment is limited to the changes made to a specific set of source files from the repository since commit e2bc0cc, as agreed upon with the Boundless Marketplace developers:

- ▶ /contracts/src/BoundlessMarket.sol
- ▶ /contracts/src/IBoundlessMarket.sol
- ▶ /contracts/src/libraries/BoundlessMarketLib.sol
- ▶ /contracts/src/router/BoundlessRouter.sol
- ▶ /contracts/src/router/adapters/OnChainAssessor.sol
- ▶ /contracts/src/router/adapters/R0BoundlessAssessorAdapter.sol
- ▶ /contracts/src/router/adapters/R0BoundlessVerifierAdapter.sol
- ▶ /contracts/src/router/interfaces/IBoundlessAssessor.sol
- ▶ /contracts/src/router/interfaces/IBoundlessJointVerifierAssessor.sol
- ▶ /contracts/src/router/interfaces/IBoundlessRouter.sol
- ▶ /contracts/src/router/interfaces/IBoundlessVerifier.sol
- ▶ /contracts/src/types/Fulfillment.sol
- ▶ /contracts/src/types/FulfillmentBatch.sol
- ▶ /contracts/src/types/Predicate.sol
- ▶ /contracts/src/types/ProofRequestBatch.sol
- ▶ /contracts/src/types/SlimRequest.sol

3.4 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

Security Review Assumptions

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 Marketplace.

- ▶ Governance is assumed to correctly configure and maintain the router classes, entries, tombstones, default route, adapter addresses, and legacy fallback configuration. In particular, governance is trusted not to intentionally register malicious verifier or assessor implementations, or to configure classes in a way that contradicts the intended validation policy.
- ▶ Verifier and assessor implementations selected through `BoundlessRouter` are assumed to honestly enforce proof validity, request binding, prover binding, and assessment semantics expected for their route.
- ▶ External verification dependencies used by the adapters, including the pinned RISC Zero verifier and signature validation primitives, are assumed to behave according to their specifications.
- ▶ Requestors and integrators are assumed to understand the difference between selecting an exact router entry and selecting a router class, including the additional trust and configuration assumptions introduced by class-based selection.
- ▶ Callback consumers are assumed to treat callbacks as permissionless and potentially repeatable, and to make callback handlers idempotent where repeated execution would otherwise create unsafe side effects.
- ▶ The legacy implementation reached through the market fallback path is assumed to behave correctly for compatibility flows that remain intentionally supported during the migration period.
- ▶ All relevant Groth16 trusted setup ceremonies were performed correctly, including secure generation of proving and verification parameters and proper disposal of any toxic waste. The review did not independently validate the setup ceremonies, parameter generation process, or operational handling of setup secrets.

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-BLMP-VUL-001	Fulfillment proofs can be copied and . . .	High	Fixed
V-BLMP-VUL-002	Class-based request selectors can accept . . .	Medium	Fixed
V-BLMP-VUL-003	Legacy fulfillment path may allow . . .	Medium	Acknowledged
V-BLMP-VUL-004	Unpaid fulfillment can finalize never- . . .	Medium	Acknowledged
V-BLMP-VUL-005	Unused Code	Info	Fixed

5.1 Detailed Description of Issues

5.1.1 V-BLMP-VUL-001: Fulfillment proofs can be copied and rebound to a different prover

Severity	High	Commit	e2bc0cc
Type	Frontrunning	Status	Fixed
Location(s)	contracts/src/router/adapters/ ▶ OnChainAssessor.sol:97-136 ▶ R0BoundlessAssessorAdapter.sol:104-163		
Confirmed Fix At	boundless/pull/2052		

Description OnChainAssessor is responsible for validating fulfillment batches before they are accepted by the market. For each fulfillment, it checks that the submitted proof satisfies the request predicate and that any attached (imageId, journal) reconstructs to the supplied claimDigest. After collecting the per-fill claimDigest values, it verifies a batch level assessorSeal, which is an ECDSA signature by batch.prover over (prover, requestDigests[], claimDigests[]). This means the batch signature binds the selected prover to the list of request and claim digests. However, the underlying proof material for each fill is only checked against the claimDigest, and it is not itself bound to batch.prover.

