QuillAudits

AUDIT REPORT

April , 2025

For

PRIME NUMBERS

Table of Content

Table of Content 02

Executive Summary 03

Number of Issues per Severity 05

Checked Vulnerabilities 06

Techniques & Methods 08

Types of Severity 10

Types of Issues 11

High Severity Issues 12

1. Bypassing Withdrawal Approval 12

Medium Severity Issues 13

1. Use of "transfer" opcode to send ETH can lock rewards and withdrawals if recipient is a contract 13

Low Severity Issues 14

1. Missing State Tracking in contract for requestWithdraw and cancel-WithdrawRequests 14

2. notifyRewardAmount() Does Not Track Rewards in contract State 15

Informational Severity Issues 16

1. Incorrect Use of Custom Errors with require Statements 16

2. Missing Checks for Math Operation Failures in Reward Logic 17

3. Attacker can delay rewards of users by calling notifyRewardAmount with some Weis 18

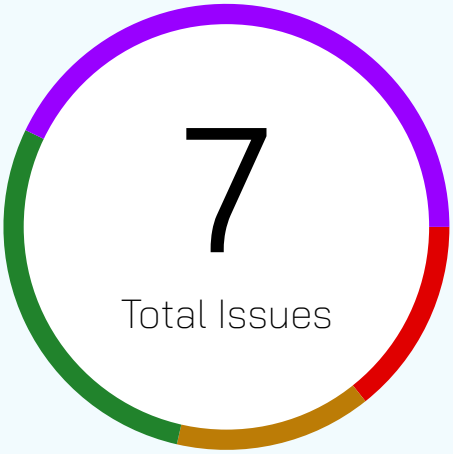
Closing Summary & Disclaimer 19

Executive Summary

Project name	Prime Number
Project URL	https://primenumbers.xyz/
Overview	The PrimeStakedXDC contract implements a native XDC staking system that allows users to deposit XDC and receive a representative token called psXDC in return.
Audit Scope	The scope of this Audit was to analyze the Prime Number Smart Contracts for quality, security, and correctness.
Source Code link	https://github.com/PrimeNumbersLabs/liquid-staking-contracts/blob/main/contracts/PrimeStakedXDC.sol
Contracts in Scope	PrimeStakedXDC.sol
Branch	Main
Commit Hash	e8832de18a66c7c6bc8a6095c6b4483c43c6952e
Language	Solidity
Blockchain	EVM
Method	Manual Analysis, Functional Testing, Automated Testing
Review 1	7th April 2025 - 11th April 2025
Updated Code Received	14th April 2025

Review 2	14th April 2025 - 16th April 2025
Fixed In	https://github.com/PrimeNumbersLabs/liquid-staking-contracts/compare/main...feature/audit 6c9b36fd4964c025889f3cfbd5fdd791aa10440d

Number of Issues per Severity



High	1 (14.29%)
Medium	1 (14.29%)
Low	2 (28.57%)
Informational	3 (42.86%)

Severity

Issues	High	Medium	Low	Informational
Open	0	0	0	0
Resolved	**1**	**1**	**2**	**1**
Acknowledged	0	0	0	**2**
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.

High Severity Issues

Bypassing Withdrawal Approval

Resolved

Path

contracts/PrimeStakedXDC.sol

Description

The approveWithdrawRequest() function sets a boolean flag _canWithdraw[user] = true, allowing a user to withdraw funds. However, this flag is not time-bound or tied to the stake amount at the time of approval.
An attacker can exploit this by getting approval for a small stake and delaying their actual withdrawal. Later, they can stake a large amount and withdraw without requiring further approval, thus bypassing the intended per-withdrawal approval mechanism.
This behavior breaks the intended protocol security model, where every withdrawal should be approved based on the user's current stake and context.

Attack Scenario

Attacker stakes 1 wei of XDC.

Attacker calls requestWithdraw().

Owner approves the request: _canWithdraw[attacker] = true.

Attacker does not withdraw, but instead stakes a large amount, e.g., 1000 XDC.

