SMART CONTRACT SECURITY AUDIT REPORT

Client: Liquidity Gate

Date: 2nd of August 2025

Appendix

Appendix....2
Audit Summary....4
Introduction....7
  Note on Updated Files Introduced by Client....7
Overview....8
Executive Summary....9
  Scope overview....9
  Methodology....9
Code Breakdown....10
Definitions....11
Critical Severity Issues....12
  C01. Missing closing parenthesis in require statement prevents compilation....12
High Severity Issues....14
  H01. Relay-Controlled Denial of Service Through Failing ETH Transfers....14
  H02. Missing Reentrancy Protection May Allow Double Execution or Inconsistent State....16
  H03. Lack of Caller Restriction on executeRewardSnapshot Allows Untrusted Execution and Front-Running....18
Medium Severity Issues....20
  M01. External Balance Reads in Reward Validation May Be Exploitable via Flash Loan Manipulation....20
  M02. Integer Truncation and Missing Division Check May Cause Unexpected Consensus Behavior....23
  M03. Partial State Update Before External Calls....25
Low Severity Issues....28
  L01. Unbounded Loop in View Function Can Revert on Large Arrays and Disrupt Off-Chain Services....28
  L02. Missing Validation of Merkle Root or Tree Data May Allow Submission of Invalid Rewards 30
  L03. Missing Input Validation in executeRewardSnapshot....32
Informational Severity Issues....34
  I01. Potential Gas Optimization by Merging Redundant Loops in Reward Aggregation....34
  I02. Unnecessary Hashing of Full Struct Increases Gas Costs....36
  I03. Acceptance of Historical Submissions That Will Never Execute....38
Disclaimers....40
Appendix 1. Severity Definitions....41
Risk Levels....41
Impact Levels....42
Likelihood Levels....43

We thank Liquidity Gate for allowing us to conduct a Smart Contract Audit. This document outlines our methodology, limitations, and results of the security audit.

Timeline: 29.07.2025 – 02.08.2025

Audit Summary

Liquidity Gate should acknowledge all the risks summed up in the risks section of the report.

13	13	0	0
Total Findings	Resolved	Acknowledged	Mitigated

Findings by severity	Findings Number	Resolved	Mitigated	Acknowledged
Critical	1	1	0	0
High	3	3	0	0
Medium	3	3	0	0
Low	3	3	0	0
Informational	3	3	0	0

Vulnerability	Severity
C01. Missing closing parenthesis in require statement prevents compilation	Critical
H01. Relay-Controlled Denial of Service Through Failing ETH Transfers	High
H02. Missing Reentrancy Protection May Allow Double Execution or Inconsistent State	High
H03. Lack of Caller Restriction on executeRewardSnapshot Allows Untrusted Execution and Front-Running	High
M01. External Balance Reads in Reward Validation May Be Exploitable via Flash Loan Manipulation	Medium
M02. Integer Truncation and Missing Division Check May Cause Unexpected Consensus Behavior	Medium
M03. Partial State Update Before External Calls	Medium
L01. Unbounded Loop in View Function Can Revert on Large Arrays and Disrupt Off-Chain Services	Low
L02. Missing Validation of Merkle Root or Tree Data May Allow Submission of Invalid Rewards	Low
L03. Missing Input Validation in executeRewardSnapshot	Low
I01. Gas Optimization Opportunity by Merging Redundant Loops in Reward Aggregation	Informational
I02. Unnecessary Hashing of Full Struct Increases Gas Costs	Informational
I03. Acceptance of Historical Submissions That Will Never Execute	Informational

Introduction

Hakflow was commissioned by Liquidity Gate to perform a Smart Contract Security Audit of the Rewards Pool token. This report details the findings from the security audit conducted between 29.07.2025 and 02.08.2025, including our remediation efforts and recommendations for enhancing the code's security posture.

The objective was to identify vulnerabilities, assess their impact and likelihood, and provide strategic insights into strengthening the client side security.

Scope:

Repository:

- https://github.com/LiquidityGate/lqg-contracts/blob/main/contracts/contract/rewards/LQGRewardsPool.sol

Commit:

- 5c2af59febd37a9f89c23d1c8175c028b225c6a8

Note on Updated Files Introduced by Client

During the audit, the client introduced updated contracts named *LQGRewardsPool.sol, LQGRewardsPool1.sol*, and *LQGRewardsPool2.sol*, stating that these files addressed all previously identified findings. However, these updated files were not formally submitted or requested for auditing as part of the current engagement. As a result, any additional issues or vulnerabilities that may arise from these new code changes or additions are considered out of scope for this audit.

Timeline:

- **Assessment Start Date:** 29.07.2025
- **Assessment End Date:** 01.08.2025
- **Report Submission:** 02.08.2025

Team Composition:

- **Project Lead**: Ensures quality control and oversees project delivery.
- **Security Auditor**: Conduct the security audit.

