OpenZeppelin | security

OIF Broadcaster Audit

OPEN INTENTS FRAMEWORK

November 26, 2025

Table of Contents

Summary

Type	Library
Timeline	From 2025-10-27 To 2025-10-31
Languages	Solidity

Total Issues	18 (12 resolved, 1 partially resolved)
Critical Severity Issues	0 (0 resolved)
High Severity Issues	1 (1 resolved)
Medium Severity Issues	1 (0 resolved)
Low Severity Issues	5 (2 resolved)
Notes & Additional Information	11 (9 resolved, 1 partially resolved)

Scope

OpenZeppelin audited 3 different scopes.

The first one was the openintentsframework/broadcaster repository at commit 3522b4c.

In scope were the following files:

```
contracts
├── interfaces
│   ├── IBlockHashProver.sol
│   ├── IBlockHashProverPointer.sol
│   ├── IBroadcaster.sol
│   └── IReceiver.sol
├── libraries
│   └── ProverUtils.sol
├── BlockHashProverPointer.sol
├── Broadcaster.sol
└── Receiver.sol
```

The second one was the OpenZeppelin/openzeppelin-contracts repository at commit d9f966f.

In scope were the following files:

```
contracts
└── utils
    └── RLP.sol
```

The third one was the openintentsframework/oif-contracts repository at commit acc7f9c.

In scope were the following files:

```
src
└── integrations
    └── oracles
        └── broadcaster
            └── BroadcasterOracle.sol
```

System Overview

The present audit encompasses three distinct scopes focusing on foundational components for cross-chain interoperability and data validation within the Open Intents Framework (OIF) ecosystem. Together, these scopes aim to provide reliable message verification, standardized data encoding, and secure broadcasting mechanisms across heterogeneous blockchain environments.

ERC-7888 Implementation

The first scope centers on the ERC-7888 standard, which defines a generalized framework for cross-chain message verification. This implementation introduces three core contracts `Broadcaster`, `Receiver`, and `BlockHashProverPointer` alongside a supporting library, `ProverUtils`.

`Broadcaster` contracts are responsible for emitting verifiable messages on the source chain, anchoring communication between related blockchain networks. On the other hand, `Receiver` contracts facilitate message ingestion on the destination chain, ensuring that only verified and finalized data originating from trusted sources are processed.

`BlockHashProverPointer` provides a flexible referencing mechanism that links to specific `BlockHashProver` implementations. These provers serve as the cryptographic bridge between chains by verifying account data and storage slots within state roots and storage Merkle Patricia tries.

By modularizing verification logic, ERC-7888 enables adaptable cross-chain communication that can evolve alongside chain upgrades or alternative verification mechanisms while maintaining strong guarantees of authenticity and consistency.

RLP Library Implementation

The second scope involves the development of a dedicated RLP (Recursive Length Prefix) library to handle data serialization and deserialization in accordance with Ethereum's canonical encoding format. The library provides efficient methods for encoding structured data into RLP format and decoding RLP-encoded payloads back into their constituent elements. Correct RLP implementation is critical for interoperability, as it ensures deterministic data interpretation across systems and contracts relying on Ethereum-compatible encoding.

`BroadcasterOracle` for the OIF Protocol

The third scope focuses on the implementation of a `BroadcasterOracle` contract, designed for the Open Intents Framework (OIF), a modular, intent-based cross-chain protocol. The OIF enables users to define and execute complex cross-chain intents, supporting customizable asset delivery and validation conditions that can be fulfilled permissionlessly by open solvers.

Operating as a component of OIF's smart contract layer, the `BroadcasterOracle` contract establishes reliable communication between broadcasted messages and on-chain verifiers. It aligns with OIF's output-input separation model, allowing independent asset collection and delivery flows, such as Output First and Input Second, via resource locks or escrow mechanisms. Through this architecture, `BroadcasterOracle` contributes to a permissionless, extensible settlement infrastructure capable of supporting hybrid and cross-chain financial workflows.

Security Model and Trust Assumptions

Each scope introduces unique trust assumptions and operational constraints that collectively define the system's security model.

ERC-7888

- **Pointer Ownership and Upgrades**: The `BlockHashProverPointer` contract relies on its owner to correctly update references to valid `BlockHashProver` implementations. A malicious or negligent owner could either DoS the system or facilitate forged messages by redirecting the pointer to a fraudulent prover.

