Security Audit Report

Animoca Staking Pool

v1.1

June 24, 2025

Table of Contents

License

Disclaimer

This audit has been performed by

Oak Security GmbH

https://oaksecurity.io/
info@oaksecurity.io

Introduction

Purpose of This Report

Oak Security GmbH has been engaged by New Frame Limited to perform a security audit of Audit of the Animoca Staking Pool smart contracts.

The objectives of the audit are as follows:

1. Determine the correct functioning of the protocol, in accordance with the project specification.
2. Determine possible vulnerabilities, which could be exploited by an attacker.
3. Determine smart contract bugs, which might lead to unexpected behavior.
4. Analyze whether best practices have been applied during development.
5. Make recommendations to improve code safety and readability.

This report represents a summary of the findings.

As with any code audit, there is a limit to which vulnerabilities can be found, and unexpected execution paths may still be possible. The author of this report does not guarantee complete coverage (see disclaimer).

Codebase Submitted for the Audit

The audit has been performed on the following target:

Repository	https://github.com/animoca/ethereum-contracts/tree/main/contracts/staking
Commit	`13471b9285ec7b3ba0215ff6b5d753ae7fc6bf51`
Scope	All contracts were in scope.
Fixes verified at commit	`c7867f97176404bbbe138bda13ec6ac48a7c9c98` Note that only fixes to the issues described in this report have been reviewed at this commit. Any further changes such as additional features have not been reviewed.

Methodology

The audit has been performed in the following steps:

1. Gaining an understanding of the code base's intended purpose by reading the available documentation.
2. Automated source code and dependency analysis.
3. Manual line-by-line analysis of the source code for security vulnerabilities and use of best practice guidelines, including but not limited to:
    a. Race condition analysis
    b. Under-/overflow issues
    c. Key management vulnerabilities
4. Report preparation

Functionality Overview

Animoca provides a modular framework for creating staking pools that support various token standards (ERC20, ERC721, ERC1155) and flexible reward mechanisms. The core logic manages user stakes as "stake points," calculates proportional rewards over time, and allows for secure staking, withdrawal, and claiming of rewards, with extensibility for different token and reward types. This framework is designed to be a foundational layer for diverse staking applications.

How to Read This Report

This report classifies the issues found into the following severity categories:

Severity	Description
Critical	A serious and exploitable vulnerability that can lead to loss of funds, unrecoverable locked funds, or catastrophic denial of service.
Major	A vulnerability or bug that can affect the correct functioning of the system, lead to incorrect states or denial of service.
Minor	A violation of common best practices or incorrect usage of primitives, which may not currently have a major impact on security, but may do so in the future or introduce inefficiencies.
Informational	Comments and recommendations of design decisions or potential optimizations, that are not relevant to security. Their application may improve aspects, such as user experience or readability, but is not strictly necessary. This category may also include opinionated recommendations that the project team might not share.

The status of an issue can be one of the following: **Pending, Acknowledged**, **Partially Resolved**, or **Resolved**.

Note that audits are an important step to improving the security of smart contracts and can find many issues. However, auditing complex codebases has its limits and a remaining risk is present (see disclaimer).

Users of the system should exercise caution. In order to help with the evaluation of the remaining risk, we provide a measure of the following key indicators: **code complexity**, **code readability**, **level of documentation**, and **test coverage**. We include a table with these criteria below.

Note that high complexity or low test coverage does not necessarily equate to a higher risk, although certain bugs are more easily detected in unit testing than in a security audit and vice versa.

Code Quality Criteria

The auditor team assesses the codebase's code quality criteria as follows:

Criteria	**Status**	**Comment**
Code complexity	**Low-Medium**	-
Code readability and clarity	**Medium-High**	-
Level of documentation	**Medium**	-
Test coverage	**High**	`hardhat coverage` reports a test coverage of `100%`

Summary of Findings

No	Description	Severity	Status
1	Incorrect `totalStaked` calculation breaks accounting	Critical	Resolved
2	First-staker advantage may create non-intuitive reward distribution in some cases	Minor	Resolved
3	Undistributed reward dust lacks handling mechanism	Minor	Resolved
4	Missing duration limits in reward distribution	Minor	Acknowledged
5	Internal `_stake` and `_withdraw` functions lack built-in reentrancy protection	Minor	Resolved
6	Incompatibility with fee-on-transfer and rebasing tokens	Minor	Resolved
7	Privileged actors	Informational	Partially Resolved
8	Miscellaneous comments	Informational	Resolved

Detailed Findings

1. Incorrect `totalStaked` calculation breaks accounting

Severity: Critical

In `ethereum-contracts/contracts/staking/linear/LinearPool.sol:146`, the `_withdraw` function incorrectly updates the `totalStaked` state variable by setting it equal to a single user's remaining stake instead of properly decrementing it. This results in the loss of accounting data for all other users' stakes after the first withdrawal.

This causes significant impacts:

- Complete disruption of the reward distribution system
- Unfair reward allocation, with some users receiving excessive rewards while others get nothing
- Potential permanent loss of stake accounting, effectively locking funds for users who stake after the first withdrawal
- Breakdown of core contract functionality as rewards are calculated based on the `totalStaked` value

When Alice stakes 100 tokens and Bob stakes 200 tokens, `totalStaked` correctly equals 300. However, when Alice withdraws 50 tokens, `totalStaked` incorrectly becomes 50 (Alice's remaining balance) instead of 250 (the correct total). This effectively erases Bob's 200 token stake from the accounting, causing catastrophic failures in reward calculations and subsequent withdrawals.

Recommendation

We recommend changing the implementation to properly decrement the global total instead of overwriting it. This can be done by replacing `totalStaked = currentStaked - stakePoints` with `totalStaked -= stakePoints`.