As a result, a front-runner who observes a pending fulfillment can copy the original fulfillment data, including the proof material, claimDigest, and request data, and then submit a new batch with their own address as batch.prover. The attacker can also generate a valid assessorSeal for their own prover address over the same request and claim digests. Since the copied proof still satisfies the same claim digest, and the new signature correctly matches the attacker controlled batch.prover, assessor verification will pass.

This is visible in verifyAssessor(), where each fill is validated against its predicate and claimDigest, while the collected claim digests are later passed into _verifyProverSignature() together with batch.prover. The prover signature covers (prover, requestDigests, claimDigests), but the app proof itself is not required to commit to the prover.

```

1 function verifyAssessor(FulfillmentBatch calldata batch, bytes32[] calldata
  requestDigests) external view
2 {
3     uint256 n = batch.requests.length;
4     if (batch.fills.length != n || requestDigests.length != n) revert LengthMismatch()
5     ;
6     // Per-fill: predicate satisfaction + claim-digest binding. Collect
7     // claimDigests for the per-batch signature hash.
8     bytes32[] memory claimDigests = new bytes32[](n);
9
10    // veridise: elided //
11
12    // Per batch: prover signature over (prover, requestDigests, claimDigests).
13    _verifyProverSignature(batch.prover, requestDigests, claimDigests, batch.
      assessorSeal);
14 }

```

Snippet 5.1: Snippet from `verifyAssessor()` in `contracts/src/router/adapters/OnChainAssessor.sol`

This allows proof material generated by one prover to be rebound to another prover, as long as the attacker can produce the required assessor authorization in time.

A related concern also exists for guest-based assessor flows, although the attack is less direct. Because the guest-based assessor commits to `batch.prover`, the attacker cannot reuse the victim's assessor proof byte-for-byte. Instead, they would need to observe the victim's pending fulfillment, reuse the underlying fulfillment material, and generate a fresh assessor proof for their own `batch.prover` quickly enough to front-run the original submission. A well-resourced attacker, particularly on chains with longer block times, may still be able to complete this process in time.

Impact A front-runner can potentially steal fulfillment credit or payment by copying another prover's fulfillment proof and rebinding it to themselves as `batch.prover`.

For never-locked requests, this may allow direct theft of the fulfillment payment. After the lock deadline, it may also allow theft of late fulfillment or collateral rewards. Before the lock deadline, payment may still be protected by the existing lock prover check, but copied submissions can still cause delivery or callback side effects if those paths are not idempotent.

Recommendation Bind the fulfillment proof to the prover that is meant to receive credit for the fulfillment. The claim, journal, app proof, or assessor-verified statement should include `batch.prover` as part of the committed data, so copied proof material cannot be rebound to a different prover.

Also ensure the same prover binding is enforced consistently across both on-chain and guest-based assessor adapters.

Developers Response The developers addressed this issue by adding a commit-reveal requirement, requiring provers to commit to the exact fulfillment batch in a prior block before revealing the proof, thereby preventing copied fulfillment proofs from being rebound to a different prover during settlement.

5.1.2 V-BLMP-VUL-002: Class-based request selectors can accept attacker-registered fake verifiers

Severity	Medium	Commit	e2bc0cc
Type	Authorization	Status	Fixed
Location(s)	contracts/src/router/BoundlessRouter.sol:549		
Confirmed Fix At	boundless/pull/2054		

Description The BoundlessRouter contract allows a request selector to represent an exact verifier entry, a verifier class, or the chain-default class. In the `_matchSignedSelector()` function, a prover-selected verifier entry is accepted when `signedSel == sealClassId`, meaning a request that signs a verifier class can be fulfilled using any entry registered under that class.