- **Chain Consistency**: When updating to a new `BlockHashProver`, the home and target chain must remain identical to the previous configuration. This is a property that cannot be programmatically verified.

- **Chain Upgrades**: Protocol security depends on stable chain storage structures. If a chain upgrade modifies where block hashes are stored (e.g., repurposing mappings on a parent chain), older `BlockHashProvers` might yield invalid or stale block hashes, potentially allowing receivers to ingest forged data.
- **Message Guarantees**: The ERC ensures that messages can be read (given finalization), but not that they will be read. Since finalization occurs sequentially across chains, message availability depends on cumulative finalization time along the route.

RLP Library

The main risk associated with the RLP library concerns boolean decoding semantics. Decoding a boolean as an integer introduces a potential mismatch with single-byte encoding expectations. This can lead to inconsistent interpretations in downstream logic where a boolean value's binary length carries semantic importance.

`BroadcasterOracle` and Route Constraints

- In the `BroadcasterOracle` implementation, the owner holds the ability to set the broadcaster ID across destination chains. Once a route is constrained, it cannot be updated. Consequently, if a chain later changes its settlement layer and requires a different route to reach the broadcaster, the route becomes irreversibly bricked, preventing further message propagation and effectively locking communication for that chain.
- Applications built on top are assumed to implement the corresponding checks to prevent double spending, multiple cross-chain verification, etc.

Privileged Roles

Throughout the system, the following privileged roles have been identified:

- **`BlockHashProverPointer` Owner**: Maintains administrative control over prover references. Responsible for ensuring that updates to the pointer reference valid and compatible `BlockHashProver` implementations. Failure to manage this correctly can result in message forgery or DoS conditions.

- `BroadcasterOracle` **Owner**: Holds the authority to configure broadcaster IDs and define message routes across destination chains. This role must be exercised with caution, as constrained routes are immutable, and improper configuration can permanently disrupt inter-chain connectivity.
- **Receiver Callers**: Although not privileged in the administrative sense, `Receiver` callers bear the responsibility of selectively reading valid messages, as ERC-7888 does not enforce message liveness or delivery guarantees.

Together, these roles and assumptions define the operational security model for the audited components, emphasizing cautious upgrade practices, responsible ownership, and alignment between protocol-level guarantees and system-level integrity.

High Severity

H-01 Prover Copies Cannot Be Updated

The `updateBlockHashProverCopy` function allows for updating the address of a copy of a remote chain prover to a new version within the local chain. Before updating the implementation, this function ensures that the version of the prover at the new address is greater than the version of the prover at the old address.

However, an issue arises because the `_blockHashProverCopies` mapping is initialized to the zero address. As a result, any attempt to update a prover copy reverts when calling the `version` getter on the zero address, causing the update to be blocked. This limitation prevents the receiver contract from correctly verifying messages from chains that involve multiple routes.

Consider performing the version check only when an implementation address for a copy has been set.

Update: *Resolved at pull request #29 at commit b66e918.*

Medium Severity

M-01 Potential for Arbitrary Application in Message Verification

When a proof of filled payloads is submitted, the `source` refers to the address of the application that has attested to the data. However, the broadcast message lacks any information about the application. Consequently, when a user verifies the message on another chain, they may provide an arbitrary application within `messageData`. Since the message does not contain any information to identify the application, this arbitrary value is used directly in the `_attestations` mapping without validation.

Consider adding the `application` information into the message hash so that it can be validated during message verification.

Update: *Acknowledged, will resolve. Drafted fix in pull request #160 at commit 1872a01.*

Low Severity

L-01 Missing Version Validation

When `BlockHashProverPointer` sets the implementation address for the first time, it does not perform any validation at all. However, all subsequent implementation changes validate that the new `version` keeps increasing compared to the old one. If the initial implementation does not support the `version` function, the pointer will not be able to set a new address again. This is because the check for the increasing version will fail when it attempts to call the `version` method on the old implementation.

Consider checking that the initial implementation supports the `version` method.

