QuillAudits

AUDIT REPORT

September 2025

For

PRIME NUMBERS

Table of Content

Executive Summary

Project Name	PrimeNumbers
Protocol Type	Decentralised Lending, Borrowing, & Rewards
Project URL	https://primenumbers.xyz/
Overview	Decentralised Lending, Borrowing, & Rewards – powered by Aavev2 + PRFI incentives. PrimeFi brings capital–efficient money markets, real–time rewards, and NFT–driven profit–sharing to EVM chains.
Audit Scope	The scope of this Audit was to analyze the PrimeNumbers Smart Contracts for quality, security, and correctness.
Source Code link	https://github.com/PrimeNumbersLabs/primefi-contracts/commit/d9282f3a10413ca9f93f999f1233d8e0f84d49c4
Branch	Main
Contracts in Scope	Prime Numbers is a fork of Aave v2 and changes are primarily related to the upgradeable contracts. 1. contracts/prime/staking/rewards/SidechainIncentivesController.sol 2. contracts/prime/staking/rewards/IncentivesControllerStorage.sol 3. contracts/prime/staking/rewards/MainchainIncentivesController.sol 4. contracts/prime/staking/rewards/IncentivesControllerDiamond.sol 5. contracts/prime/oracles/chainlink/ChainlinkMiddleware.sol 6. contracts/prime/oracles/chainlink/DataStreamConsumer.sol QuillAudits Team mainly focused on the six contracts mentioned above and the changes made, but at the same time we also looked into the rest of the codebase on Surface level to ensure there's no unintended impact on the business logic.
Commit Hash	d9282f3a10413ca9f93f999f1233d8e0f84d49c4
Language	Solidity

Blockchain	EVM
Method	Manual Analysis, Functional Testing, Automated Testing
Review 1	13th August 2025 - 29th August 2025
Updated Code Received	3rd September 2025
Review 2	3rd September 2025 - 10th September 2025
Fixed In	https://github.com/PrimeNumbersLabs/primefi-contracts/issues

Verify the Authenticity of Report on QuillAudits Leaderboard:

https://www.quillaudits.com/leaderboard

Number of Issues per Severity

Critical	1 (6.25%)
High	5 (31.25%)
Medium	1 (6.25%)
Low	2 (12.5%)
Informational	7 (43.75%)

Severity

Issues	Critical	High	Medium	Low	Informational
Open	0	1	0	0	0
Acknowledged	0	0	0	0	2
Partially Resolved	0	0	0	0	0
Resolved	1	4	1	2	5

Summary of Issues

Issue No.	Issue Title	Severity	Status
1	Missing rewards accumulation causes permanent loss upon user disqualification	Critical	Resolved
2	Excess native fee payment becomes permanently locked in contract	High	Resolved
3	Cache-based protocol value calculation enables reward manipulation	High	Resolved
4	Reward per token calculation enables reward extraction by manipulating the protocol value	High	Open
5	Claim bypasses pause and access control, enabling unauthorized reward claims	High	Resolved
6	Oracle price staleness checks are missing in _getAndUpdateTokenValueInUSD, risking reward manipulation	High	Resolved
7	Missing reward token transfer in notifyRewardAmount risks claim failures	Medium	Resolved
8	Lack of Ownable inheritance prevents secure ownership transfer	Low	Resolved

Issue No.	Issue Title	Severity	Status
9	Overwriting global totalSupply without chain context	Low	Resolved
10	Missing aggregator check in removeTokenAggregator allows invalid removals	Informational	Resolved
11	Lack of zero-value check when setting prfiPerSecond	Informational	Resolved
12	Lack of zero-value check when setting updateOraclePeriod	Informational	Resolved
13	Manual Setting of Token Decimals Instead of Dynamic Retrieval	Informational	Resolved
14	Inefficient token decimal normalization	Informational	Acknowledged
15	Unused internal function increases contract bytecode without providing functionality	Informational	Resolved
16	Usage of SafeMath Is Redundant in Solidity ⩾ 0.8.0	Informational	Acknowledged

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
- Using inline assembly
- Style guide violation
- Unsafe type inference
- 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 Static Analysis.

