// Security Assessment

03.07.2025 - 03.11.2025

Cross Chain Cumulative Merkle

Ether.fi

HALBORN

Cross Chain Cumulative Merkle - Ether.fi

Prepared by: HALBORN

Last Updated 03/12/2025

Date of Engagement by: March 7th, 2025 - March 11th, 2025

Summary

0% ⓘ OF ALL REPORTED FINDINGS HAVE BEEN ADDRESSED

ALL FINDINGS	CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
10	0	0	0	2	8

TABLE OF CONTENTS

1. Introduction

Ether.Fi engaged Halborn to conduct a security assessment on their smart contracts beginning on March 7th, 2025 and ending on March 12th, 2025. The security assessment was scoped to the smart contracts provided to Halborn. Commit hashes and further details can be found in the Scope section of this report.

The Ether.Fi codebase in scope consists of a cross-chain token distribution contract built on LayerZero that allows users to claim tokens based on Merkle proofs.

2. Assessment Summary

Halborn was provided 4 days for the engagement and assigned 1 full-time security engineer to review the security of the smart contracts in scope. The engineer is a blockchain and smart contract security expert with advanced penetration testing and smart contract hacking skills, and deep knowledge of multiple blockchain protocols.

The purpose of the assessment is to:

- Identify potential security issues within the smart contracts.
- Ensure that smart contract functionality operates as intended.

In summary, Halborn identified some improvements to reduce the likelihood and impact of risks, which should be addressed by the Ether.Fi team. The main ones are the following:

- Add appropriate access control to the initializeLayerZero() function.
- Implement an administrative withdrawal function to allow the contract owner or a designated role to manage excess funds.
- Switch modifier order to consistently place the nonReentrant modifier as the first one to run.

3. Test Approach And Methodology

Halborn performed a combination of manual review of the code and automated security testing to balance efficiency, timeliness, practicality, and accuracy in regard to the scope of this assessment. While manual testing is recommended to uncover flaws in logic, process, and implementation; automated testing techniques help enhance coverage of smart contracts and can quickly identify items that do not follow security best practices.

The following phases and associated tools were used throughout the term of the assessment:

- Research into architecture, purpose and use of the platform.
- Smart contract manual code review and walkthrough to identify any logic issue.
- Thorough assessment of safety and usage of critical Solidity variables and functions in scope that could led to arithmetic related vulnerabilities.
- Local testing with custom scripts (Foundry).
- Fork testing against main networks (Foundry).
- Static analysis of security for scoped contract, and imported functions (Slither).

4. Static Analysis Report

4.1 Description

Halborn used automated testing techniques to enhance the coverage of certain areas of the smart contracts in scope. Among the tools used was Slither, a Solidity static analysis framework. After Halborn verified the smart contracts in the repository and was able to compile them correctly into their abis and binary format, Slither was run against the contracts. This tool can statically verify mathematical relationships between Solidity variables to detect invalid or inconsistent usage of the contracts' APIs across the entire code-base.

The security team assessed all findings identified by the Slither software, however, findings with related to external dependencies are not included in the below results for the sake of report readability.