Later, attacker calls withdraw() and is not required to request or receive approval again.

The attacker withdraws all funds based on a previous approval, bypassing approval for 1000 XDC.

Recommendation

Instead of a boolean flag, use a structure to store approved withdrawal amounts. Only allow withdrawal if:
approvedAmount >= currentStake.

Medium Severity Issues

Use of "transfer" opcode to send ETH can lock-rewards and withdrawals if recipient is a contract

Resolved

Path

contracts/PrimeStakedXDC.sol

Description

In the PrimeStakedXDC contract, native XDC is sent to users during reward claims and withdrawals-using Solidity's .transfer function. This opcode only forwards 2300 gas, which is risky and outdated.
This is dangerous for two reasons:
- Gas costs of EVM instructions may change signifcantly during hard forks which may previously be assumed fxed gas costs. EIP 1884 as an example, broke several existing smart contracts due to a cost increase of the SLOAD instruction.
- If the recipient is a contract or a multisig safe, with a "receive"/"fallback" function which requires >2300 gas, e.g. safes that execute extra logic in the "receive"/"fallback" function, the transfer function will always fail for them due to out of gas errors.

Impact

Makes the protocol incompatible with a wide range of DeFi tools and multisig wallets.

Recommendation

Replace .transfer with a low-level call that forwards all available gas and checks for success

```
(bool success, ) = user.call{value: amount}("");
require(success, "ETH transfer failed");
```

Low Severity Issues

Missing State Tracking in contract for requestWithdraw and cancelWithdrawRequests

Resolved

Path

contracts/PrimeStakedXDC.sol

Description

The requestWithdraw() and cancelWithdrawRequest() functions emit events to signal a user's intent-to withdraw or cancel a withdrawal. However, these functions do not update any internal state to refect the user's current withdrawal status.

The actual logic that allows a user to withdraw is fully controlled by the owner via approveWithdrawRequest(), which sets canWithdraw[user] = true. As a result, there's no on-chain tracking of whether a user has an active withdrawal request or has canceled it; only off-chain listeners can infer that by watching events.

Impact

Without internal state tracking, these functions can be called repeatedly without restriction, allowing attackers or bots to spam requestWithdraw() and cancelWithdrawRequest(). This can:

- A malicious user could spam requestWithdraw() many times (e.g., 10+ calls), each emitting an event.
- Off-chain systems (e.g., servers or relayers watching for WithdrawRequested events)- may interpret each event as a valid new request and respond with repeated calls to approveWithdrawRequest().
- This allows an attacker to drain the relayer's gas or rate-limit capacity by forcing it to process redundant or unnecessary transactions.
- Make it impossible to enforce rules like cooldowns, delays, or automatic approvals based on on-chain request time
- Result in a poor user experience where users believe they've made a valid withdrawal request, but nothing has changed in contract state.

Recommendation

Introduce a mechanism to explicitly track withdrawal and cancel withdrawal requests on-chain.

notifyRewardAmount() Does Not Track Rewards in contract State

Resolved

Path

contracts/PrimeStakedXDC.sol

Description

The notifyRewardAmount() function allows the owner to deposit ETH (XDC) into the contract as rewards for stakers. While it emits a RewardAdded(reward) event, the function-does not store the reward amount in state or update any cumulative reward tracking variable.

This means there's no on-chain record of how much total reward has been added over time. The only way to trace this is through off-chain event logs, which can be incomplete or unreliable if logs are pruned, missed, or not indexed properly.

Impact

On-chain reward history is lost, there is no way to audit total rewards added over time.

Recommendation

Introduce a variable inside the contract that gets updated every time notifyRewardAmount-function has been called. This enables reliable on-chain accounting and makes reward emissions auditable, even without external indexers.