Overview

Repository	https://github.com/LiquidityGate/lqg-contracts/blob/main/contracts/contract/rewards/LQGRewardsPool.sol
Commit	5c2af59febd37a9f89c23d1c8175c028b225c6a8
Branch	main
Platform	EVM
Language	Solidity
Tags	Rewards

Executive Summary

The Liquidity Gate Rewards Pool contract implements a staking ecosystem that collects and distributes rewards to participants. It derives from a common base contract and is based on a rewards-pool interface, showing that it follows a standard structure within its system. The contract keeps track of an ever-increasing reward index, records submissions from trusted operators and then delivers both LQG tokens and ETH to the appropriate recipients once certain conditions are met. Events are used to announce when rewards are submitted, when snapshots are finalized and if any relay has trouble delivering rewards.

The contract acts like a ledger and distribution hub. It gathers reward information from trusted network nodes, verifies that the totals match what is available, then tallies the number of submissions and waits until there is sufficient agreement before sending out the rewards. The design includes safeguards like limiting how many networks can be included in a submission and checking that the submission periods are valid. Once everything is in order, the contract interacts with other system components, minting new tokens if needed, updating timestamps and sending tokens and ETH to relay contracts that handle the final payouts.

Scope overview

The Hackflow audit team audited only the code of the LQGRewardsPool.sol contract. Our review focused on how this contract collects reward information, maintains internal records and distributes assets. Any integration with external contracts or other dependencies that this contract uses and utilizes was not included in the audit scope. We recommended auditing those dependencies separately because they contribute to the overall behavior and have a significant impact on how the contract operates.

Methodology

The security evaluation was conducted using a blend of automated scanning tools and manual inspection to identify and validate security weaknesses. The assessment included:

- **Manual Review:** Deep review source code.

Code Breakdown

The contract begins by declaring events to log reward submissions, snapshot executions and any distribution failures. It maintains a MAX_NETWORK_COUNT constant to limit the size of reward submissions and sets its version number upon construction. Several public view functions return information such as the current reward index, the amount of LQG and ETH available for distribution, and details about claim intervals. Functions also allow callers to see what percentage of rewards a particular claiming contract is entitled to and whether trusted nodes have submitted their data.

The core functionality lies in submitRewardSnapshot, which allows trusted nodes to send a snapshot of the rewards to be distributed. This function confirms that submissions are enabled, checks that the submission is for a valid period and ensures that arrays within the submission have matching lengths. It then totals the proposed rewards and compares them to what is available. To avoid duplicate entries, the contract records each node's submission status. It emits an event to announce the submission and, if the submission matches the current reward index, assesses whether there are enough submissions to proceed.

When enough submissions are received, either through submitRewardSnapshot or explicitly via executeRewardSnapshot, the contract finalizes the snapshot. This process mints any pending tokens, updates the reward index and claim interval times, and records the block and address used for execution. It then transfers a portion of LQG to a treasury address, sends any user-entitled ETH to a smoothing pool and loops through each network to distribute the respective rewards. For each network, it retrieves a relay contract address from storage, withdraws the appropriate amount of tokens and ETH and calls the relay to handle the actual distribution. If a relay encounters an issue, the contract logs it via an event, allowing the rest of the distribution to continue smoothly.

Overall, the contract presents a systematic approach to recording reward data and distributing rewards once consensus is reached. By clearly separating the submission, verification and distribution steps, and by using events to communicate progress, the contract provides a transparent and orderly mechanism for managing staking rewards.

Definitions

Table. Issue Severity Definition

Severity	Description
Critical	These issues present a major security vulnerability that poses a severe risk to the system. They require immediate attention and must be resolved to prevent a potential security breach or other significant harm.
High	These issues present a significant risk to the system, but may not require immediate attention. They should be addressed in a timely manner to reduce the risk of the potential security breach.
Medium	These issues present a moderate risk to the system and cannot have a great impact on its function. They should be addressed in a reasonable time frame, but may not require immediate attention.
Low	These issues present no risk to the system and typically relate to the code quality problems or general recommendations. They do not require immediate attention and should be viewed as a minor recommendation.
Informational	These findings provide insights into the system or code but do not represent immediate or direct risks to its functionality, security, or stability. They typically offer guidance for potential improvements and best practices.

Issue Status Definition

Status	Description
New	The issue was presented to the customer. The remediation after the initial discovery was not yet made.
Resolved	The issue has been fully addressed and fixed.
Acknowledged	The issue was not fixed as a result of the remediation. The customer was informed of the risks associated with it.
Mitigated	The issue was reduced or controlled, but not necessarily completely eliminated.