4.2 Output

The findings obtained as a result of the Slither scan were reviewed, and many were not included in the report because they were determined as false positives.

```
INFO:Detectors:
TimelockController._execute(address,uint256,bytes) (node_modules/@openzeppelin/contracts/governance/TimelockController.sol#412-415) ignores return value by Address.verifyCallResult(success,returndata) (node_modules/@openzeppelin/contracts/governance/TimelockController.sol#414)
CumulativeMerkleDrop.addChain(uint32,uint128,bytes32) (src/merkle-drop/CumulativeMerkleDrop.sol#184-187) ignores return value by peerToGasLimit.set(uint256(eid),uint256(singleMessageGasLimit)) (src/merkle-drop/CumulativeMerkleDrop.sol#186)
CumulativeMerkleDrop.removeChain(uint32) (src/merkle-drop/CumulativeMerkleDrop.sol#192-195) ignores return value by peerToGasLimit.remove(uint256(eid)) (src/merkle-drop/CumulativeMerkleDrop.sol#194)
CumulativeMerkleDrop.topUpPeer(uint32,uint256) (src/merkle-drop/CumulativeMerkleDrop.sol#289-311) ignores return value by IERC20(token).approve(address(oft),amount) (src/merkle-drop/CumulativeMerkleDrop.sol#309)
CumulativeMerkleDrop.topUpPeer(uint32,uint256) (src/merkle-drop/CumulativeMerkleDrop.sol#289-311) ignores return value by oft.send{value: msgFee.nativeFee}(param,msgFee,msg.sender) (src/merkle-drop/CumulativeMerkleDrop.sol#310)
CumulativeMerkleDrop._lzSendFromContractBalance(uint32,bytes,bytes,MessagingFee) (src/merkle-drop/CumulativeMerkleDrop.sol#317-319) ignores return value by endpoint.send{value: msgFee.nativeFee}(MessagingParams(dstEid,_getPeerOrRevert(dstEid),message,options,msgFee.lzTokenFee > 0),msg.sender) (src/merkle-drop/CumulativeMerkleDrop.sol#318)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#unused-return
INFO:Detectors:
CumulativeMerkleDrop.constructor(address,address,address)._token (src/merkle-drop/CumulativeMerkleDrop.sol#96) lacks a zero-check on :
        - token = _token (src/merkle-drop/CumulativeMerkleDrop.sol#97)
CumulativeMerkleDrop.constructor(address,address,address)._oftAdapter (src/merkle-drop/CumulativeMerkleDrop.sol#96) lacks a zero-check on :
        - oftAdapter = _oftAdapter (src/merkle-drop/CumulativeMerkleDrop.sol#98)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#missing-zero-address-validation
INFO:Detectors:
CumulativeMerkleDrop.batchUpdateClaimEid(address[],uint32) (src/merkle-drop/CumulativeMerkleDrop.sol#245-263) has external calls inside a loop: endpoint.eid() != getClaimEid(accounts[i]) (src/merkle-drop/CumulativeMerkleDrop.sol#252)
CumulativeMerkleDrop._lzSendFromContractBalance(uint32,bytes,bytes,MessagingFee) (src/merkle-drop/CumulativeMerkleDrop.sol#317-319) has external calls inside a loop: endpoint.send{value: msgFee.nativeFee}(MessagingParams(dstEid,_getPeerOrRevert(dstEid),message,options,msgFee.lzTokenFee > 0),msg.sender) (src/merkle-drop/CumulativeMerkleDrop.sol#318)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#calls-inside-a-loop
INFO:Detectors:
Reentrancy in CumulativeMerkleDrop.batchUpdateClaimEid(address[],uint32) (src/merkle-drop/CumulativeMerkleDrop.sol#245-263):
        External calls:
        - _lzSendFromContractBalance(dstEid,message,OptionsBuilder.newOptions().addExecutorLzReceiveOption(1,0),msgFee) (src/merkle-drop/CumulativeMerkleDrop.sol#260)
                - endpoint.send{value: msgFee.nativeFee}(MessagingParams(dstEid,_getPeerOrRevert(dstEid),message,options,msgFee.lzTokenFee > 0),msg.sender) (src/merkle-drop/CumulativeMerkleDrop.sol#318)
        Event emitted after the call(s):
        - ClaimEidUpdatedBatched(dstEid) (src/merkle-drop/CumulativeMerkleDrop.sol#262)
Reentrancy in CumulativeMerkleDrop.setAndBroadcastMerkleRoot(bytes32) (src/merkle-drop/CumulativeMerkleDrop.sol#268-284):
        External calls:
        - _lzSendFromContractBalance(dstEid,message,getExecutorReceiveOptions(dstEid),msgFee) (src/merkle-drop/CumulativeMerkleDrop.sol#281)
                - endpoint.send{value: msgFee.nativeFee}(MessagingParams(dstEid,_getPeerOrRevert(dstEid),message,options,msgFee.lzTokenFee > 0),msg.sender) (src/merkle-drop/CumulativeMerkleDrop.sol#318)
        Event emitted after the call(s):
        - MerkleRootBroadcasted(dstEid,merkleRoot) (src/merkle-drop/CumulativeMerkleDrop.sol#282)
Reentrancy in CumulativeMerkleDrop.updateClaimEid(uint32,MessagingFee) (src/merkle-drop/CumulativeMerkleDrop.sol#225-238):
        External calls:
        - _lzSend(dstEid,message,getExecutorReceiveOptions(dstEid),msgFee,msg.sender) (src/merkle-drop/CumulativeMerkleDrop.sol#235)
                - returndata = address(token).functionCall(data) (node_modules/@openzeppelin/contracts/token/ERC20/utils/SafeERC20.sol#96)
                - endpoint.send{value: messageValue}(MessagingParams(_dstEid,_getPeerOrRevert(_dstEid),_message,_options,_fee.lzTokenFee > 0),_refundAddress) (node_modules/layerzero-v2/oapp/contracts/oapp/OAppSenderUpgradeable.sol#93-98)
                - IERC20(lzToken).safeTransferFrom(msg.sender,address(endpoint),_lzTokenFee) (node_modules/layerzero-v2/oapp/contracts/oapp/OAppSenderUpgradeable.sol#130)
                - (success,returndata) = target.call{value: value}(data) (node_modules/@openzeppelin/contracts/utils/Address.sol#87)
        External calls sending eth:
        - _lzSend(dstEid,message,getExecutorReceiveOptions(dstEid),msgFee,msg.sender) (src/merkle-drop/CumulativeMerkleDrop.sol#235)
                - endpoint.send{value: messageValue}(MessagingParams(_dstEid,_getPeerOrRevert(_dstEid),_message,_options,_fee.lzTokenFee > 0),_refundAddress) (node_modules/layerzero-v2/oapp/contracts/oapp/OAppSenderUpgradeable.sol#93-98)
                - (success,returndata) = target.call{value: value}(data) (node_modules/@openzeppelin/contracts/utils/Address.sol#87)
        Event emitted after the call(s):
        - ClaimEidUpdated(msg.sender,dstEid) (src/merkle-drop/CumulativeMerkleDrop.sol#237)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#reentrancy-vulnerabilities-3
INFO:Slither:. analyzed (132 contracts with 100 detectors), 13 result(s) found
```