For verifier classes, the assessor is not selected by the requestor. Instead, the verifier class defines a `requiredAssessorClass`, and `verifyBatch()` only checks that the assessor selector supplied in `assessorSeal` belongs to that required class:

```

1 bytes4 required = classes[verifierClassId].requiredAssessorClass;
2 if (asEntry.classId != required) {
3     revert AssessorClassMismatch(required, asEntry.classId);
4 }

```

Entries are added through `instantiate()`. When the parent class has `permissionlessInstantiate = true`, non-admin users can register new entries under that class. The registration check only verifies ERC-165 interface support for the expected interface; it does not verify that the implementation performs honest proof or assessor validation.

This creates two related risks. First, if a requestor signs a permissionless verifier class, a malicious prover can register a dummy verifier under that class, select it through `seal`, and have it accept a bogus proof. Second, if governance creates a permissionless assessor class and a verifier class uses it as `requiredAssessorClass`, then requestors targeting that verifier class cannot pin a specific assessor. A malicious prover can register a dummy assessor under the required assessor class, select it through `assessorSeal`, and bypass the intended assessor validation.

In both cases, the requestor cannot reliably evaluate the safety of a permissionless class at signing time, because new verifier or assessor entries can be added shortly before fulfillment.

Impact A malicious prover may be able to make `ROUTER.verifyBatch()` succeed and receive payment without producing a valid proof, depending on how verifier and assessor classes are configured. The likelihood is lower because this requires either the requestor to select a permissionless verifier class, or governance to configure a verifier class with a permissionless assessor class. Exact verifier selector pinning mitigates the first case, but does not help when the selected verifier class depends on a permissionless assessor class.

Recommendation Any verifier or assessor implementation used in settlement should require governance/admin approval before it can be selected during fulfillment. In particular, verifier classes should not depend on permissionless assessor classes, since requestors cannot choose a specific assessor entry. ERC-165 interface support should not be treated as sufficient evidence that an implementation performs correct verification.

Developers Response The developers addressed the issue by disallowing permissionless assessor classes. They also noted that the permissionless verifier path is opt-in, and requesters are expected to perform due diligence.

5.1.3 V-BLMP-VUL-003: Legacy fulfillment path may allow bypassing proof type restrictions

Severity	Medium	Commit	e2bc0cc
Type	Authorization	Status	Acknowledged
Location(s)	contracts/src/BoundlessMarket.sol		
Confirmed Fix At	N/A		

Description BoundlessMarket delegates selectors that are not implemented on the current market ABI to LEGACY_IMPL through its fallback path. This preserves backward compatibility, but it also leaves an alternate fulfillment path that may not enforce the proof type restrictions introduced by the current router. See snippet below for reference.

In the new fulfillment flow, submissions are routed through the current router, which checks whether the proof type and verifier path are still allowed. This lets governance disable proof types through router registry, class, or tombstone configuration, after which those proof types should no longer be accepted through the new path.

However, legacy only fulfillment selectors can still be routed to the legacy implementation. If the legacy verifier and assessor path still accepts a disabled proof type, a prover can submit fulfillment through the fallback path instead of the current router path. In that case, the fulfillment is checked against legacy verification rules rather than the current router policy. As a result, a proof type that is correctly rejected through the new fulfillment path may still be accepted through the legacy path, provided the legacy verifier and assessor accept it.

Impact A prover may be able to fulfill requests using a proof type that governance has disabled in the new router, by routing the submission through the legacy fulfillment path. This bypasses the intended verification policy and can cause the market to pay for proofs that should no longer be accepted under the current router configuration.

Recommendation Ensure that governance also disables the corresponding legacy verifier and assessor paths whenever a proof type is disabled in the current router.

Developers Response The developers have acknowledged the issue but will not provide a fix, stating:

The legacy fallback is a temporary backward-compatibility shim for the broker migration window; once in-flight legacy traffic has settled, the proxy will be upgraded to remove it, closing the alternate path entirely. The legacy implementation is immutable, so until then we adopt the recommendation operationally: governance disables the corresponding legacy verifier/assessor entry whenever a proof type is disabled in the new router, keeping the two policies in sync (added to the governance runbook). At launch the router's accepted proof types mirror the legacy verifier set, so there is no divergence to exploit; a gap could only arise from a later governance action, which the runbook covers. No contract change.