Types of Severity

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

Critical: Immediate and Catastrophic Impact

Critical issues are the ones that an attacker could exploit with relative ease, potentially leading to an immediate and complete loss of user funds, a total takeover of the protocol's functionality, or other catastrophic failures. Critical vulnerabilities are non-negotiable; they absolutely must be fixed.

High (H): Significant Risk of Major Loss or Compromise

High-severity issues represent serious weaknesses that could result in significant financial losses for users, major malfunctions within the protocol, or substantial compromise of its intended operations. While exploiting these vulnerabilities might require specific conditions to be met or a moderate level of technical skill, the potential damage is considerable. These findings are critical and should be addressed and resolved thoroughly before the contract is put into the Mainnet.

Medium (M): Potential for Moderate Harm Under Specific Circumstances

Medium-severity bugs are loopholes in the protocol that could lead to moderate financial losses or partial disruptions of the protocol's intended behavior. However, exploiting these vulnerabilities typically requires more specific and less common conditions to occur, and the overall impact is generally lower compared to high or critical issues. While not as immediately threatening, it's still highly recommended to address these findings to enhance the contract's robustness and prevent potential problems down the line.

Low (L): Minor Imperfections with Limited Repercussions

Low-severity issues are essentially minor imperfections in the smart contract that have a limited impact on user funds or the core functionality of the protocol. Exploiting these would usually require very specific and unlikely scenarios and would yield minimal gain for an attacker. While these findings don't pose an immediate threat, addressing them when feasible can contribute to a more polished and well-maintained codebase.

Informational (I): Opportunities for Improvement, Not Immediate Risks

Informational findings aren't security vulnerabilities in the traditional sense. Instead, they highlight areas related to the clarity and efficiency of the code, gas optimization, the quality of documentation, or adherence to best development practices. These findings don't represent any immediate risk to the security or functionality of the contract but offer valuable insights for improving its overall quality and maintainability. Addressing these is optional but often beneficial for long-term health and clarity.

Types of Issues

Open

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

Resolved

These are the issues identified in the initial audit and have been successfully fixed.

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.

Severity Matrix

Impact

Likelihood	High	Medium	Low
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

Impact

- **High** - leads to a significant material loss of assets in the protocol or significantly harms a group of users.
- **Medium** - only a small amount of funds can be lost (such as leakage of value) or a core functionality of the protocol is affected.
- **Low** - can lead to any kind of unexpected behavior with some of the protocol's functionalities that's not so critical.

Likelihood

- High - attack path is possible with reasonable assumptions that mimic on-chain conditions, and the cost of the attack is relatively low compared to the amount of funds that can be stolen or lost.
- Medium - only a conditionally incentivized attack vector, but still relatively likely.
- Low - has too many or too unlikely assumptions or requires a significant stake by the attacker with little or no incentive.

Critical Severity Issues

Missing rewards accumulation causes permanent loss upon user disqualification

Resolved

Path

contracts/prime/staking/rewards/MainchainIncentivesController.sol#L879

Function Name

`stopEmissionsFor`

Description

In contracts/prime/staking/rewards/MainchainIncentivesController.sol#L879, the stopEmissionsFor function disqualifies users by updating their _userRewardPerTokenPaid mapping and setting their eligibility to false. However, this function fails to capture the user's accumulated rewards in the $.rewards[_chainEid][_user] mapping before disqualification occurs. The function only updates the user's reward per token paid checkpoint without preserving their earned rewards up to that point.