5. RISK METHODOLOGY

Every vulnerability and issue observed by Halborn is ranked based on **two sets** of **Metrics** and a **Severity Coefficient**. This system is inspired by the industry standard Common Vulnerability Scoring System.

The two **Metric sets** are: **Exploitability** and **Impact**. **Exploitability** captures the ease and technical means by which vulnerabilities can be exploited and **Impact** describes the consequences of a successful exploit.

The **Severity Coefficients** is designed to further refine the accuracy of the ranking with two factors: **Reversibility** and **Scope**. These capture the impact of the vulnerability on the environment as well as the number of users and smart contracts affected.

The final score is a value between 0-10 rounded up to 1 decimal place and 10 corresponding to the highest security risk. This provides an objective and accurate rating of the severity of security vulnerabilities in smart contracts.

The system is designed to assist in identifying and prioritizing vulnerabilities based on their level of risk to address the most critical issues in a timely manner.

5.1 EXPLOITABILITY

ATTACK ORIGIN (AO):

Captures whether the attack requires compromising a specific account.

ATTACK COST (AC):

Captures the cost of exploiting the vulnerability incurred by the attacker relative to sending a single transaction on the relevant blockchain. Includes but is not limited to financial and computational cost.

ATTACK COMPLEXITY (AX):

Describes the conditions beyond the attacker's control that must exist in order to exploit the vulnerability. Includes but is not limited to macro situation, available third-party liquidity and regulatory challenges.