Update: *Resolved at pull request #38 at commit 807810f.*

L-02 Lack of Validation for Payload Length

Currently, there is no limit on the number of payloads that can be submitted to `BroadcasterOracle`. However, during the message verification process, the system extracts the length of the array of payloads using only 2 bytes of data. As a result, if the number of payloads submitted to the oracle exceeds the limit that can be represented by 2 bytes, the message will not be verifiable on the destination chain.

Consider limiting the amount of payloads allowed on the `submit` function.

Update: *Resolved at pull request #159 at commit fb9575a.*

L-03 RLP Address Encoding Allows Leading Zero Bytes

The `RLP` library currently encodes an `address` as a 20-byte array. This representation can contain leading zero bytes.

This is not necessarily a problem in itself. However, the Ethereum Yellow Paper states the following:

> When interpreting RLP data, if an expected fragment is decoded as a scalar and leading zeroes are found in the byte sequence, clients are required to consider it non-canonical and treat it in the same manner as otherwise invalid RLP data, dismissing it completely.

This ambiguity could cause implementations that treat the `address` as a scalar value to fail when decoding RLP data containing an `address` with a leading zero.

To obtain better compatibility and alignment with the specification, consider treating the `address` as a scalar value and encoding it using its `uint256` representation. In this case, any leading zeroes will not be included in the encoded byte array.

Update: *Acknowledged, not resolved. The team stated:*

> After some reviewing, the conclusion is that encoding without the leading zeros would not be consistent with the current ethereum ecosystem. If someone wants to encode an Address without the leading zeros, they can manually do the casting to uint256 and then call the corresponding encode function. However this should not be the default encoding.

L-04 RLP Address Decoding Allows Only Fixed Address Lengths

The `RLP` library's `address` decoding function currently only allows encoded addresses with lengths of 1 byte (for `address(0)` to `address(127)`) or 21 bytes (a `0x94` prefix followed by 20 bytes of the `address`).

This is not necessarily a problem in itself. However, this strict check means the implementation does not treat the `address` as a scalar. The Ethereum Yellow Paper states:

> When interpreting RLP data, if an expected fragment is decoded as a scalar and leading zeroes are found in the byte sequence, clients are required to consider it non-canonical and treat it in the same manner as otherwise invalid RLP data, dismissing it completely.

This ambiguity could cause the decoder to fail when processing addresses encoded as scalar values by other implementations, which might omit leading zeros and thus have different lengths.

To obtain better compatibility and alignment with the specification, consider treating the `address` as a scalar value and decoding it using its `uint256` representation. In this case, the implementation will support the decoding of addresses expressed as scalars with an arbitrary length, and the length check could be simplified to `length <= 21`.

Update: *Acknowledged, not resolved. The team stated:*

> After some reviewing, the conclusion is that encoding without the leading zeros would not be consistent with the current ethereum ecosystem. If someone wants to encode an Address without the leading zeros, they can manually do the casting to uint256 and then call the corresponding encode function. However this should not be the default encoding.

L-05 Stuck Oracle Verifications for Migrated Chains

The owner of the `BroadcasterOracle` contract is responsible for setting the `broadcasterId` for a specific chain. This setting is immutable, meaning, it cannot be changed after it is initially set.

This immutability is problematic given the possibility that L2s may change their settlement layer. For example, the migration of the settlement layer of ZKchains from Ethereum to the Gateway illustrates this scenario. When an L2 changes its parent chain, the route to verify messages adds a new pointer. This will cause the `broadcasterId` accumulator to change. Therefore, if the mapping is not updatable, the new accumulator will not match the stored `broadcasterId`, which would halt oracle verifications for that chain.

Consider adding a mechanism to update the `broadcasterId` for a chain in the event it changes its parent chain.

Update: *Acknowledged, not resolved. The team stated:*

> The team understands the issue but it is a design choice to have the setting the `broadcasterId` for a specific chain immutable. The idea is to have the least trust requirements possible on the oracle. In this case, although we need an owner to update the mapping, in order to decrease the trust assumptions, we believe it's better to not allow for updates on it, so users and solvers are sure that the oracle won't change. We also believe that a chain changing its parent chain is probably a rare event and, if that happens, we could always deploy a new oracle.

Notes & Additional Information

N-01 Gas Optimization

Within the `BlockHashProverPointer` contract, in the `setImplementationAddress` function, the `_implementationAddress` storage variable is fetched twice within the same scope. This results in an unnecessary `sload` operation.