The rewards system operates on a checkpoint mechanism where earned calculates rewards based on the difference between current rewardPerToken and the user's last checkpoint in _userRewardPerTokenPaid. When stopEmissionsFor updates this checkpoint without first storing accumulated rewards, it effectively resets the user's reward calculation baseline, causing all previously earned rewards to become permanently inaccessible. This differs from the correct implementation pattern seen in the updateReward modifier at lines 65-66, which properly captures earned rewards before updating the checkpoint.

Consequently, users who become ineligible through the disqualification process lose all rewards they accumulated during their participation period. This creates an unfair penalty where disqualification results in complete forfeiture of legitimately earned rewards rather than simply stopping future reward accrual. The financial impact scales with the duration of user participation before disqualification, potentially representing significant value loss for long-term protocol participants who later fail eligibility requirements.

Recommendation

In contracts/prime/staking/rewards/MainchainIncentivesController.sol#L879, the stopEmissionsFor function disqualifies users by updating their _userRewardPerTokenPaid mapping and setting their eligibility to false. However, this function fails to capture the user's accumulated rewards in the $.rewards[_chainEid][_user] mapping before disqualification occurs. The function only updates the user's reward per token paid checkpoint without preserving their earned rewards up to that point.

High Severity Issues

Excess native fee payment becomes permanently locked in contract

Resolved

Path

contracts/prime/staking/rewards/SidechainIncentivesController.sol#L312

Function

`claimAll`

Description

In contracts/prime/staking/rewards/SidechainIncentivesController.sol#L312, the claimAll function requires that msg.value be greater than or equal to fee.nativeFee to cover LayerZero messaging costs. However, the function only uses the exact fee.nativeFee amount when calling the LayerZero endpoint through _send at line 319, which internally transfers exactly fee.nativeFee to the endpoint via $.endpoint.send{value: _fee.nativeFee}. Any excess native currency sent by the user beyond the required fee amount remains trapped in the contract without any mechanism for retrieval.

The contract lacks a refund mechanism to return overpaid amounts to users, and there is no administrative function to recover these locked funds. This design flaw creates an accumulation point where user overpayments become permanently inaccessible. While LayerZero typically handles refunds through the _refundAddress parameter, the current implementation sets this to the contract address rather than the user address, preventing automatic refunds from reaching the original sender.

Consequently, users who accidentally send more native currency than required for transaction fees will permanently lose the excess amount. Overtime, these locked funds accumulate in the contract balance, representing lost value that neither users nor administrators can recover. The financial impact on individual users may be small but becomes significant in aggregate across multiple transactions and users.

Recommendation

We recommend implementing a refund mechanism by either calculating and returning excess msg.value amounts immediately within claimAll, or by setting the _refundAddress parameter in _send to the user address instead of the contract address to enable automatic LayerZero refunds and also add reentrancy guard within the function to avoid reentrancy attacks during refund

Cache-based protocol value calculation enables reward manipulation

Resolved

Path

primefi-contracts/contracts/prime/staking/rewards/MainchainIncentivesController.sol#L355

Function

`earned`

Description

In primefi-contracts/contracts/prime/staking/rewards/MainchainIncentivesController.sol#L355, the earned function calculates user rewards by using _getAndUpdateTokenValueInUSD to determine the protocol's total value in USD. This function relies on cached token prices that are only updated periodically, while user balances are valued using _getTokenValueInUSD, which always fetches the latest oracle price. However, this discrepancy allows the protocol's total value to lag behind real market prices, especially during periods of rapid price movement.

However, this flaw enables a malicious user to exploit the timing difference between cached and real-time prices. By waiting for a market event that increases token prices, the user can trigger an updateReward when the cached protocol value is still low, but their own balance is valued at the higher, current price. This results in the user receiving a disproportionately large share of the rewards, as the denominator in the reward calculation is artificially deflated while the numerator is inflated.

Consequently, this vulnerability can lead to excessive reward extraction by attackers, draining the reward pool and undermining the intended distribution mechanism. Honest users may receive less than their fair share, and the protocol's sustainability is threatened by the potential for repeated exploitation. The issue is particularly severe in volatile markets, where price changes can be significant between cache updates.