5.1.4 V-BLMP-VUL-004: Unpaid fulfillment can finalize never-locked requests

Severity	Medium	Commit	e2bc0cc
Type	Logic Error	Status	Acknowledged
Location(s)	contracts/src/BoundlessMarket.sol:609-610		
Confirmed Fix At	N/A		

Description In the never-locked fulfillment path, `_fulfillAndPayNeverLocked()` marks the request as fulfilled and emits `RequestFulfilled` before checking whether the requestor has enough balance to pay the prover.

Code path:

```
1 clientAccount.setRequestFulfilled(idx);
2 emit RequestFulfilled(id, assessorProver, requestDigest);
3 if (clientAccount.balance < price) {
4     return abi.encodeWithSelector(InsufficientBalance.selector, client);
5 }
```

Because this path returns a `paymentError` instead of reverting, the fulfillment transaction can still complete with a valid proof delivered, `ProofDelivered` emitted, and callbacks executed, while the prover receives no payment. A malicious requestor can exploit this by withdrawing their deposited balance before the prover's `priceAndFulfill()` transaction is included, causing the payment check to fail after the proof has already been accepted.

Impact A prover can lose the value of their work: they submit a valid proof, the request becomes fulfilled, but no payment is transferred. The requestor can grief or avoid payment by front-running fulfillment with `withdraw()`.

Recommendation Move the balance check before setting the fulfilled flag and emitting fulfillment events, and revert or return before accepting the proof if the requestor cannot pay.

Preferably, make insufficient balance in the never-locked path revert the fulfillment for that request, so the request is not finalized unpaid. Alternatively, require requestors to escrow the required payment before never-locked fulfillment can be accepted.

Developers Response The developers have acknowledged the issue but will not provide a fix, stating:

This is a known behavior and was responded to in a previous audit: V-RISC0-VUL-006 in [https://github.com/boundless-xyz/boundless-security/blob/main/audits/2025_04_Veridise%20\(Boundless%20Core\).pdf](https://github.com/boundless-xyz/boundless-security/blob/main/audits/2025_04_Veridise%20(Boundless%20Core).pdf)

5.1.5 V-BLMP-VUL-005: Unused Code

Severity	Info	Commit	e2bc0cc
Type	Maintainability	Status	Fixed
Location(s)	contracts/src/ ▶ <code>BoundlessMarket.sol:43</code> ▶ <code>IBoundlessMarket.sol:91-9191</code>		
Confirmed Fix At	boundless/pull/2057		

Description The analysts identified some code constructs that are currently unused and should be removed or updated to prevent inconsistencies as the codebase evolves.

The following constructs are currently unused:

- ▶ Error `MismatchedRequestId` in `contracts/src/BoundlessMarket.sol` is never used.
- ▶ The custom `Upgraded` event declared in `contracts/src/IBoundlessMarket.sol` is never emitted by the implementation. Instead, upgrades emit OpenZeppelin's standard UUPS/ERC1967 upgrade event, making the interface event misleading for integrators and monitoring tools.

Impact Unused or obsolete code can become outdated over time, increasing maintenance effort and making the codebase more difficult to understand.

Additionally, the unused `Upgraded` event creates a misleading public interface, which may cause monitoring tools or integrations to rely on an event that is never emitted.

Recommendation Remove unused code constructs that are no longer required. Where a construct is intended to remain part of the public interface, ensure it accurately reflects the implementation or update the implementation accordingly.

Developer Response The developers addressed the issue by removing the mentioned unused constructs.