Consider caching `_implementationAddress` to avoid the extra storage read.

***Update:** Resolved at pull request #38 at commit 807810f.*

N-02 Incomplete Docstrings

Throughout the codebase, multiple instances of incomplete docstrings were identified:

- In `BlockHashProverPointer.sol`, the `implementationCodeHash` function has no documentation for the returned value.
- In `Broadcaster.sol`, the `hasBroadcasted` function has no documentation for parameters.
- In `Receiver.sol`, the `blockHashProverCopy` function has no documentation for the parameter nor for the returned value.
- In `IReceiver.sol`, the `blockHashProverCopy` function has no documentation for the `bhpPointerId` parameter nor the returned value. Even though the interface has been extracted directly from the EIP specification, it is highly recommended to add this documentation.

Consider thoroughly documenting all functions/events (and their parameters or return values) that are part of a contract's public API. When writing docstrings, consider following the Ethereum Natural Specification Format (NatSpec).

***Update:** Resolved in pull request #39 at commit 2d7bf91.*

N-03 Floating Pragma

Pragma directives should be fixed to clearly identify the Solidity version with which the contracts will be compiled.

Throughout the codebase, multiple instances of floating pragma directives were identified:

- `BlockHashProverPointer.sol` has the `solidity ^0.8.27` floating pragma directive.
- `Broadcaster.sol` has the `solidity ^0.8.27` floating pragma directive.
- `Receiver.sol` has the `solidity ^0.8.27` floating pragma directive.
- `BroadcasterOracle.sol` has the `solidity ^0.8.26` floating pragma directive.

Consider using fixed pragma directives.

Update: *Partially Resolved in pull request #40 in commit f811731.*

N-04 Missing Docstrings

Throughout the codebase, multiple instances of missing docstrings were identified:

- In `BlockHashProverPointer.sol`, the `BlockHashProverPointer` contract
- In `BlockHashProverPointer.sol`, the `implementationAddress` function
- In `BlockHashProverPointer.sol`, the `setImplementationAddress` function
- In `Broadcaster.sol`, the `Broadcaster` contract
- In `Broadcaster.sol`, the `broadcastMessage` function
- In `Receiver.sol`, the `Receiver` contract
- In `Receiver.sol`, the `verifyBroadcastMessage` function
- In `Receiver.sol`, the `updateBlockHashProverCopy` function

Consider thoroughly documenting all functions (and their parameters) that are part of any contract's public API. Functions implementing sensitive functionality, even if not public, should be clearly documented as well. When writing docstrings, consider following the Ethereum Natural Specification Format (NatSpec).

Update: *Resolved in pull request #41 at commits 141e3da and 411487c.*

N-05 Use Custom Errors

Since Solidity version `0.8.4`, custom errors provide a cleaner and more cost-efficient way to explain to users why an operation failed.

Multiple instances of `revert` and/or `require` messages were found within `ProverUtils.sol` and `RLP`:

- In `ProverUtils`, the `require(blockHash == keccak256(rlpBlockHeader), "Block hash does not match")` statement
- In `ProverUtils`, the `require(accountExists, "Account does not exist")` statement
- In `RLP`, the `require(bytes1(item.load(0)) != 0x00)` statement

For conciseness and gas savings, consider replacing `require` and `revert` messages with custom errors.

Update: *Resolved in pull request #42 at commit 720d8a1.*

N-06 Inconsistent Use of Returns in Functions

Throughout the codebase, multiple instances of inconsistent returned values were identified:

- In `BroadcasterOracle.sol`, the `_hashPayloadHashes` function's named return value
- In `BroadcasterOracle.sol`, the `_getMessage` function's named return value
- In `RLP.sol`, the `encode(Encoder memory self)` function's named return value
- In `RLP.sol`, the `_decodeLength` function's named return value

Consider removing the redundant `return` statement in functions with named returns to improve code clarity and maintainability.

