QuillAudits

# AUDIT REPORT

June , 2025

For

PRIME NUMBERS

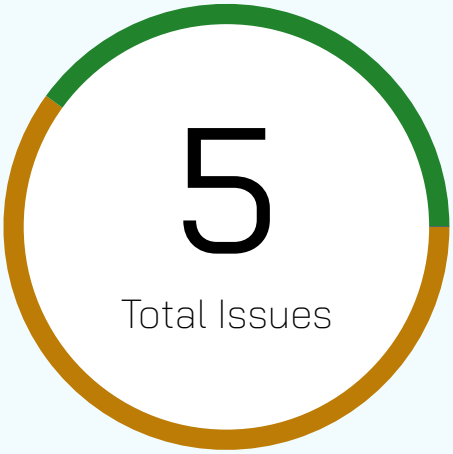
# Table of Content

# Executive Summary

| | |
|---|---|
| **Project name** | Prime Number |
| **Project URL** | https://primenumbers.xyz/ |
| **Overview** | The RewardDistributionController is a cross-chain reward distribution system built on LayerZero's OApp V2 framework. It facilitates multi-chain reward tracking, calculation, and vesting of the protocol's native token (PRFI) based on a user's holdings across registered tokens. The contract operates in two modes: main chain (where reward state is centralized) and sidechains (which communicate updates back to the main chain via LayerZero messaging). |
| **Audit Scope** | The scope of this Audit was to analyze the Prime Number Smart Contracts for quality, security, and correctness. |
| **Source Code link** | contracts/prime/staking/rewards/RewardDestributionController.sol<br>Diff audit with Radiant |
| **Contracts in Scope** | Reward Distribution Controller |
| **Branch** | Main |
| **Commit Hash** | 604126dc8662de60eafc87b77436419f60617106 |
| **Language** | Solidity |
| **Blockchain** | EVM |

| | |
|---|---|
| **Method** | Manual Analysis, Functional Testing, Automated Testing |
| **Review 1** | 29th May 2025 - 7th June 2025 |
| **Updated Code Received** | 13th June 2025 |
| **Review 2** | 17th June 2025 - 19th June 2025 |
| **Fixed In** | 06863e8d99a90aef8957da6117a86340daa9618b |

# Number of Issues per Severity




| | |
|---|---|
| High | 0 (0.00%) |
| Medium | 3 (60.00%) |
| Low | 2 (40.00%) |
| Informational | 0 (0.00%) |

Severity

| Issues | High | Medium | Low | Informational |
|---|---|---|---|---|
| Open | 0 | 0 | 0 | 0 |
| Resolved | 0 | **3** | **1** | 0 |
| Acknowledged | 0 | 0 | **1** | 0 |
| Partially Resolved | 0 | 0 | 0 | 0 |

# Checked Vulnerabilities

- ✔ Access Management
- ✔ Arbitrary write to storage
- ✔ Centralization of control
- ✔ Ether theft
- ✔ Improper or missing events
- ✔ Logical issues and flaws
- ✔ Arithmetic Computations Correctness
- ✔ Race conditions/front running
- ✔ SWC Registry
- ✔ Re-entrancy
- ✔ Timestamp Dependence
- ✔ Gas Limit and Loops
- ✔ Exception Disorder
- ✔ Gasless Send
- ✔ Use of tx.origin
- ✔ Malicious libraries
- ✔ Compiler version not fixed
- ✔ Address hardcoded
- ✔ Divide before multiply
- ✔ Integer overflow/underflow
- ✔ ERC's conformance
- ✔ Dangerous strict equalities
- ✔ Tautology or contradiction
- ✔ Return values of low-level calls
- ✔ Missing Zero Address Validation
- ✔ Private modifier
- ✔ Revert/require functions
- ✔ Multiple Sends
- ✔ Using suicide
- ✔ Using delegatecall
- ✔ Upgradeable safety
- ✔ Using throw

- [x] Using inline assembly
- [x] Unsafe type inference
- [x] Style guide violation
- [x] Implicit visibility level

# Techniques and Methods

Throughout the audit of smart contracts, care was taken to ensure:

- The overall quality of code.
- Use of best practices.
- Code documentation and comments, match logic and expected behavior.
- Token distribution and calculations are as per the intended behavior mentioned in the whitepaper.
- Implementation of ERC standards.
- Efficient use of gas.
- Code is safe from re-entrancy and other vulnerabilities.

**The following techniques, methods, and tools were used to review all the smart contracts.**

Structural Analysis

In this step, we have analyzed the design patterns and structure of smart contracts. A thorough check was done to ensure the smart contract is structured in a way that will not result in future problems.

Static Analysis

A static Analysis of Smart Contracts was done to identify contract vulnerabilities. In this step, a series of automated tools are used to test the security of smart contracts.