■ ■ Critical Severity Issues

C01. Missing closing parenthesis in require statement prevents compilation

Severity	Critical
Impact	High
Likelihood	High
Status	**Resolved**

Description:

The `LQGRewardsPool` contract includes a `require` statement that is responsible for ensuring a trusted node does not submit a reward snapshot more than once for the same reward index. This check is critical in maintaining the integrity of the reward consensus mechanism by preventing duplicate voting from the same node. However, the statement is missing a closing parenthesis at the end of the line, which causes a compilation failure.

Because of this syntax error, the contract cannot be deployed or executed in its current form. This halts the entire reward submission and distribution process, rendering the core functionality of the contract inoperable until fixed. The impact is severe from a deployment and operational perspective, as it blocks the use of the contract entirely in production environments.

Applicable Code:

This finding applies to the following lines of code in the submitRewardSnapshot function:

Line#168

```
require(!getBool(keccak256(abi.encode("rewards.snapshot.submitted.node", msg.sender,
_submission.rewardIndex)), "Already submitted for this reward period");
```

Recommendation:

The missing parenthesis in the `require` statement should be added to restore proper syntax and enable the contract to compile and deploy correctly. The development team should also consider running syntax linting or static analysis tools to catch similar issues before deployment, as such errors are easily preventable during the development phase.

■■High Severity Issues

H01. Relay-Controlled Denial of Service Through Failing ETH Transfers

Severity	High
Impact	High
Likelihood	Medium
Status	**Resolved**

Description:

The `_executeRewardSnapshot` function is responsible for distributing LQG and ETH rewards to multiple relay contracts corresponding to different networks. Each relay receives the designated reward amounts, and afterward, a call is made to trigger the actual reward handling. Although this process is partially wrapped in a `try-catch` block to prevent full reverts when the relay logic fails, there are still crucial external calls, such as ETH transfers, that occur outside this protection.

Because ETH transfers rely on the fallback or receive logic of the recipient relay contract, a malicious or faulty relay can revert during the transfer phase itself, before reaching the `try-catch` scope. This would cause the entire `_executeRewardSnapshot` transaction to revert and prevent all rewards from being distributed, even to valid relays. Consequently, a single misbehaving relay can indefinitely block the reward process for all networks, effectively resulting in a permanent denial of service until the affected relay is manually fixed or replaced.

Applicable Code:

This finding applies to the following lines of code in the _executeRewardSnapshot function:

Line#502

```
relay = LQGRewardsRelayInterface(networkRelayAddress);
  }
  // Transfer rewards
  if (rewardsLQG > 0) {
      // LQG rewards are withdrawn from the vault
      lqgVault.withdrawToken(address(relay), LQGContract, rewardsLQG);
```

```
    }
    if (rewardsETH > 0) {
        // ETH rewards are withdrawn from the smoothing pool
        lqgSmoothingPool.withdrawEther(address(relay), rewardsETH);
    }
```

Recommendation:

We recommend decoupling the reward dispatch logic so that each relay's transfer and execution are processed independently within separate transactions. Additionally, all external calls to relay contracts could be wrapped in `try-catch` blocks to prevent single points of failure. This layered approach increases resilience and ensures that a faulty or malicious relay cannot block the entire reward distribution process.

H02. Missing Reentrancy Protection May Allow Double Execution or Inconsistent State

Severity	High
Impact	High
Likelihood	Medium
Status	**Resolved**

Description:

The `_executeRewardSnapshot` function performs critical state updates—such as incrementing the reward index and updating claim interval timestamps—before making external calls to the vault, smoothing pool, and relay contracts. These external calls interact with upgradeable or potentially malicious third-party contracts. If any of them re-enter the `submitRewardSnapshot` or `executeRewardSnapshot` functions directly or indirectly, it may lead to unintended side effects, including reward index manipulation, double execution, or storage inconsistencies.

This pattern breaks the recommended "checks-effects-interactions" security principle, increasing the risk of reentrancy-related vulnerabilities. Because reward state changes are committed before the external transfers are complete, the system becomes vulnerable to denial-of-service scenarios or corrupted reward distribution records—especially if re-entry occurs mid-snapshot.

Applicable Code:

This finding applies to the _executeRewardSnapshot function:

Line#498

```
function _executeRewardSnapshot(RewardSubmission calldata _submission)
...
  incrementRewardIndex();
  ...
  lqgVault.withdrawToken(address(relay), ...);
  lqgSmoothingPool.withdrawEther(address(relay), ...);
  relay.relayRewards(...);
… }
```

Recommendation:

To prevent reentrancy risks, the contract should inherit from OpenZeppelin’s `ReentrancyGuard` and apply the `nonReentrant` modifier to `executeRewardSnapshot,` `executeRewardSnapshot` and `_executeRewardSnapshot`. Additionally, where practical, all state mutations should be moved after external calls to follow the effects-interactions pattern. Alternatively, sensitive state values can be cached in local variables before interacting with untrusted contracts to reduce the surface for manipulation.