Informational Severity Issues

Incorrect Use of Custom Errors with require Statements

Acknowledged

Path

contracts/PrimeStakedXDC.sol

Description

The contract defines custom errors like ZeroStakeAmount(), RewardApyTooHigh(), and others. These-are intended to be used as a clearer and more efficient alternative to revert strings. However,-in multiple places, these errors are incorrectly passed as arguments to require():

require(amount > 0, ZeroStakeAmount()); // incorrect

This is not the correct pattern for using custom errors. According to Solidity documentation, custom errors should be used with an if statement followed by a revert:

if (amount == 0) revert ZeroStakeAmount(); // correct

Using require(..., CustomError()) still compiles because custom errors are interpreted as expressions,-but this pattern is invalid as it leads to inconsistent error handling and poor readability.

Recommendation

Update validation checks that use custom errors incorrectly with require() to use the recommended pattern:

// Replace:
require(condition, CustomError());

// With:
if (!condition) revert CustomError();
This aligns the implementation with Solidity's recommended usage, ensures clear and consistent error reporting.

Developer Response

We would like a more detailed explanation, since according to official documentation from several versions ago it is already completely equivalent.

Auditor Response

This isnt an issue instead of reported as best practice, there are multiple resources indication that custom errors are meant to be used inside if statment with reverts. Here are few example resources of correct implementation.

https://dev.to/george_k/embracing-custom-errors-in-solidity-55p8
https://medium.com/coinmonks/mastering-solidity-require-and-custom-errors-in-ethereum-contracts-b491565f1592
https://soliditylang.org/blog/2021/04/21/custom-errors/

Developer Response

We would like a more detailed explanation, since according to official documentation from several versions ago it is already completely equivalent.

Auditor Response

We agree our current way of writing is valid no further refute.

Missing Checks for Math Operation Failures in Reward Logic

Acknowledged

Path

contracts/PrimeStakedXDC.sol

Description

Throughout the reward calculation logic in the contract, the code uses Solidity's tryAdd, trySub, and-tryMul functions from the Math library to perform arithmetic operations. These functions return-a (bool success, uint256 result) and in the current implementation, the success flag is ignored in most cases.

```
(bool success, uint256 result) = rewardPerToken().trySub(s.userRewardPerTokenPaid[account]);
(success, result) = result.tryMul(balanceOf(account));
(success, result) = result.tryDiv(WHOLE);
(success, result) = result.tryAdd(_rewards[account]);
```

Recommendation

Instead of silently ignoring failed math operations, it is recommended to explicitly validate the result of each arithmetic step.

```
(bool success, uint256 delta) =
rewardPerToken().trySub(s.userRewardPerTokenPaid[account]);
require(success, "reward calc failed");

(success, delta) = delta.tryMul(balanceOf(account));
require(success, "reward calc failed");

(success, delta) = delta.tryDiv(WHOLE);
require(success, "reward calc failed");
```

Developer Response

We think only one verification is needed on the earned and rewardPerToken functions, since every other result will give a successful operation. We also put gas limit on calls (instead of transfer).

Auditor Response

No further Refute.

Attacker can delay rewards of users by calling notifyRewardAmount with some Weis

Resolved

Path

contracts/PrimeStakedXDC.sol

Description

The notifyRewardAmount() function can be called by any address with no access control or minimum reward threshold. This enables a griefing attack where an adversary sends trivial reward amounts (e.g., 1 wei) repeatedly to reset the lastUpdateTime and extend periodFinish by the full rewardsDuration. As a result, rewardPerToken() accrual is disrupted, and stakers experience significant delays or denial of expected rewards. This attack does not incur meaningful cost for the attacker but can indefinitely block proper reward distribution for all users.

Developer Response

We add onlyOwner to notifyRewardAmount function to fix it

Closing Summary

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

1 issue of high severity ,1 issue of medium severity, 2 low issues of low severity, & 3 issues of informatory severity was found. In end the Prime Number team resolved almost all issues and acknowledged 2 issues

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+ Years of Expertise	**1M+** Lines of Code Audited
$30B+ Secured in Digital Assets	**1400+** Projects Secured

Follow Our Journey

AUDIT REPORT

April , 2025

For

PRIME NUMBERS

QuillAudits

Canada, India, Singapore, UAE, UK

www.quillaudits.com audits@quillaudits.com