A.1 Intended Behavior: Non-Issues of Note

A.1.1 V-BLMP-APP-VUL-001: `_settleBatch` executes callbacks despite non-empty payment errors

Severity	Medium	Commit	e2bc0cc
Type	Logic Error	Status	Intended Behavior
Location(s)	contracts/src/BoundlessMarket.sol:347		

Description The `_settleBatch()` function calls `_fulfillAndPay()` and stores the returned error data in `paymentError[outIdx]`. However, when deciding whether to execute a callback, `_settleBatch()` only checks whether the request is not expired and whether a callback address exists:

```
1 (paymentError[outIdx], expired) = _fulfillAndPay(...);
2
3 if (!expired && slim.callback.addr != address(0)) {
4     _executeCallback(...);
5 }
```

This means callbacks can still be executed even when the `_fulfillAndPay()` function returned a non-empty payment error, such as `RequestIsFulfilled`, `RequestIsLocked`, `InsufficientBalance`, or `PartialPayment`. These errors do not necessarily revert the transaction, so settlement can fail or be skipped while callback execution still proceeds.

Impact Callbacks may execute even when the request was already fulfilled, the submitting prover was not eligible to be paid, or the client did not have enough balance. Integrations that treat callbacks as proof of successful one-time fulfillment and payment settlement may perform duplicate or invalid external side effects.

Recommendation Only execute callbacks after successful settlement. At minimum, the `_settleBatch()` function should check that `paymentError[outIdx].length == 0` before calling `_executeCallback()`. Consider also adding explicit replay protection to ensure a callback cannot be executed more than once for the same fulfilled request.

Developers Response The developers have acknowledged the issue but will not provide a fix, stating:

This is intended, documented behavior. Boundless callbacks are at-least-once, permissionless, and payment-independent by design. Per the callbacks docs (<https://docs.boundless.network/developers/tutorials/callbacks>): "any user can submit a request that causes any contract's callback to be executed, potentially

re-using proofs that have been previously submitted," callbacks must be "robust to multiple invocations," and "successful execution of the callback is not required for a request to be marked as fulfilled." The ProofDelivered event NatSpec likewise states it may be logged multiple times per request. Crucially, ROUTER.verifyBatch runs (and reverts on failure) before _settleBatch, so the callback always receives a genuinely verified proof - a non-empty paymentError is an accounting outcome, not a proof-validity one.

We therefore won't implement the recommended changes:

- ▶ Gating on paymentError.length == 0 would suppress the callback on legitimate first deliveries where payment failed for client-side reasons (e.g. an under-funded never-locked request - itself intended behavior), and still would not be "exactly once."
- ▶ Contract-level replay protection contradicts the intentional at-least-once / permissionless-resubmission model that callback consumers already rely on; idempotency is the consumer's responsibility (the Counter example demonstrates it with an AlreadyVerified guard).
- ▶ Integrations must not treat a callback as proof of one-time, paid settlement - the docs are explicit on this.

A.1.2 V-BLMP-APP-VUL-002: Reusable request IDs can cause ambiguous settlement against stale lock state

Severity	Low	Commit	e2bc0cc
Type	Data Validation	Status	Intended Behavior
Location(s)	contracts/src/BoundlessMarket.sol		

Description Requests are keyed by `RequestId`, but the protocol allows a requestor to reuse the same `RequestId` for a later request with different parameters. This creates ambiguity because lock state is also tracked by the same identifier and may remain associated with an earlier request.

Consider a request `R1` with `RequestId = I`. An honest prover locks `R1`, causing collateral and lock metadata to be stored under `requestLocks[I]`. If the prover misses the lock deadline, the requestor can later create a new request `R2` using the same `RequestId = I`, but with different request parameters.

When `R2` is fulfilled, the settlement path may still observe the stale lock state from `R1`. If `R2` is priced and fulfilled in the same transaction, `_fulfillAndPay()` can accept the current pricing context without requiring the fulfilled request digest to match the digest stored in the old lock. Because the old lock deadline has passed, execution can enter the "was locked" fulfillment path and treat the late fulfiller of `R2` as the prover eligible for settlement effects tied to the old lock.

This creates a mismatch between the request being fulfilled and the lock state being used during settlement. In the worst case, a malicious requestor can create a difficult or infeasible request `R1` with attractive terms, wait for a prover to lock and miss the deadline, then reuse the same `RequestId` for an easier request `R2`. By fulfilling `R2`, the requestor or an address they control may become the late fulfiller associated with the stale lock and later receive the unburned portion of the original prover's slashed collateral.