H03. Lack of Caller Restriction on executeRewardSnapshot Allows Untrusted Execution and Front-Running

Severity	High
Impact	High
Likelihood	High
Status	**Resolved**

Description:

The `executeRewardSnapshot` function is declared as `external` and is not restricted to trusted nodes or any specific executor role. This means any externally owned account or contract can invoke it, regardless of whether they participated in the reward submission or are authorized to finalize the snapshot. As a result, untrusted actors can front-run intended submitters, race to claim execution credit, or flood the transaction pool with redundant calls. This not only leads to unnecessary gas competition but also makes it difficult to attribute which trusted node triggered the final reward distribution in the event logs.

By leaving this function open to all callers, the contract introduces an opportunity for griefing, operational inefficiencies, and potential confusion in automated execution pipelines. In a DAO-based rewards system that relies on accurate attribution and controlled execution, such openness undermines trust and traceability.

Applicable Code:

This finding applies to the executeRewardSnapshot function:

Line#383

```
 function executeRewardSnapshot(RewardSubmission calldata _submission)
        external
        override
        onlyLatestContract("lqgRewardsPool", address(this))
    {
    …
    }
```

Recommendation:

Restrict access to `executeRewardSnapshot` using `onlyTrustedNode(msg.sender)` to ensure only authorized participants can trigger finalization. Alternatively, store a hash or identifier (such as the merkle root) of the winning submission during the snapshot phase and require that the execution caller's submission matches it. This ensures execution flows from legitimate, consensus-approved actors and discourages unnecessary front-running.

Medium Severity Issues

M01. External Balance Reads in Reward Validation May Be Exploitable via Flash Loan Manipulation

Severity	Medium
Impact	High
Likelihood	Medium
Status	**Resolved**

Description:

The `submitRewardSnapshot` function validates the total submitted rewards by comparing them to the current on-chain balances of LQG and ETH available to the contract. These balances are retrieved using `getPendingLQGRewards` and `getPendingETHRewards`, which internally rely on external contract state—specifically the vault's token balances and the ETH balance of the smoothing pool contract. While this setup provides real-time accuracy, it also introduces a critical dependency on externally visible balances that may be influenced by third-party actors.

Because these values are read directly from external contracts without temporal locking or balance caching, an attacker may exploit this logic through flash loan attacks. For example, by temporarily inflating the LQG or ETH balance via a token deposit or ETH transfer during the same transaction, an attacker could satisfy the reward limit checks and manipulate the reward snapshot logic. If reentrant calls or front-running scenarios are also possible in the wider system, these vulnerabilities could lead to unauthorized or excessive reward distributions.

Applicable Code:

This finding applies to the getLQGBalance function:

Line#65

```
   function getLQGBalance() public view override returns (uint256) {
        // Get the vault contract instance
        LQGVaultInterface lqgVault =
            LQGVaultInterface(getContractAddress("lqgVault"));
        // Check contract LQG balance
        return lqgVault.balanceOfToken(
            "lqgRewardsPool", IERC20(getContractAddress("lqgTokenLQG"))
        );
    }

 function getPendingLQGRewards() public view override returns (uint256) {
        LQGTokenLQGInterface LQGContract =
            LQGTokenLQGInterface(getContractAddress("lqgTokenLQG"));
        uint256 pendingInflation = LQGContract.inflationCalculate();
        // Any inflation that has accrued so far plus any amount that would be
minted if we called it now
        return getLQGBalance() + pendingInflation;
    }



function submitRewardSnapshot(RewardSubmission calldata _submission)
....
       uint256 totalRewardsLQG = _submission.treasuryLQG;

            for (uint256 i = 0; i < _submission.nodeLQG.length; ++i) {
                totalRewardsLQG = totalRewardsLQG + _submission.nodeLQG[i];
            }
            for (uint256 i = 0; i < _submission.trustedNodeLQG.length; ++i) {
                totalRewardsLQG =
                    totalRewardsLQG + _submission.trustedNodeLQG[i];
            }
            require(
                totalRewardsLQG <= getPendingLQGRewards(), "Invalid LQG rewards"
            );
… }
```

Recommendation:

Ensure that flash loan attacks or transient balance manipulations are mitigated by enforcing stricter validation logic around token and ETH balances. This could include implementing snapshot-based accounting, using internal accounting variables instead of raw balances, or introducing minimum delay windows before accepting balances as valid. Additionally, reentrancy protection and isolation of sensitive flows can help minimize the risk of balance-related manipulation during snapshot submission.

M02. Integer Truncation and Missing Division Check May Cause Unexpected Consensus Behavior

Severity	Medium
Impact	Medium
Likelihood	Medium
Status	**Resolved**

Description:

The contract calculates whether consensus has been reached using the formula `(calcBase * submissionCount) / memberCount >= threshold`, where `calcBase` represents a fixed scaling factor and `threshold` defines the minimum ratio required for consensus. However, since Solidity performs integer division by rounding down, this calculation can underestimate the true ratio. For example, a `submissionCount` of 3 out of 5 members (60%) will be truncated below a two-thirds (66%) threshold, causing consensus to fail even when it should logically pass. This may unintentionally raise the bar for valid submissions and disrupt the intended reward distribution flow.