Status: Resolved

2. First-staker advantage may create non-intuitive reward distribution in some cases

Severity: Minor

In `ethereum-contracts/contracts/staking/linear/LinearPool.sol:58-61`, the `_updateReward` function conditionally updates `lastUpdated` only when the distribution has ended or when there are active stakes. This design choice results in the first staker after a zero-stake period receiving all accumulated rewards from that period, regardless of how long they actually stake. While this does not remove legitimate rewards from other stakers based on their staking duration, it creates a timing-dependent advantage where early entry after zero-stake periods provides disproportionate rewards. This behavior may be unexpected to users who assume rewards are distributed proportionally to actual staking participation time.

Recommendation

We recommend either documenting this behavior to set appropriate expectations or implementing a more sophisticated approach that tracks unallocated rewards during zero-stake periods separately.

Status: Resolved

3. Undistributed reward dust lacks handling mechanism

Severity: Minor

In `ethereum-contracts/contracts/staking/linear/LinearPool.sol:197-241`, the `addReward` function calculates dust (the remainder after dividing reward by duration) in all distribution scenarios. This dust is passed to `_computeAddReward` and included in the `RewardAdded` event, but no mechanism exists to actually distribute this dust to stakers. This creates a small but permanent loss of rewards that accumulates over multiple reward additions.

Recommendation

We recommend either documenting that dust amounts are intentionally not distributed or implementing a mechanism to handle dust (such as adding it to the next distribution or creating a separate mechanism to periodically distribute accumulated dust).

Status: Resolved

4. Missing duration limits in reward distribution

Severity: Minor

In `ethereum-contracts/contracts/staking/linear/LinearPool.sol:176-219`, the `addReward` function lacks upper bounds validation on the duration parameter. While it checks for the duration to not be zero, extremely large durations could be set accidentally or maliciously by rewarders, potentially making the reward system impractical or effectively disabling it for unreasonably long periods.

Recommendation

We recommend adding reasonable maximum duration limits to prevent operational issues. Consider adding validation that rejects durations exceeding a sensible upper bound, such as several years, to prevent both accidental misuse and potential malicious exploitation while maintaining flexibility for legitimate long-term reward distributions.

Status: Acknowledged

5. Internal `_stake` and `_withdraw` functions lack built-in reentrancy protection

Severity: Minor

In `ethereum-contracts/contracts/staking/linear/LinearPool.sol:109-116`, `137-149`, the internal `_stake` and `_withdraw` functions do not include the `nonReentrant` modifier, with reentrancy protection only applied at the external function level.

While this is an intentional design choice to provide flexibility for derived contracts, it creates a potential footgun where future implementations could inadvertently bypass reentrancy protection by overriding external functions without maintaining the modifier or by calling internal functions directly. This pattern, while not currently exploitable, deviates from defensive programming best practices and could introduce vulnerabilities in derived contracts, particularly when integrating with tokens that support callback mechanisms like ERC777.

Recommendation

We recommend considering adding the `nonReentrant` modifier directly to the internal `_stake` and `_withdraw` functions to ensure reentrancy protection cannot be inadvertently bypassed in derived contracts. Alternatively, clearly document this design choice and provide guidelines for derived contract implementations to ensure they maintain appropriate reentrancy protection when overriding these functions.

Status: Resolved

6. Incompatibility with fee-on-transfer and rebasing tokens

Severity: Minor

In `ethereum-contracts/contracts/staking/linear/stake/ERC20StakingLinearPool.sol:46-52`, the `_computeStake` function assumes the amount transferred equals the amount received, which breaks with fee-on-transfer tokens. Additionally, the `recoverERC20s` function in line `70` calculates recoverable amounts based on `totalStaked` without accounting for actual balance discrepancies. This creates accounting mismatches where `totalStaked` may exceed the actual token balance, potentially preventing withdrawals and causing the recovery function to fail.

Recommendation

We recommend explicitly documenting that fee-on-transfer and rebasing tokens are not supported, or implementing balance-change checks by measuring contract balance before and after transfers to ensure accurate accounting.

Status: Resolved

7. Privileged actors

Severity: Informational

The smart contracts under review implement functionality that relies on privileged actors with significant control over reward distribution:

- In `ethereum-contracts/contracts/staking/linear/reward/LinearPool_ERC20Rewards.sol`, the contract owner can change the `rewardHolder` address at any time via `setRewardHolder`, potentially disrupting ongoing reward distributions.
- Also, the `_computeClaim` function relies on `safeTransferFrom(rewardHolder, staker, reward)`, meaning the `rewardHolder` can revoke token approval to pause all reward claims.
- In `ethereum-contracts-main/contracts/staking/linear/LinearPool.sol`, addresses with `REWARDER_ROLE` can modify active reward distributions through `addReward`, allowing them to dilute existing rewards, extend timeframes, or adjust rates in ways that may disadvantage current stakers.

Recommendation

Consider implementing time-locked changes for critical parameters and documenting these privileged actors for users.

Status: Partially Resolved

The team has implemented a fix that correctly prevents reward dilution and acknowledges the fact the library may be used to implement staking solutions with privileged actors.

8. Miscellaneous comments

Severity: Informational

Miscellaneous recommendations can be found below.

Recommendation

The following are some recommendations to improve the overall code quality and readability:

- The variable name `newDisributionEnd` contains a typo and should be `newDistributionEnd` in `ethereum-contracts/contracts/staking/linear/LinearPool.sol:187`.
- The comment in line `ethereum-contracts/contracts/staking/linear/reward/LinearPool_ERC20Rewards.sol:L45` is incorrect. It states the function computes rewards for a staker, while this function should take care of token transfers when rewards are added to the contract.

Status: Resolved