More generally, reusing a `RequestId` causes stale state to bleed across logically separate requests. The reused request may inherit lock flags, collateral, price, or settlement behavior from an earlier request even though the request parameters have changed. This makes the lifecycle of a request ID ambiguous and allows later requests to be evaluated against stale state from prior requests.

Impact Requestors can reuse a `RequestId` to create ambiguous settlement behavior against stale lock state. This can allow a later, easier request to interact with collateral or lock metadata from an earlier request, potentially redirecting part of a missed lock slash to the requestor or an address they control.

Even when no direct loss is triggered, request ID reuse creates confusing and unsafe lifecycle semantics where different requests can share lock and settlement state.

Recommendation Enforce uniqueness of `RequestId` across the full request lifecycle, or require any reused request ID to have all prior lock, pricing, and settlement state fully cleared before it can be accepted again.

Additionally, fulfillment of a locked request should require the fulfilled request digest to match the digest stored in the lock, so stale lock state cannot be applied to a different request.

Developer Response The developers have acknowledged the issue but will not provide a fix, stating:

This is intended, pre-existing behavior. `RequestId = client + 32-bit index` is a client-controlled slot, and the market deliberately supports stake rollover on index reuse: if a locked request's lock expires unfulfilled, a later request reusing the same index can be fulfilled, and that fulfiller inherits the expired lock's slash. This is a deliberate feature with dedicated tests

(`testFulfillWasLockedRequestRepeatIndexStakeRollover*`, whose comments state "slash should award stake to the fulfiller of the second request"), and it is byte-identical to the deployed, previously-audited market.

Note the before-lock-deadline path already binds to the lock - only the locked request and its locker can be paid (`lock.requestDigest / lock.prover` are checked). The open rollover semantics apply only after the lock deadline, by design.

The described worst case requires a prover to lock a request it cannot fulfill - locking is the prover's risk gate: collateral is committed against the specific request the prover chose to lock and must evaluate beforehand. We therefore won't implement both recommendations: enforcing `RequestId` uniqueness / state-clearing removes legitimate index reuse, and requiring the fulfilled digest to match the lock on the post-deadline path removes the stake-rollover feature.

A.1.3 V-BLMP-APP-VUL-003: Multiple request IDs can alias to the same lifecycle flags

Severity	Low	Commit	e2bc0cc
Type	Logic Error	Status	Intended Behavior
Location(s)	contracts/src/types/RequestId.sol		

Description Requests progress through a lifecycle of being locked, fulfilled, and potentially slashed. To track this lifecycle, the market stores per-request status flags indicating whether a request has been locked or fulfilled.

The protocol treats `RequestId` as the request identifier throughout much of the request lifecycle. A `RequestId` encodes the client address, request index, and the smart contract signature flag, as shown in the snippet below.

```

1  /// @notice Extracts the client address and index from a RequestId.
2  /// @param id The RequestId to extract from.
3  /// @return The client address and the 32-bit index, and true if the signature is a
4  ///         smart contract >signature.
5  function clientIndexAndSignatureType(RequestId id) internal pure returns (address,
6  ///         uint32, bool) {
7  ///     uint256 unwrapped = RequestId.unwrap(id);
8  ///     if (unwrapped & (type(uint256).max << 193) != 0) {
9  ///         revert IBoundlessMarket.InvalidRequest();
10  ///     }
11  ///     return (address(uint160(unwrapped >> 32)), uint32(unwrapped), (unwrapped & >
12  ///         SMART_CONTRACT_SIGNATURE_FLAG) != 0);
13  }

```

Snippet A.1: Snippet from `clientIndexAndSignatureType()` in `contracts/src/types/RequestId.sol`