Additionally, the denominator `memberCount` is derived from the DAO's trusted node set. If, for any reason, the member count is ever set to zero, whether due to a misconfiguration or during contract initialization, this would result in a division-by-zero error and cause the transaction to revert. This scenario must be guarded explicitly to ensure system robustness.

Applicable Code:

This finding applies to the submitRewardSnapshot function:

Line#369

```
       if (
            (calcBase * submissionCount) / lqgDAONodeTrusted.getMemberCount()
                >= lqgDAOProtocolSettingsNetwork.getNodeConsensusThreshold()
        ) {
            _executeRewardSnapshot(_submission);
        }
```

Recommendation:

To avoid unexpected behavior, the consensus formula should incorporate rounding safeguards that more accurately reflect fractional comparisons. This can be done by rearranging the inequality to avoid division or by using ceiling-style logic. Furthermore, a check should be added to ensure that `memberCount` is greater than zero before performing the division, thereby preventing possible division-by-zero errors and unintended reverts.

M03. Partial State Update Before External Calls

Severity	Medium
Impact	Medium
Likelihood	Medium
Status	**Resolved**

Description:

In `_executeRewardSnapshot()`, protocol state (such as incrementing the reward index and updating interval timestamps) is updated before external relay distributions are attempted through `_distributeToRelay()`. If any relay distribution fails, the snapshot is still marked as executed, leaving some allocations undistributed while preventing further retries. This creates the risk of rewards becoming permanently stuck and introduces accounting inconsistencies across relays, as the protocol cannot distinguish between successful and failed distributions.

Applicable Code:

This finding applies to the _executeRewardSnapshot and _distributeToRelay functions:

Line#224,264

```
  function _executeRewardSnapshot(RewardSubmission calldata _submission) private {
        …
        incrementRewardIndex();
        …
            _distributeToRelay(
                i,
                networkRelayAddress,
                rewardsLQG,
                rewardsETH,
                _submission.rewardIndex,
                _submission.merkleRoot
            );
        }
    }

 function _distributeToRelay(
        uint256 networkIndex,
        address relayAddress,
        uint256 rewardsLQG,
        uint256 rewardsETH,
        uint256 rewardIndex,
        bytes32 merkleRoot
    ) private {
        LQGVaultInterface lqgVault =
LQGVaultInterface(getContractAddress("lqgVault"));
        LQGTokenInterface lqgTokenContract =
LQGTokenInterface(getContractAddress("lqgToken"));
        LQGSmoothingPoolInterface lqgSmoothingPool =
LQGSmoothingPoolInterface(getContractAddress("lqgSmoothingPool"));
        LQGRewardsRelayInterface relay = LQGRewardsRelayInterface(relayAddress);
        try this._safeTransferRewards(relayAddress, rewardsLQG, rewardsETH) {
            try relay.relayRewards(rewardIndex, merkleRoot, rewardsLQG, rewardsETH)
{
                // Success
            } catch {
                emit RelayRewardsFailure(rewardIndex, networkIndex, rewardsLQG,
rewardsETH);
            }
        } catch {
            emit RelayRewardsFailure(rewardIndex, networkIndex, rewardsLQG,
rewardsETH);
        }
    }
```

Recommendation:

Instead of finalising the snapshot unconditionally, the contract should explicitly track per-relay distribution outcomes and allow targeted retries for failures. A practical approach is to mark any failed distribution in a mapping during execution, while still progressing overall state. Afterwards, expose a retry function that can safely re-attempt only the failed distributions and clear their flags once they succeed. This ensures rewards are eventually delivered, preserves execution progress, and avoids permanently stuck allocations due to transient relay errors.

Low Severity Issues

L01. Unbounded Loop in View Function Can Revert on Large Arrays and Disrupt Off-Chain Services

Severity	Low
Impact	Low
Likelihood	Low
Status	**Resolved**

Description:

The `getClaimingContractsPerc` view function processes an external input array `_claimingContracts` without enforcing any upper bound on its length. For each element, it calls into another contract to fetch the corresponding reward percentage. While the function does not modify state, it performs a loop that scales linearly with the array size, which can lead to excessive gas consumption. If a UI or off-chain service unintentionally or maliciously submits a large array, the call may exceed the gas limit and revert, potentially causing service disruptions or broken dashboards.

Because view functions are often used in automated scripts, frontends, or monitoring tools, unbounded iterations like this one can degrade user experience and create denial-of-service vectors—even without touching on-chain logic. Although not exploitable in a traditional sense, this is a notable design risk in user-facing infrastructure.