Recommendation

We recommend ensuring that both user balances and protocol value are calculated using the same, up-to-date price data by synchronising cache updates or always using the latest oracle prices for all reward calculations.

Reward per token calculation enables reward extraction by manipulating the protocol value

Open

Path

primefi-contracts/contracts/prime/staking/rewards/MainchainIncentivesController.sol#L290-294

Function

`rewardPerToken`

Description

In primefi-contracts/contracts/prime/staking/rewards/MainchainIncentivesController.sol#L290- 294, the rewardPerToken function calculates the reward per token by dividing the PRFI token emission rate by the totalProtocolValue, which is denominated in USD. This code section is responsible for determining how much reward each participant accrues over time based on the protocol's total value. However, the calculation uses the USD value of deposited assets as the divisor, and the PRFI tokens emission amount as a dividend rather than its USD value of PRFI tokens emission rate. This creates a mismatch between the reward emission and the protocol's actual value, allowing the rewardPerToken value to increase when the totalProtocolValue decreases.

However, this flaw allows a malicious user to monitor the protocol and trigger reward updates when the totalProtocolValue is temporarily low, such as during a market downturn or oracle manipulation. By claiming rewards at these moments, the user can receive a disproportionately high share of the distributed PRFI tokens. The calculation does not account for the real-time value of PRFI in USD, so the system can be gamed by timing claims to coincide with periods of low protocol value, extracting more tokens than intended.

Consequently, this vulnerability can lead to significant over-distribution of rewards, draining the reward pool and undermining the intended incentive structure. Honest users may receive less than their fair share, while attackers can extract excess rewards by exploiting fluctuations in the protocol's asset value. This undermines the fairness and sustainability of the reward mechanism and exposes the protocol to financial losses.

Recommendation

We recommend revising the rewardPerToken calculation to use the USD value of PRFI tokens as the dividend to make sure that the reward emission rate is constant in terms of USD value.

Claim bypasses pause and access control, enabling unauthorized reward claims

Resolved

Path

primefi-contracts/contracts/prime/staking/rewards/MainchainIncentivesController.sol:481

Function

`_claim`

Description

In primefi-contracts/contracts/prime/staking/rewards/MainchainIncentivesController.sol:481, the _claim function is declared as public and lacks the whenNotPaused modifier. This function is responsible for processing user reward claims by resetting the user's reward balance and vesting tokens via the _vestTokens function. The claim and claimAll functions, which are the intended public interfaces for claiming rewards, both invoke _claim and are correctly protected by the whenNotPaused modifier.

However, because _claim is public and not restricted by whenNotPaused, any user can directly call _claim even when the contract is paused. This bypasses the intended pause mechanism, which is designed to halt all reward claims during emergencies or maintenance. Additionally, _claim accepts arbitrary chain endpoint IDs, allowing users to claim rewards for any chain, circumventing the intended cross-chain claim flow that should only be initiated via the _lzReceive function from the SidechainIncentiveController contract.

Consequently, the pause mechanism is rendered ineffective, as users can still claim rewards during paused states, undermining emergency controls. This weakens the protocol's operational security and could result in financial loss or disruption of intended reward distribution logic.

Recommendation

We recommend restricting the _claim function to internal or private visibility and ensuring that only the intended public interfaces, which are protected by the whenNotPaused modifier, can invoke reward claims. This will enforce the pause mechanism and prevent unauthorized or cross-chain reward claims.

Oracle price staleness checks are missing in _getAndUpdateTokenValueInUSD, risking reward manipulation

Resolved

Path

primefi-contracts/contracts/prime/staking/rewards/MainchainIncentivesController.sol#L505

Function

`_getAndUpdateTokenValueInUSD`

Description

In primefi-contracts/contracts/prime/staking/rewards/MainchainIncentivesController.sol#L505, the _getAndUpdateTokenValueInUSD function updates the cached price for a token by calling the latestRoundData method of the IChainlinkAggregator contract and storing the result. This code section is responsible for updating the oracle price and timestamp when the update period has elapsed.