METRICS:

EXPLOITABILITY METRIC (M_E)	METRIC VALUE	NUMERICAL VALUE
Attack Origin (AO)	Arbitrary (AO:A) Specific (AO:S)	1 0.2

EXPLOITABILITY METRIC (M_E)	METRIC VALUE	NUMERICAL VALUE
Attack Cost (AC)	Low (AC:L) Medium (AC:M) High (AC:H)	1 0.67 0.33
Attack Complexity (AX)	Low (AX:L) Medium (AX:M) High (AX:H)	1 0.67 0.33

Exploitability E is calculated using the following formula:

$$E = \prod m_e$$

5.2 IMPACT

CONFIDENTIALITY (C):

Measures the impact to the confidentiality of the information resources managed by the contract due to a successfully exploited vulnerability. Confidentiality refers to limiting access to authorized users only.

INTEGRITY (I):

Measures the impact to integrity of a successfully exploited vulnerability. Integrity refers to the trustworthiness and veracity of data stored and/or processed on-chain. Integrity impact directly affecting Deposit or Yield records is excluded.

AVAILABILITY (A):

Measures the impact to the availability of the impacted component resulting from a successfully exploited vulnerability. This metric refers to smart contract features and functionality, not state. Availability impact directly affecting Deposit or Yield is excluded.

DEPOSIT (D):

Measures the impact to the deposits made to the contract by either users or owners.

YIELD (Y):

Measures the impact to the yield generated by the contract for either users or owners.

METRICS:

IMPACT METRIC (M_I)	METRIC VALUE	NUMERICAL VALUE
Confidentiality (C)	None (I:N) Low (I:L) Medium (I:M) High (I:H) Critical (I:C)	0 0.25 0.5 0.75 1
Integrity (I)	None (I:N) Low (I:L) Medium (I:M) High (I:H) Critical (I:C)	0 0.25 0.5 0.75 1
Availability (A)	None (A:N) Low (A:L) Medium (A:M) High (A:H) Critical (A:C)	0 0.25 0.5 0.75 1
Deposit (D)	None (D:N) Low (D:L) Medium (D:M) High (D:H) Critical (D:C)	0 0.25 0.5 0.75 1
Yield (Y)	None (Y:N) Low (Y:L) Medium (Y:M) High (Y:H) Critical (Y:C)	0 0.25 0.5 0.75 1

Impact I is calculated using the following formula:

$$I = max(m_I) + \frac{\sum m_I - max(m_I)}{4}$$

5.3 SEVERITY COEFFICIENT

REVERSIBILITY (R):

Describes the share of the exploited vulnerability effects that can be reversed. For upgradeable contracts, assume the contract private key is available.

SCOPE (S):

Captures whether a vulnerability in one vulnerable contract impacts resources in other contracts.

METRICS:

SEVERITY COEFFICIENT (C)	COEFFICIENT VALUE	NUMERICAL VALUE
Reversibility (r)	None (R:N) Partial (R:P) Full (R:F)	1 0.5 0.25

SEVERITY COEFFICIENT (C)	COEFFICIENT VALUE	NUMERICAL VALUE
Scope (s)	Changed (S:C) Unchanged (S:U)	1.25 1

Severity Coefficient C is obtained by the following product:

$$C = rs$$

The Vulnerability Severity Score S is obtained by:

$$S = min(10, EIC * 10)$$

The score is rounded up to 1 decimal places.

SEVERITY	SCORE VALUE RANGE
Critical	9 - 10
High	7 - 8.9
Medium	4.5 - 6.9
Low	2 - 4.4
Informational	0 - 1.9

6. SCOPE

FILES AND REPOSITORY

(a) Repository: lrt-square-sc

(b) Assessed Commit ID: eff3ba8

(c) Items in scope:

- src/merkle-drop/CumulativeMerkleCodec.sol
- src/merkle-drop/CumulativeMerkleDrop.sol

Out-of-Scope: Third party dependencies and economic attacks.

Out-of-Scope: New features/implementations after the remediation commit IDs.

7. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW
0	0	0	2

INFORMATIONAL
8

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
HAL-01 - UNPROTECTED REINITIALIZER FUNCTION ALLOWS UNAUTHORIZED CONTRACT CONFIGURATION	LOW	PENDING
HAL-02 - MISSING EMERGENCY WITHDRAWAL MECHANISM	LOW	PENDING

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
HAL-03 - ORDERING FOR NONREENTRANT MODIFIER	INFORMATIONAL	PENDING
HAL-04 - UNHANDLED RETURN VALUES	INFORMATIONAL	PENDING
HAL-05 - MISSING EVENTS	INFORMATIONAL	PENDING
HAL-06 - COMMENTED CODE	INFORMATIONAL	PENDING
HAL-07 - FLOATING PRAGMA	INFORMATIONAL	PENDING
HAL-08 - LACK OF NAMED MAPPINGS	INFORMATIONAL	PENDING
HAL-09 - INEFFICIENT CODE	INFORMATIONAL	PENDING
HAL-10 - USE OF REVERT STRINGS INSTEAD OF CUSTOM ERRORS	INFORMATIONAL	PENDING

8. FINDINGS & TECH DETAILS

8.1 (HAL-01) UNPROTECTED REINITIALIZER FUNCTION ALLOWS UNAUTHORIZED CONTRACT CONFIGURATION

// LOW

Description

The `initializeLayerZero()` function from the `CumulativeMerkleDrop` contract lacks access control protection.

```
function initializeLayerZero(uint128 _batchMessageGasLimit) external r
    __OAppCore_init(owner());
    batchMessageGasLimit = _batchMessageGasLimit;
}
```

While the `reinitializer(2)` modifier ensures this function can only be called once when upgrading from version 1 to 2, it doesn't restrict *who* can call it. This allows any external actor to execute this critical initialization function before the legitimate contract owner, potentially setting the `batchMessageGasLimit` to an arbitrary value, possibly affecting cross-chain message execution and gas consumption.

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:L/I:L/D:N/Y:N (3.1)

Recommendation

Add appropriate access control to the `initializeLayerZero()` function by using the existing role-based mechanisms, for example:

```
if (msg.sender != owner()) revert Unauthorized();
```

References

LRT2-protocol/lrt-square-sc/src/merkle-drop/CumulativeMerkleDrop.sol#L111-L114

8.2 (HAL-02) MISSING EMERGENCY WITHDRAWAL MECHANISM

// LOW

Description

The `CumulativeMerkleDrop` contract includes a `receive() external payable {}` function that allows it to accept native assets, and requires sufficient balance to pay for cross-chain Layer Zero messaging fees. However, the contract lacks any mechanism to withdraw excess or unused funds.

While the contract intentionally holds and uses native assets for legitimate purposes, it still lacks any mechanism to withdraw excess or unused funds that might accumulate.

If the contract receives more native assets than needed for operations, unintentionally, or if operations need to be paused/migrated, there's no way to recover these funds.

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:N/D:L/Y:N (2.5)

Recommendation

Implement an administrative withdrawal function to allow the contract owner or a designated role to manage excess funds.

References

LRT2-protocol/lrt-square-sc/src/merkle-drop/CumulativeMerkleDrop.sol#L419

8.3 (HAL-03) ORDERING FOR NONREENTRANT MODIFIER

// INFORMATIONAL

Description

In Solidity, if a function has multiple modifiers, they are executed in the order specified. If checks or logic of modifiers depend on other modifiers, this has to be considered in their ordering.

Several functions of the contracts in scope have multiple modifiers, with one of them being `nonReentrant` which prevents reentrancy behavior on the functions. Ideally, the `nonReentrant` modifier should be the first one to prevent even the execution of other modifiers in case of reentrancy behavior.

While there is currently no obvious vulnerability with `nonReentrant` not being the first modifier, placing it first ensures that all other modifiers are executed only if the call is non-reentrant. This is a safer practice and can prevent potential issues in future updates or unforeseen scenarios.

BVSS

AO:A/AC:H/AX:H/R:N/S:U/C:N/A:N/I:N/D:L/Y:N (0.3)

Recommendation

Switch modifier order to consistently place the `nonReentrant` modifier as the first one to run so that all other modifiers are executed only if the call is non-reentrant.