Applicable Code:

This finding applies to the getClaimingContractsPerc function:

Line#173

```
    function getClaimingContractsPerc(string[] memory _claimingContracts)
        external
        view
        override
        returns (uint256[] memory)
    {
        …
        // Get the % amount allocated to this claim contract
        uint256[] memory percentages = new uint256[](_claimingContracts.length);
        for (uint256 i = 0; i < _claimingContracts.length; ++i) {
            percentages[i] =
                daoSettingsRewards.getRewardsClaimerPerc(_claimingContracts[i]);
        }
        return percentages;
    }
```

Recommendation:

Introduce a length cap on the _claimingContracts array to prevent excessively large requests. Alternatively, refactor the function to include pagination-style parameters, such as start and count, allowing off-chain clients to fetch reward percentages in manageable chunks. This ensures stable performance and avoids unintended reverts in view calls.

L02. Missing Validation of Merkle Root or Tree Data May Allow Submission of Invalid Rewards

Severity	Low
Impact	Low
Likelihood	Low
Status	**Resolved**

Description:

The `submitRewardSnapshot` function accepts a full `RewardSubmission` that includes a `merkleRoot` and a `merkleTreeCID` string. These values are crucial because the root represents the final snapshot used by relays to distribute rewards to recipients off-chain. However, the contract does not validate the structure, format, or consistency of the `merkleRoot`, nor does it confirm that the `merkleTreeCID` is a valid or expected value.

Since there is no check that ties the merkle root to the actual reward data being submitted, a malicious or misconfigured trusted node could submit a reward snapshot with a mismatched or incorrect root. While relays may verify the root off-chain, the on-chain logic trusts it blindly. This weakens the trust assumptions of the entire reward distribution process and could lead to incorrect rewards being propagated and accepted without challenge.

Applicable Code:

This finding applies to the _executeRewardSnapshot function:

Line#410

```
     function _executeRewardSnapshot(RewardSubmission calldata _submission)
        private
    {
        ...
            // Call into relay contract to handle distribution of rewards
            try relay.relayRewards(
                _submission.rewardIndex,
                _submission.merkleRoot,
                rewardsLQG,
                rewardsETH
            ) {
                ...
            }
```

Recommendation:

Introduce a validation step (on-chain or off-chain via signed metadata) that ensures the submitted `merkleRoot` and `merkleTreeCID` are consistent with the reward data. Alternatively, enforce format checks or known-prefix validation on the CID to avoid accepting arbitrary or malformed strings. This will help ensure the integrity and traceability of the reward distribution snapshot.

L03. Missing Input Validation in `executeRewardSnapshot`

Severity	Low
Impact	Low
Likelihood	Low
Status	**Resolved**

Description:

The `submitRewardSnapshot()` function enforces an upper bound on submission size via `require(_submission.node.length <= MAX_NETWORK_COUNT)`, but `executeRewardSnapshot()` lacks the same guard. This asymmetry allows oversized `_submission.node` arrays to reach execution, increasing gas usage, risking out-of-gas/DoS, and creating inconsistent assumptions between submission and execution paths.

Applicable Code:

This finding applies to the `submitRewardSnapshot and executeRewardSnapshot()` functions.

Line#158

```
    function submitRewardSnapshot(RewardSubmission calldata _submission)
        override
        external
        nonReentrant
        onlyLatestContract("lqgRewardsPool", address(this))
        onlyTrustedNode(msg.sender)
    {
        …
        // Prevent bloated submissions that could cause gas limit issues
        require(_submission.node.length <= MAX_NETWORK_COUNT, "Too many networks");
```

Recommendation:

To ensure consistency and prevent oversized submissions from bypassing safeguards, the contract should enforce the same validation in `executeRewardSnapshot()` as in `submitRewardSnapshot()`. Specifically, add a check requiring `_submission.node.length <= MAX_NETWORK_COUNT` at the start of `executeRewardSnapshot()` (or within a shared internal helper) so that both submission and execution paths apply identical bounds. This alignment reduces the risk of excessive gas usage, potential out-of-gas failures, and execution inconsistencies caused by unchecked array sizes.

■ ■ Informational Severity Issues

I01. Potential Gas Optimization by Merging Redundant Loops in Reward Aggregation

Severity	Info
Impact	Low
Likelihood	Low
Status	**Resolved**

Description:

The contract calculates total reward amounts by separately iterating over three arrays: `nodeLQG`, `trustedNodeLQG`, and `nodeETH`. Each loop adds the respective reward amounts into cumulative totals before validating them against available balances. However, these arrays are guaranteed to have the same length as part of the reward submission validation. This means that a single loop could be used to process all three reward sources in parallel, avoiding repeated iteration logic.

By using a single `for` loop to sum values from all three arrays simultaneously, the contract would reduce gas consumption during snapshot submissions. Since these computations occur in a hot path that is called frequently, especially during consensus rounds, optimizing this segment has measurable impact on gas efficiency without affecting functionality or readability.