However, the function does not perform any staleness or validity checks on the returned price data. Unlike the _getTokenValueInUSD function, which enforces that the price is positive, the update time is recent, and the round is complete, _getAndUpdateTokenValueInUSD omits these checks. This omission allows stale, incomplete, or invalid price data to be cached and used in protocol calculations. As a result, the protocol may operate on outdated or incorrect price information, which can be exploited by users to manipulate reward calculations or protocol value assessments.

Consequently, the absence of staleness and validity checks in this function undermines the reliability of the protocol's price feeds. Attackers may exploit this by timing their actions to coincide with stale or manipulated oracle data, extracting unearned rewards or causing financial discrepancies. This weakens the protocol's economic security and may result in user losses or protocol insolvency if exploited at scale.

Recommendation

We recommend adding explicit checks in _getAndUpdateTokenValueInUSD to ensure the price is positive, the update time is within the allowed period, and the round is complete, mirroring the validation logic in _getTokenValueInUSD. This will ensure only fresh and valid price data is used for protocol operations.

Medium Severity Issues

Missing reward token transfer in notifyRewardAmount risks claim failures

Resolved

Path

primefi-contracts/contracts/prime/staking/rewards/MainchainIncentivesController.sol#L182

Path

`notifyRewardAmount`

Description

In primefi-contracts/contracts/prime/staking/rewards/MainchainIncentivesController.sol#L182, the notifyRewardAmount function is responsible for initialising and updating the reward distribution parameters, including the reward amount and the distribution period. However, this function does not perform a transfer of the specified reward tokens from the owner or a designated funding account into the contract. As a result, the contract may not hold the required reward tokens when the distribution period begins, even though the internal accounting assumes the tokens are available for claims.

However, this flaw means that if the contract is not pre-funded with the correct amount of reward tokens during notifyRewardAmount is called, subsequent reward claims by users may fail due to insufficient contract balance. The absence of an explicit transferFrom operation in notifyRewardAmount creates a dependency on manual funding, which is error-prone and can lead to operational failures. This design also increases the risk of reward underfunding, as there is no guarantee that the contract will be able to fulfil its reward obligations once the distribution period starts.

Consequently, users may experience failed reward claims, loss of trust, or financial losses if the contract is not properly funded at the time of reward distribution. The lack of an atomic funding mechanism undermines the reliability and predictability of the reward system, potentially exposing the protocol to reputational and operational risks.

Recommendation

We recommend updating notifyRewardAmount to include a transferFrom operation that moves the specified reward amount from the owner or funding account into the contract, ensuring the contract is fully funded before reward distribution begins.

Low Severity Issues

Lack of Ownable inheritance prevents secure ownership transfer

Resolved

Path

primefi-contracts/contracts/prime/oracles/chainlink/DataStreamConsumer.sol:114-116

Description

In primefi-contracts/contracts/prime/oracles/chainlink/DataStreamConsumer.sol:114-116, the DataStreamConsumer contract defines an onlyOwner modifier that restricts access to certain functions based on a private owner variable set at deployment. This code section is responsible for enforcing access control to privileged operations. However, the contract does not inherit from the standard Ownable contract, and it does not provide any mechanism to transfer or renounce ownership after deployment. As a result, the owner's address is immutable for the lifetime of the contract.

However, this design flaw means that if the original owner's address becomes inaccessible, compromised, or needs to be rotated for operational or security reasons, there is no way to update the owner. The absence of a transferOwnership function or similar mechanism prevents the contract from adapting to changes in project governance, team structure, or security posture. This limitation is especially problematic in production environments where ownership transfer is a standard requirement for secure and flexible contract management.