Update: *Resolved in pull request #161 in commit cd44a53 and in pull request #6106 in commit 47c8048.*

N-07 Ambiguous Documentation Of `bytes[]` Encoding

The `RLP` library provides an `encode` function for `bytes[]`-type values. The implementation simply concatenates the byte arrays provided in the input. However, this implementation can be misleading. A naive interpretation may suggest that the function encodes an array of raw byte strings. This method would cause information about the length of each individual byte array to be lost. According to the Yellow Paper specification, an array is encoded as the concatenation of the encoding of its items. The `encode(bytes[] memory input)` function actually expects a list of already encoded items. This requires users to first call `encode(string memory input)` (or a similar `encode` function) on each item before passing the resulting array to `encode(bytes[] memory input)`.

Consider improving the docstrings for the `encode(bytes[] memory input)` function. The documentation should clearly state that the function expects an array of already encoded byte strings, not raw strings, to prevent potential misuse and confusion.

Update: *Resolved in pull request #6106 in commit 78f643d.*

N-08 Unreachable Checks

Within the `_decodeLength` function of the `RLP` library, there are multiple unreachable `bytes1(item.load(0)) != 0x00` checks. The first byte of the item corresponds to the RLP prefix. In cases where this byte is `0x00`, the execution flow would have already branched in the first two `if` statements of the function (`prefix < LONG_OFFSET` and `prefix < SHORT_OFFSET`), so this check can never be reached.

Consider modifying the check to inspect the second element (index `1`) instead of the first (index `0`). This will correctly verify that the big-endian expression of the data's length is non-zero.

Update: *Resolved in pull request #6051 in commits d3c84f5 and 3e96235.*

N-09 Misleading Documentation

In the `RLP` contract, a comment within the `readBytes` function states that "Length is checked by {`toBytes`}". However, this is misleading. The length check is not performed directly by the `toBytes` function, but rather by the `slice` function, which `toBytes` calls.

Consider updating the comment to accurately reflect the fact that the `slice` function performs the length check.

Update: *Resolved in pull request #6106 in commit 55b33a0.*

N-10 Non-Canonical Long-string Decoding Acceptance

The `RLP` library's decoding function for long strings accepts length specifications that contain leading zero bytes. However, these encodings are considered non-canonical. This behavior diverges from other well-known RLP implementations, such as Go-ethereum (geth), which do not accept them. This discrepancy could lead to interoperability issues where data is considered valid by this library but invalid by other standard Ethereum clients.

Consider reverting when these non-canonical encodings are provided to align with standard RLP implementation behavior.

***Update:** Acknowledged, not resolved. The team stated:*

> As mentioned in L-03 and L-04, in order to be consistent with other libraries in the ecosystem (such as ethers.js), we chose to accept non canonical encodings with leading zeros.

N-11 Inconsistent Integer Base in Inline Assembly When Setting RLP Prefixes

In the `RLP.sol` library, RLP prefix assignments are performed using inline assembly. The integer base for these assignments is inconsistent. Both decimal and hexadecimal notations are used interchangeably across the library. The following instances of hexadecimal bases have been identified:

- `mstore(result, 0x01)`
- `mstore(result, 0x15)`

Consider consistently using the decimal integer base to improve code clarity.

***Update:** Resolved in pull request #6106 in commit 61b695f.*

Conclusion

The present Open Intents Framework (OIF) audit covered three foundational components designed to enable secure, standardized, and permissionless cross-chain interoperability: the ERC-7888 implementation, the RLP Library, and the `BroadcasterOracle` contract. The ERC-7888 contracts establish a modular verification framework for authentic cross-chain messaging. The RLP Library ensures efficient and deterministic data encoding consistent with Ethereum's canonical format, while the `BroadcasterOracle` contract integrates message broadcasting and verification within OIF's intent-based protocol, supporting complex multi-chain settlement flows through modular execution models.

During the audit, one high-severity issue was identified in the `Receiver` contract, impacting multi-route message verification, along with a medium-severity issue related to application validation within the `BroadcasterOracle`. In addition, several trust assumptions and opportunities for improving code clarity, maintainability, and overall consistency were noted, accompanied by recommendations to strengthen validation boundaries and reduce reliance on trusted components. Overall, the codebase was found to be well-structured, modular, and clearly documented, enhancing auditability and integration across OIF's cross-chain ecosystem.

The OIF team demonstrated strong technical proficiency and responsiveness throughout the review process. Their willingness to provide detailed explanations, clarify architectural decisions, and collaborate on issue resolution greatly contributed to the effectiveness of the assessment and reflected a clear commitment to delivering a robust and extensible interoperability framework.