### Code Review / Manual Analysis

Manual Analysis or review of code was done to identify new vulnerabilities or verify the vulnerabilities found during the static analysis. Contracts were completely manually analyzed, their logic was checked and compared with the one described in the whitepaper. Besides, the results of the automated analysis were manually verified.

### Gas Consumption

In this step, we have checked the behavior of smart contracts in production. Checks were done to know how much gas gets consumed and the possibilities of optimization of code to reduce gas consumption.

### Tools And Platforms Used For Audit

Remix IDE, Foundry, Solhint, Mythril, Slither, Solidity statistical analysis.

# Types of Severity

Every issue in this report has been assigned to a severity level. There are four levels of severity, and each of them has been explained below

### High Severity Issues

A high severity issue or vulnerability means that your smart contract can be exploited. Issues on this level are critical to the smart contract's performance or functionality, and we recommend these issues be fixed before moving to a live environment.

### Medium Severity Issues

The issues marked as medium severity usually arise because of errors and deficiencies in the smart contract code. Issues on this level could potentially bring problems, and they should still be fixed.

### Low Severity Issues

Low-level severity issues can cause minor impact and are just warnings that can remain unfixed for now. It would be better to fix these issues at some point in the future.

### Informational Issues

These are four severity issues that indicate an improvement request, a general question, a cosmetic or documentation error, or a request for information. There is low-to-no impact.

# Types of Issues

**Open**

Security vulnerabilities identified that must be resolved and are currently unresolved.

**Resolved**

Security vulnerabilities identified that must be resolved and are currently unresolved.

**Acknowledged**

Vulnerabilities which have been acknowledged but are yet to be resolved.

**Partially Resolved**

Considerable efforts have been invested to reduce the risk/ impact of the security issue, but are not completely resolved.

# Medium Severity Issues

## LayerZero fee refunds misdirected to contract address.

Resolved

### Path

contracts/prime/staking/rewards/RewardDestributionController.sol

### Description

In the claimAll() function, when executing a LayerZero cross-chain message, the refund address passed to **`_send()`** is set as address(this). This address receives any excess
Ether left over from the nativeFee used for the messaging. However, the actual cost of the message is incurred by the sender (**`msg.sender`**), not the contract. The claimAll function accepts msg.value equal to / greater than the fee quoted by LayerZero to cover LayerZero's messaging costs. It forwards them to the endpoint contract, from which any unused value is refunded to the refundAddress specified.

However, these refunds don't work as expected as the refund address is always specified as address(this), which is the contract itself in the context of the **`RewardDestributionController.sol`** con-tract. Thus, any refunds received from LayerZero'sn endpoint are locked indefinitely in the Reward-DestributionController.sol contract.

### Code Affected

```
_send(_mainChain, payload, options, fee, payable(address(this)));
```

### Impact

Users who overpay for cross-chain messaging may lose excess ETH forever. Over time, this can ac-cumulate significant trapped funds and lead to poor user experience or financial loss.

### Recommendation

1. Make sure msg.value is strictly equal to the quoted fee, making sure no refunds are possible.
2. Forwarding a user-specified refund address to LayerZero's endpoint to process refunds properly.

```
_send(_mainChain, payload, options, fee, payable(msg.sender));
```

## Refund Handling in _lzSend is Broken and Can Permanently Lock ETH

Resolved

### Path

contracts/prime/staking/rewards/RewardDestributionController.sol

### Description

The contract attempts to serve as the refund address in LayerZero messaging calls, but this behavior introduces multiple flaws:

1. Contract Cannot Receive Refunds
The contract lacks a **`receive()`** or **`fallback()`** function, which means it cannot accept native ETH via plain transfers. Since the **`_lzSend()`** function designates address(this) as the refund address, any excess ETH intended as a refund by LayerZero will cause a revert when sent back.

2. ETH Can Be Trapped Forever
In claimAll(), the call chain: claimAll() → leads to this code:

```
require(address(this).balance >=_fee.nativeFee, InsufficientFee());
endpoint.send{ value:_fee.nativeFee }(...)
```

The logic allows the user to send more than _fee.nativeFee, but only that exact amount is forwarded. For example: User sends 1 ETH and the _fee.nativeFee is set to 0.8 ETH which means only 0.8 ETH is forwarded to LayerZero endpoint and remaining 0.2 ETH stays in contract with no way to refund or withdraw it.

### Impact

1. Funds trapped: Excess ETH sent by users is stuck permanently with no refund path.

2. Reverts: Any refund attempt from LayerZero to address(this) will fail unless a receive function is added.

### Recommendation

Forward the entire msg.value to LayerZero endpoint

```
uint256 balance = address(this).balance;
endpoint.send{ value: balance }(...)
```

## Stale Price Handling Missing in _getTokenValueInUSD

Resolved

### Path

contracts/prime/staking/rewards/RewardDestributionController.sol

### Description

The `_getTokenValueInUSD()` function retrieves token prices from Chainlink using `latestRoundData()` but does not validate the freshness or integrity of the returned price. Specifically, it does not check if:

- The price is non-zero
- The update timestamp (updatedAt) is recent

This creates an assumption that Chainlink is always live and updating, which is dangerous in production deployments.