Consequently, the inability to transfer ownership can lead to operational bottlenecks, increased risk in the event of key loss or compromise, and reduced maintainability. If the owner account is lost or compromised, privileged functions become permanently inaccessible or exposed to unauthorised control, undermining the contract's security and reliability. The lack of standard ownership management also complicates integration with tooling and best practices that expect Ownable semantics.

Recommendation

We recommend inheriting from the OpenZeppelin Ownable contract to provide robust, standardised ownership management, including the ability to transfer and renounce ownership as needed.

Overwriting global totalSupply without chain context

Resolved

Path

primefi-contracts/contracts/prime/staking/rewards/MainchainIncentivesController.sol#l384

Function name

`removeTokenAggregator`

Description

In the _updateBalance function, the assignment $.totalSupply[token] = totalSupply_; overwrites the global totalSupply for a token without considering the chain context. Since the protocol supports multiple chains, this can lead to incorrect total supply tracking and reward calculations, as updates from one chain may overwrite values from another.

Recommendation

Store and update total supply per chain using a mapping like $.totalSupplyByChain[chainEid][token] instead of a global variable. This ensures each chain's supply is tracked independently and prevents cross-chain state corruption.

Informational Issues

Missing aggregator check in removeTokenAggregator allows invalid removals

Resolved

Path

primefi-contracts/contracts/prime/staking/rewards/MainchainIncentivesController.sol#l384

Function Name

`removeTokenAggregator`

Description

In primefi-contracts/contracts/prime/staking/rewards/MainchainIncentivesController.sol#l384, the removeTokenAggregator function removes a token's aggregator without verifying if one exists. However, this omission allows the function to be called on tokens with no aggregator, leading to misleading state changes and event emissions.

Consequently, this can confuse users and off-chain systems, as the contract may emit removal events for non-existent aggregators and update internal mappings incorrectly.

Recommendation

We recommend adding a check to ensure the token has an aggregator set before removal, reverting if none exists

Lack of zero-value check when setting prfiPerSecond

Resolved

Path

contracts/prime/staking/rewards/MainchainIncentivesController.sol#L399

Function Name

`setPrfiPerSecond`

Description

The setPrfiPerSecond function does not validate that the input value prfiPerSecond_ is non-zero before assignment. Allowing a zero value disables all reward emissions, which may be accidental and could disrupt protocol incentives or user expectations.

Recommendation

We recommend adding a require(prfiPerSecond_ > 0, "ZeroValueNotAllowed"); check to prevent setting the emission rate to zero. This ensures reward emissions cannot be unintentionally or maliciously disabled.

Lack of zero-value check when setting updateOraclePeriod

Resolved

Path

contracts/prime/staking/rewards/MainchainIncentivesController.sol#L407

Function Name

setUpdateOraclePeriod

Description

The setUpdateOraclePeriod function does not validate that the input value updateOraclePeriod_ is non-zero before assignment. Setting this value to zero would cause all oracle price updates to be considered immediately stale, potentially breaking price feeds and reward calculations.

Recommendation

We recommend adding a require(updateOraclePeriod_ > 0, "ZeroValueNotAllowed"); check to prevent setting the oracle update period to zero. This ensures the protocol always enforces a valid, non-zero update interval for price feeds.

Manual Setting of Token Decimals Instead of Dynamic Retrieval

Resolved

Path

contracts/prime/staking/rewards/MainchainIncentivesController.sol#L407

Function Name

`setUpdateOraclePeriod`

Description

The setTokenDecimals function allows the contract owner to manually set the decimals for each token. This approach is error-prone, as it relies on off-chain input and can lead to misconfiguration. Most ERC20 tokens expose a decimals() function, which should be queried directly to ensure accuracy and reduce operational risk.

Recommendation

We recommend removing the manual decimals setting and instead fetch the decimals dynamically from each token contract using the decimals() function. This ensures the contract always uses the correct value and eliminates the risk of human error.