References

LRT2-protocol/lrt-square-sc/src/merkle-drop/CumulativeMerkleDrop.sol#L126

8.4 (HAL-04) UNHANDLED RETURN VALUES

// INFORMATIONAL

Description

The call to the `_lzSend()` function inside the `updateClaimEid()` external function of the `CumulativeMerkleDrop` contract returns a receipt of type `MessagingReceipt.` However, this return value is not handled inside the function.

Similarly, the `send()` function for the IOFT interface is expected to return types `MessagingReceipt` and `OFTReceipt` which are not handled.

Ignoring return values is a common anti-pattern that may obscure important error conditions, making it difficult to detect failures, debug issues, or ensure operations completed as expected. This can ultimately compromise the reliability and security of the system.

BVSS

AO:S/AC:M/AX:M/R:N/S:U/C:N/A:N/I:L/D:N/Y:N (0.2)

Recommendation

Check and handle return values from function calls to prevent silent failures and ensure expected outcomes, especially for critical operations that may affect system state or security.

References

LRT2-protocol/lrt-square-sc/src/merkle-drop/CumulativeMerkleDrop.sol#L247
LRT2-protocol/lrt-square-sc/src/merkle-drop/CumulativeMerkleDrop.sol#L320

8.5 (HAL-05) MISSING EVENTS

// INFORMATIONAL

Description

In the contract in scope, there are instances where administrative functions change contract state by modifying core state variables without them being reflected in event emissions.

The absence of events may hamper effective state tracking in off-chain monitoring systems.

Instances of this issue can be found in:

- `CumulativeMerkleDrop.setUserChainSwitchingEnabled()`
- `CumulativeMerkleDrop.setBatchMessageGasLimit()`
- `CumulativeMerkleDrop.addChain()`
- `CumulativeMerkleDrop.removeChain()`

BVSS

AO:S/AC:H/AX:H/R:N/S:U/C:N/A:N/I:L/D:N/Y:N (0.1)

Recommendation

Emit events for all state changes that occur as a result of administrative functions to facilitate off-chain monitoring of the system.

References

LRT2-protocol/lrt-square-sc/src/merkle-drop/CumulativeMerkleDrop.sol#L395-L401
LRT2-protocol/lrt-square-sc/src/merkle-drop/CumulativeMerkleDrop.sol#L191-L205

8.6 (HAL-06) COMMENTED CODE

// INFORMATIONAL

Description

The `CumulativeMerkleDrop` contract has an instance where code has been commented out, indicating that certain functionality may have been removed or disabled.

While commenting out code can be useful for debugging or testing purposes, it can also lead to confusion and make the codebase harder to maintain.

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:N/D:N/Y:N (0.0)

Recommendation

It is recommended to remove the commented code.

References

LRT2-protocol/lrt-square-sc/src/merkle-drop/CumulativeMerkleDrop.sol#L55

8.7 (HAL-07) FLOATING PRAGMA

// INFORMATIONAL

Description

The `CumulativeMerkleDrop` contract currently uses floating pragma versions `^0.8.24` which means that the code can be compiled by any compiler version that is greater than or equal to `0.8.24`, and less than `0.9.0`.

However, it is recommended that contracts should be deployed with the same compiler version and flags used during development and testing. Locking the pragma helps to ensure that contracts do not accidentally get deployed using another pragma. For example, an outdated pragma version might introduce bugs that affect the contract system negatively.

Additionally, from Solidity versions `0.8.20` through `0.8.24`, the default target EVM version is set to `Shanghai`, which results in the generation of bytecode that includes `PUSH0` opcodes. Starting with version `0.8.25`, the default EVM version shifts to `Cancun`, introducing new opcodes for transient storage, `TSTORE` and `TLOAD`.

In this aspect, it is crucial to select the appropriate EVM version when it's intended to deploy the contracts on networks other than the Ethereum mainnet, which may not support these opcodes. Failure to do so could lead to unsuccessful contract deployments or transaction execution issues.

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:N/D:N/Y:N (0.0)

Recommendation

Lock the pragma version to the same version used during development and testing.

Additionally, make sure to specify the target EVM version when using Solidity versions from `0.8.20` and above if deploying to chains that may not support newly introduced opcodes. Additionally, it is crucial to stay informed about the opcode support of different chains to ensure smooth deployment and compatibility.