### Impact

If Chainlink oracles are paused, delayed, or manipulated, the protocol will continue using stale or zero prices. This affects all reward-related logic:

- Users may receive inaccurate rewards

- Attackers may exploit outdated prices to manipulate PRFI distribution

### Recommendation

Add validation for

```
(uint80 roundId, int256 price, , uint256 updatedAt, uint80 answeredInRound) =
aggregator.latestRoundData();

require(price > 0, "Invalid price");
require(updatedAt >= block.timestamp - MAX_STALE_PERIOD, "Stale price");
require(answeredInRound >= roundId, "Incomplete round");
```

# Low Severity Issues

## Missing Event Emission on Important State Changes

Resolved

### Path

contracts/prime/staking/rewards/RewardDestributionController.sol

### Description

According to the LayerZero integration checklist, when a contract requests **`msg.value`** through the OptionsBuilder in a cross-chain message, the expected value must be explicitly encoded into the message payload and validated upon receipt.

This enforcement is missing in the current **`_lzReceive()`** implementation, the actual msg.value required by the receiver is not encoded in the message nor validated. This opens a subtle but critical attack surface.

Without enforcing **`msg.value`**, a malicious or misconfigured executor may supply less ETH than ex-pected, causing incomplete or unintended execution

### Recommendation

Emit clearly named events in each of the above functions. Example:

```
event PrfiTokenUpdated(address oldToken, address newToken);
event MfdUpdated(address oldMfd, address newMfd);
event RewardMinterSet(address rewardMinter);
event PoolConfiguratorSet(address poolConfigurator);
event TokenAggregatorUpdated(Token token, address aggregator);
event TokenAggregatorRemoved(Token token);
event MainChainUpdated(uint32 oldChain, uint32 newChain);
event PrfiPerSecondChanged(uint256 oldRate, uint256 newRate);
```

## Missing Enforcement of msg.value in _ lzReceive AllowsUnderfunded Execution

Acknowledged

### Path

contracts/prime/staking/rewards/RewardDestributionController.sol

### Description

Throughout the codebase, several setter and configuration functions update critical protocol state, such as addresses, reward configurations, oracle sources, and system parameters without emitting corresponding events. This limits transparency, breaks compatibility with o-chain indexers, and makes it dicult to audit or trace administrative actions after deployment.

A well-designed contract should emit events for all critical changes, especially those that influence reward logic, oracle feeds, or core system addresses.

Code Affected:

```
// In _lzReceive():
function _lzReceive(...) internal override {
// No check for received msg.value or enforcement of minimum ETH
}
```

### Recommendation

Encode the expected msg.value in your payload when constructing the message. In _lzReceive(), compare the actual **`msg.value`** received against the expected value:

```
require(msg.value >= expectedValue, "Underfunded execution");
```

# Automated Tests

No major issues were found. Some false positive errors were reported by the tools. All the other issues have been categorized above according to their level of severity.

# Closing Summary

In this report, we have considered the security of Prime Number. We performed our audit according to the procedure described above.

3 issue of medium severity, 2 low issues of low severity were found. Prime Number Team acknowledged one and resolved rest of them .

# Disclaimer

At QuillAudits, we have spent years helping projects strengthen their smart contract security. However, security is not a one-time event—threats evolve, and so do attack vectors. Our audit provides a security assessment based on the best industry practices at the time of review, identifying known vulnerabilities in the received smart contract source code.

This report does not serve as a security guarantee, investment advice, or an endorsement of any platform. It reflects our findings based on the provided code at the time of analysis and may no longer be relevant after any modifications. The presence of an audit does not imply that the contract is free of vulnerabilities or fully secure.

While we have conducted a thorough review, security is an ongoing process. We strongly recommend multiple independent audits, continuous monitoring, and a public bug bounty program to enhance resilience against emerging threats.

Stay proactive. Stay secure.

# About QuillAudits

QuillAudits is a secure smart contracts audit platform designed by QuillHash Technologies. We are a team of dedicated blockchain security experts and smart contract auditors determined to ensure that Smart Contract-based Web3 projects can avail the latest and best security solutions to operate in a trustworthy and risk-free ecosystem

QuillAudits

| | |
|---|---|
| **7+**<br>Years of Expertise | **1M+**<br>Lines of Code Audited |
| **$30B+**<br>Secured in Digital Assets | **1400+**<br>Projects Secured |

Follow Our Journey

# AUDIT REPORT

June , 2025

For

PRIME NUMBERS

QuillAudits

Canada, India, Singapore, UAE, UK

www.quillaudits.com audits@quillaudits.com