Inefficient token decimal normalization

Acknowledged

Path

contracts/prime/staking/rewards/MainchainIncentivesController.sol#L439

Function Name

`setUpdateOraclePeriod`

Description

The current logic for normalizing _balance and _totalSupply to protocol decimals is difficult to read and maintain. The code uses conditional branches to handle cases where tokenDecimals is less than, equal to, or greater than PROTOCOL_VALUE_DECIMALS, with manual multiplication or division by powers of ten. This approach is verbose and error-prone, especially as the protocol evolves.

Additionally, if tokenDecimals > PROTOCOL_VALUE_DECIMALS, there is a risk of precision loss. The code does not enforce a safety check to prevent excessive precision loss or unexpected behavior for tokens with unusually high decimals.

Recommendation

We recommend refactoring the normalization logic to use a single, clear formula

```
require(tokenDecimals <= PROTOCOL_VALUE_DECIMALS, "Token decimals exceed
protocol limit");
uint256 scale = 10 ** (PROTOCOL_VALUE_DECIMALS - tokenDecimals);
_balance = _balance * scale;
_totalSupply = _totalSupply * scale;
```

Alternatively, if supporting tokens with more decimals is required, use a safe scaling approach and document the precision loss.

Unused internal function increases contract bytecode without providing functionality

Resolved

Path

contracts/prime/staking/rewards/SidechainIncentivesController.sol#L322

Function Name

`_vestTokens`

Description

In contracts/prime/staking/rewards/SidechainIncentivesController.sol#L322, the _vestTokens function is defined as an internal function that handles token vesting operations but is never called within the contract. The function contains complete implementation logic including input validation and external contract interactions, yet serves no purpose in the current sidechain controller architecture.

The presence of this unused code increases deployment gas costs and contract complexity without adding functional value. Dead code also creates maintenance overhead and potential confusion for developers and auditors who might assume it serves a purpose in the contract operation.

Recommendation

We recommend removing the _vestTokens function entirely from the SidechainIncentivesController contract to reduce bytecode size, improve code clarity, and eliminate maintenance overhead associated with unused functionality.

Usage of SafeMath Is Redundant in Solidity ⩾ 0.8.0

Acknowledged

Description

The contract imports and uses OpenZeppelin's SafeMath library for arithmetic operations. However, starting from Solidity version 0.8.0, integer overflow and underflow checks are performed natively by the compiler. This makes the use of SafeMath redundant and unnecessary in all contracts compiled with Solidity 0.8.0 or later.

Consequently, Unnecessary code complexity and additional bytecode size, missed opportunity for minor gas savings and improved readability.

Recommendation

Remove SafeMath imports and perform the mathematical operations without using .add(), sub(), mul() and div().

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 Labs. We performed our audit according to the procedure described above.

Issues of critical , high , medium , low and informational issues were found
A few issues have been resolved, others have been acknowledged, and one remains unresolved. We strongly recommend conducting multiple audits and participating in bug bounty programs to ensure the highest level of security.

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 leading name in Web3 security, offering top-notch solutions to safeguard projects across DeFi, GameFi, NFT gaming, and all blockchain layers. With seven years of expertise, we've secured over 1400 projects globally, averting over $3 billion in losses. Our specialists rigorously audit smart contracts and ensure DApp safety on major platforms like Ethereum, BSC, Arbitrum, Algorand, Tron, Polygon, Polkadot, Fantom, NEAR, Solana, and others, guaranteeing your project's security with cutting-edge practices.

7+ Years of Expertise	**1M+** Lines of Code Audited
50+ Chains Supported	**1400+** Projects Secured

Follow Our Journey

AUDIT REPORT

September 2025

For

PRIME NUMBERS

QuillAudits

Canada, India, Singapore, UAE, UK

www.quillaudits.com audits@quillaudits.com