Applicable Code:

This finding applies to the submitRewardSnapshot function:

Line#280

```
  for (uint256 i = 0; i < _submission.nodeLQG.length; ++i) {
                totalRewardsLQG = totalRewardsLQG + _submission.nodeLQG[i];
            }
            for (uint256 i = 0; i < _submission.trustedNodeLQG.length; ++i) {
                totalRewardsLQG =
                    totalRewardsLQG + _submission.trustedNodeLQG[i];
            }
            require(
                totalRewardsLQG <= getPendingLQGRewards(), "Invalid LQG rewards"
            );
        }
        // Calculate ETH reward total and validate
        {
            // Scope to prevent stack too deep
            uint256 totalRewardsETH = 0;
            for (uint256 i = 0; i < _submission.nodeETH.length; ++i) {
```

Recommendation:

Refactor the logic to combine the three separate loops into one, aggregating `nodeLQG`, `trustedNodeLQG`, and `nodeETH` concurrently. This improves runtime efficiency and reduces gas costs, especially when handling reward submissions involving large numbers of networks.

I02. Unnecessary Hashing of Full Struct Increases Gas Costs

Severity	Info
Impact	Low
Likelihood	Low
Status	**Resolved**

Description:

The contract uses the full `RewardSubmission` struct as input to `keccak256` when generating keys for storage mappings like `submissionCountKey` and `nodeSubmissionKey`. However, this struct includes several dynamic and heavy fields, such as three arrays and a string, which are not all necessary to uniquely identify a submission. Most of the logic only relies on a few core fields, like `rewardIndex` and `merkleRoot`.

By hashing the entire struct, the contract pays extra gas for processing data that isn't actually used in decision-making. This adds unnecessary cost, especially in scenarios with large reward submissions across many networks.

Applicable Code:

This finding applies to the submitRewardSnapshot and executeRewardSnapshot functions:

Line#240,349

```
function submitRewardSnapshot(RewardSubmission calldata _submission)
...
  bytes32 nodeSubmissionKey = keccak256(
                abi.encode(
                    "rewards.snapshot.submitted.node.key",
                    msg.sender,
                    _submission
                )
            );
...
```

```
}

    function executeRewardSnapshot(RewardSubmission calldata _submission)
        external
        override
        onlyLatestContract("lqgRewardsPool", address(this))
    {
        ...
        // Get submission count
        bytes32 submissionCountKey = keccak256(
            abi.encode("rewards.snapshot.submitted.count", _submission)
        );
```

Recommendation:

Hash only the fields that are essential for identifying a reward submission, such as `rewardIndex`, `intervalsPassed`, and `merkleRoot`. Avoid including full arrays and strings in the hash if they aren't strictly needed. This will reduce calldata size, improve gas efficiency, and make the contract more performant during reward processing.

I03. Acceptance of Historical Submissions That Will Never Execute

Severity	Info
Impact	Low
Likelihood	Low
Status	**Resolved**

Description:

The `submitRewardSnapshot()` accepts submissions where `_submission.rewardIndex <= currentRewardIndex`, but execution only occurs when it equals the current index. As a result, trusted nodes can submit snapshots for past periods that are already finalised. If historical storage is not explicitly required, this behaviour invites unnecessary gas spend, state growth, and potential griefing via redundant snapshots that can never execute.

Applicable Code:

This finding applies to the submitRewardSnapshot function:

Line#154,184

```
   function submitRewardSnapshot(RewardSubmission calldata _submission)
        override
        external
        nonReentrant
        onlyLatestContract("lqgRewardsPool", address(this))
        onlyTrustedNode(msg.sender)
    {
       ….
        uint256 rewardIndex = getRewardIndex();
        require(_submission.rewardIndex <= rewardIndex, "Can only submit snapshot for
periods up to next");
```

```
   ….
        // Return if already executed
         if (_submission.rewardIndex != rewardIndex) {
             return;
         }
```

Recommendation:

To prevent unnecessary gas consumption and storage bloat from outdated submissions, the function should strictly enforce that `_submission.rewardIndex == currentRewardIndex` at the time of submission, unless there is an explicit protocol requirement to archive historical data. This ensures only relevant snapshots are accepted and eligible for execution, maintaining lean state management and reducing the risk of spam or redundant submissions. If historical tracking is ever needed, it should be implemented as a separate, clearly documented feature rather than being implicitly supported by the current submission flow.

Disclaimers

Hakflow Disclaimer

The extension reviewed in this audit has been analyzed based on the best industry practices available at the time of this report. The evaluation includes assessments of the configurations, deployment strategies and functionalities to perform intended functions.

This audit does not provide any warranties regarding the security of the systems or code assessed. It should not be considered as an exhaustive evaluation of the security, reliability, or error-free operation, nor should it be seen as an endorsement of the overall safety.