However, lifecycle state is not keyed by the full `RequestId`. Instead, the signature flag is discarded and only the `(client, idx)` pair is used when retrieving the request's lifecycle flags through `clientAndIndex()`.

```

1  /// @notice Extracts the client address and index from a RequestId.
2  /// @param id The RequestId to extract from.
3  /// @return The client address and the 32-bit index.
4  function clientAndIndex(RequestId id) internal pure returns (address, uint32) {
5  ///     uint256 unwrapped = RequestId.unwrap(id);
6  ///     if (unwrapped & (type(uint256).max << 193) != 0) {
7  ///         revert IBoundlessMarket.InvalidRequest();
8  ///     }
9  ///     return (address(uint160(unwrapped >> 32)), uint32(unwrapped));
10  }

```

Snippet A.2: Snippet from `clientAndIndex()` in `contracts/src/types/RequestId.sol`

For example, during fulfillment the protocol derives the `(client, idx)` pair from the `RequestId` and immediately uses it to query the request's lock and fulfillment status.

```

1  /// Complete the fulfillment logic after having verified the app and assessor
2  /// receipts. 'requestDigest' is the verified EIP-712 digest reconstructed
3  /// from the slim payload; the caller has already asserted it matches the
4  /// stored binding via '_verifyBinding'. 'id' comes from the trusted slim
5  /// payload (positionally paired with 'fill').
6  function _fulfillAndPay(Fulfillment calldata fill, RequestId id, bytes32
7      requestDigest, address prover)
8      internal
9      returns (bytes memory paymentError, bool expired)
10 {
11     (address client, uint32 idx) = id.clientAndIndex();
12     Account storage clientAccount = accounts[client];
13     (bool locked, bool fulfilled) = clientAccount.requestFlags(idx);
14     // veridise: elided //

```

Snippet A.3: Snippet from `_fulfillAndPay()` in `contracts/src/BoundlessMarket.sol`

Because the smart contract signature flag is not considered when accessing the lifecycle flags, two distinct `RequestId` values that differ only by that flag resolve to the same lock and fulfillment state. This creates a lifecycle aliasing issue. If one request is locked or fulfilled, another request with the same `(client, idx)` but a different signature flag will also appear locked or fulfilled, even though it is a distinct request. This can prevent the second request from progressing through its lifecycle independently.

The aliasing can also affect slashing behavior. If request **A** is locked, request **B** with an aliased lifecycle entry may also appear locked. Since **B** does not have a corresponding `RequestLock`, `requestLocks[B]` resolves to its default value, causing `lock.deadline()` to return 0. As a result, `slash(B)` satisfies the lock-expiry check immediately. Although no collateral is transferred because no corresponding `RequestLock` exists, the request can still be incorrectly marked as slashed and emit an erroneous slashing event.

Impact Distinct requests can share lock, fulfillment, or slashing status because their lifecycle flags alias to the same `(client, idx)` entry.

This can cause requests to become incorrectly locked, fulfilled, or slashed, emit misleading events, and prevent valid requests from progressing through their expected lifecycle. While this may not directly lead to collateral loss in the default-lock case, it creates ambiguous and unsafe request state.

Recommendation Key request lifecycle flags by the full `RequestId`, including the smart contract signature flag, or otherwise enforce that only one canonical `RequestId` can exist for each `(client, idx)` pair.

All lifecycle state, including lock, fulfillment, and slashing status, should use the same request identity to avoid aliasing between distinct requests.

Developer Response The developers have acknowledged the issue but will not provide a fix, stating:

Intended, pre-existing behavior. A RequestId is a client-controlled slot (client + index), and the lifecycle flags are deliberately keyed by that id rather than the full request digest. This lets a client publish several request variants under one id (same id, different pricing terms) with the guarantee that at most one of them is ever locked or fulfilled. The shared flags are what enforce that mutual exclusion: once one variant is locked, no other can be, and once one is fulfilled the rest are closed out.

The aliasing applies only to the flags, not to payment: the lock stores the specific variant's request digest and binds the rest of the lifecycle to it, so there is no ambiguity over which request was settled or who is paid. This is the same client-controlled-RequestId design as 1260 (and is byte-identical to the previously-audited legacy market). We therefore decline to enforce id uniqueness - it would remove the intended multi-variant capability without adding any safety the digest binding doesn't already provide.