References

LRT2-protocol/lrt-square-sc/src/merkle-drop/CumulativeMerkleDrop.sol#L25

8.8 (HAL-08) LACK OF NAMED MAPPINGS

// INFORMATIONAL

Description

The project contains several unnamed mappings despite using a Solidity version that supports named mappings.

Named mappings improve code readability and self-documentation by explicitly stating their purpose.

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:N/D:N/Y:N (0.0)

Recommendation

Consider refactoring the mappings to use named arguments, which will enhance code readability and make the purpose of each mapping more explicit. For example:

```
mapping(address myAddress => bool myBool) myMapping;
```

References

LRT2-protocol/lrt-square-sc/src/merkle-drop/CumulativeMerkleDrop.sol#L69-L74

8.9 (HAL-09) INEFFICIENT CODE

// INFORMATIONAL

Description

In the `setAndBroadcastMerkleRoot` function, the contract performs unnecessary storage reads (`SLOAD` operations) of the `merkleRoot` state variable, which is inefficient from a gas perspective.

The function first updates the `merkleRoot` storage variable through the `setMerkleRoot()` call, but then reads this same value from storage twice:
1. When encoding the message: `CumulativeMerkleCodec.encodeMerkleRoot(merkleRoot)`
2. In the event emission: `emit MerkleRootBroadcasted(dstEid, merkleRoot)`

```
function setAndBroadcastMerkleRoot(bytes32 merkleRoot_) external onlyR
    setMerkleRoot(merkleRoot_);

    bytes memory message = CumulativeMerkleCodec.encodeMerkleRoot(merk

    // enumerate all the peers and broadcast message
    uint256[] memory allPeers = peerToGasLimit.keys();
    for (uint256 i = 0; i < allPeers.length; i++) {
        uint32 dstEid = uint32(allPeers[i]);

        MessagingFee memory msgFee = quoteBroadcastMerkleRoot(dstEid);
        if (address(this).balance < msgFee.nativeFee) revert Insuffici

        _lzSendFromContractBalance(dstEid, message, getExecutorReceive
        emit MerkleRootBroadcasted(dstEid, merkleRoot);
    }
}
```

Each storage read (`SLOAD`) costs a minimum of 100 gas, and these operations are unnecessary because the new merkle root value is already available as the `merkleRoot_` parameter, which is in memory.

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:N/D:N/Y:N (0.0)

Recommendation

Replace the storage reads of `merkleRoot` with the parameter `merkleRoot_` that's already in memory.

References

LRT2-protocol/lrt-square-sc/src/merkle-drop/CumulativeMerkleDrop.sol#L278-L294

8.10 (HAL-10) USE OF REVERT STRINGS INSTEAD OF CUSTOM ERRORS

// INFORMATIONAL

Description

In the `encodeBatch()` function of the `CumulativeMerkleCodec` library, there is use of revert strings over custom errors.

In Solidity development, replacing hard-coded revert message strings with the `Error()` syntax is an optimization strategy that can significantly reduce gas costs. Hard-coded strings, stored on the blockchain, increase the size and cost of deploying and executing contracts.

The `Error()` syntax allows for the definition of reusable, parameterized custom errors, leading to a more efficient use of storage and reduced gas consumption. This approach not only optimizes gas usage during deployment and interaction with the contract but also enhances code maintainability and readability by providing clearer, context-specific error information.

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:N/D:N/Y:N (0.0)

Recommendation

Consider replacing all revert strings with custom errors. For example:

```
error ConditionNotMet();

if (!condition) revert ConditionNotMet();
```

or starting from Solidity `0.8.27` :

```
require(condition, ConditionNotMet());
```

For more references, see here and here.

References

LRT2-protocol/lrt-square-sc/src/merkle-drop/CumulativeMerkleCodec.sol#L33

Halborn strongly recommends conducting a follow-up assessment of the project either within six months or immediately following any material changes to the codebase, whichever comes first. This approach is crucial for maintaining the project's integrity and addressing potential vulnerabilities introduced by code modifications.