While we have endeavored to conduct a thorough analysis and provide accurate findings in this report, it is important to recognize that this report should not be the sole resource relied upon. We strongly recommend engaging in additional independent audits and continuous security monitoring to ensure the ongoing security and integrity of the environment.

Technical Disclaimer

Systems and their underlying code are complex and involve multiple layers of technology, including software, and network components. Each of these components can have inherent vulnerabilities that could potentially be exploited. Therefore, despite the thoroughness of the audit, we cannot guarantee absolute security of the audited systems or code. Continuous vigilance, regular updates, and a proactive security strategy are crucial to maintaining the security and integrity of any environment.

Appendix 1. Severity Definitions

In assessing the project's security posture, we employ a risk-based approach that evaluates the potential impact and exploitability of identified vulnerabilities. This method utilizes a matrix of impact and likelihood, a standard tool in risk management, to aid in the assessment and prioritization of risks.

Risk Levels

Critical: Critical vulnerabilities in an environment are those that are easily exploitable and can lead to severe consequences such as complete service disruption or compliance violations. These vulnerabilities generally allow unauthorized access or control over resources, leading to a direct impact on business operations and security.

High: High vulnerabilities are more challenging to exploit and might require specific conditions to be met. While their impact may be somewhat less devastating than critical vulnerabilities, they can still result in considerable damage such as data leaks, partial service disruption, or compliance issues.

Medium: Medium vulnerabilities include issues that may lead to certain operational inefficiencies or security risks that are more difficult to exploit and usually do not result in direct asset loss. These might include minor compliance deviations, moderate data exposure risks, or inefficiencies in resource utilization. Such vulnerabilities often require specific conditions or sequences of actions to be exploited.

Low: Low risk vulnerabilities pose minimal risk to system operation or security. They are typically related to non-critical inefficiencies, minor coding practices, or low-level configuration issues that have a very limited chance of being exploited. While these findings may not immediately affect system performance or security, addressing them can still contribute to overall improvement.

Informational: Informational vulnerabilities provide helpful insights or observations without indicating any immediate or potential risk to system functionality, security, or performance. They are typically suggestions, or areas where further optimization could be beneficial.

Impact Levels

High Impact: Risks with a high impact are associated with severe consequences, such as significant financial losses, substantial reputational damage, or major operational disruptions. High impact issues in environments typically involve breaches leading to data loss, denial of service attacks disrupting service availability, or security incidents that affect regulatory compliance.

Medium Impact: Risks with medium impact could lead to moderate financial losses or reputational harm. These may include less severe breaches that result in limited data exposure, minor compliance issues, or performance degradations that affect user experience but do not completely halt operations.

Low Impact: Risks with low impact are unlikely to cause direct financial losses or significant operational disruption. These issues generally pertain to inefficiencies or minor non-compliance with best practices that may affect the system's optimal functioning or minor security best practices deviations which do not directly compromise security but could lead to vulnerabilities if not addressed.

Likelihood Levels

High Likelihood: Risks with a high likelihood are those that are expected to occur frequently or are very likely to occur due to existing vulnerabilities or weaknesses in the system configuration or security measures. This category also includes risks from common attack vectors that target prevalent vulnerabilities across similar platforms.

Medium Likelihood: Risks with a medium likelihood are possible but not as probable as those in the high likelihood category. These might be due to less critical vulnerabilities that require specific conditions to be exploited or are known to a smaller group of potential attackers, representing a moderate level of threat.

Low Likelihood: Risks of low likelihood are those that are unlikely to occur and may require highly specific conditions or complex strategies to exploit. These risks typically stem from obscure or theoretical vulnerabilities that are not generally targeted by attackers due to the high effort and low success rate associated.

Risk Level	High Impact	Medium Impact	Low Impact
High Likelihood	**Critical**	High	Medium
Medium Likelihood	**High**	Medium	Low
Low Likelihood	**Medium**	Low	Low

Document

Name	Smart Contract Audit Report for Liquidity Gate
Approved By	George Kougias | BDM at Hakflow
Audited By	Dimitris Pallis | CEO at Hakflow
Website	https://hakflow.com
Changelog	01.08.2025 – Preliminary Report 02.08.2025 - Final Report

Hakflow is at the forefront of the cybersecurity industry, forging impactful partnerships with leading entities such as *Plume Network* and *VaasBlock* in the blockchain space and *NSFOCUS* in traditional cybersecurity. Our commitment to excellence is further demonstrated by our active participation in renowned security conferences, including *ETHSofia* and *GISEC Dubai*.

Client Testimonial - Rizwan Zareef, CEO of Darkguard

"*Through our collaboration with Hakflow, our clients have proactively strengthened their cybersecurity defenses and are now better prepared for the evolving threat landscape*"

Industry Certified

OffSec OSCP · OFFENSIVE SECURITY OSWE · aws certified Security SPECIALTY · aws certified Cloud Practitioner FOUNDATIONAL