Phi Staking
Security Review

JULY, 2025

www.chaindefenders.xyz
https://x.com/DefendersAudits

Lead Auditors

PeterSR

0x539.eth

Table of Contents

Protocol Summary

A comprehensive staking system for PHI tokens with time-weighted rewards, governance voting, and multi-token reward distribution.

The PHI Staking Contract implements an advanced staking mechanism that rewards users based on both their stake amount and holding duration through the Weighted Average Holding Period (WAHP) system. Users stake PHI tokens to receive non-transferable sPHI tokens that represent their voting power in governance decisions.

Disclaimer

The Chain Defenders team makes all effort to find as many vulnerabilities in the code in the given time period, but holds no responsibilities for the findings provided in this document. A security audit by the team is not an endorsement of the underlying business or product. The audit was time-boxed and the review of the code was solely on the security aspects of the Solidity implementation of the contracts.

Risk Classification

Likelihood/Impact	High	Medium	Low
High	H	H/M	M
Medium	H/M	M	M/L
Low	M	M/L	L

Audit Details

Commit Hash

eeb43b88f59541430d7c5157fdf95be83813de2f

Scope

Id	Files in scope
1	PHI.sol
2	PHIStaking.sol

Roles

Id	Roles
1	Owner
2	Emergency Role
3	Distributor Role
4	User

Executive Summary

Issues found

Severity	Count	Description
High	1	Critical vulnerabilities
Medium	0	Significant risks
Low	1	Minor issues with low impact
Informational	4	Best practices or suggestions
Gas	0	Optimization opportunities

Findings

High

High 01 Sweep Reward Token State Corruption

Location

PHIStaking.sol:264

Description

In `PHIStaking` there is an issue that arises when a reward token is swept using `sweepRewardToken` and then reused for new reward distributions. The contract fails to reset the reward token's state after sweeping, leading to corrupted reward accounting. Specifically:

1. Stale Reward State: After sweeping, the reward token's state (in `rewards[_token]`) retains old values like `rewardPerTokenStored`, `lastUpdate`, and `periodFinish`. These values remain even though the token balance has been transferred out.
2. Corrupted New Distributions: When new rewards are deposited for the same token:

- `depositReward` uses the stale state to calculate new reward rates, incorrectly mixing old and new reward periods.
- User reward calculations (`accumulatedRewardsPerToken`, `claimableAmounts`) reference outdated values, causing:
  - Old rewards (that were swept) to be paid from new deposits.
  - New rewards to be incorrectly distributed due to corrupted state.

Proof of Concept

Add the following test to the `PHIStakingTest.sol` file:

```
function test_SweepRewardToken_andThenSetNewRewards() public {
    uint256 depositAmount = 1000 ether;
    uint256 rewardAmount = 100 ether;
    address sweepRecipient = makeAddr("sweepRecipient");

    // Setup staking first
    vm.startPrank(user1);
    phi.approve(address(phiStaking), depositAmount);
    phiStaking.deposit(depositAmount, user1);
    vm.stopPrank();

    vm.startPrank(user2);
    phi.approve(address(phiStaking), depositAmount);
    phiStaking.deposit(depositAmount, user2);
    vm.stopPrank();

    // Add reward token
    phiStaking.addRewardToken(IERC20(address(rewardToken1)));
```

```
19
20      // Deposit rewards
21      vm.startPrank(distributor);
22      rewardToken1.approve(address(phiStaking), rewardAmount);
23      phiStaking.depositReward(address(rewardToken1), rewardAmount, 0);
24      vm.stopPrank();
25
26      vm.warp(block.timestamp + REWARD_DURATION + phiStaking.SWEEP_DELAY
        ()/2 - 1);
27
28      vm.startPrank(user1);
29      phiStaking.initiateWithdrawal(depositAmount);
30      vm.stopPrank();
31
32      vm.warp(block.timestamp + REWARD_DURATION + phiStaking.SWEEP_DELAY
        () - 1);
33      vm.startPrank(user1);
34      phiStaking.withdraw(false, user1);
35      vm.stopPrank();
36
37      // Fast forward past reward period + sweep delay
38      vm.warp(block.timestamp + REWARD_DURATION + phiStaking.SWEEP_DELAY
        () + 1);
39
40      // Check that token is sweepable
41      assertTrue(phiStaking.isSweepable(address(rewardToken1)));
42      uint256 sweepableAmount = phiStaking.getSweepableAmount(address(
        rewardToken1));
43      assertGt(sweepableAmount, 0);
44
45      // Perform sweep
46      vm.startPrank(distributor);
47      vm.expectEmit(true, true, false, true);
48      emit PHIStakingEvents.RewardTokenSwept(address(rewardToken1),
        sweepRecipient, sweepableAmount);
49
50      phiStaking.sweepRewardToken(address(rewardToken1), sweepRecipient)
        ;
51      vm.stopPrank();
52
53      // Verify sweep results
54      assertEq(rewardToken1.balanceOf(sweepRecipient), sweepableAmount);
55      assertEq(rewardToken1.balanceOf(address(phiStaking)), 0);
56
57      vm.warp(block.timestamp + REWARD_DURATION + phiStaking.SWEEP_DELAY
        () + 10);
58
```

```
59      phiStaking.setClaimEnabled(true);
60
61
62      vm.startPrank(distributor);
63      rewardToken1.approve(address(phiStaking), rewardAmount);
64      phiStaking.depositReward(address(rewardToken1), rewardAmount, 0);
65      vm.stopPrank();
66
67      // Claim rewards
68      vm.prank(user1);
69      phiStaking.claim(user1, user1);
70
71      // Check reward was transferred
72      assertGt(rewardToken1.balanceOf(user1), 0);
73  }
```

Recommendation

Reset the reward token's state during sweeping to prevent reuse of corrupted data and implement an epoch-based rewarding mechanism:

1. Remove token from `rewardTokens` array: Prevent global checkpoints from processing the swept token.
2. Delete reward state: Clear all stored data for the token in `rewards` mapping.
3. Re-add token for new distributions: If the token should be reused, require it to be re-added via `addRewardToken` for a fresh state.
4. Implement an epoch-based rewarding mechanism to escape state confusion due to persisting user data.

Update `sweepRewardToken` as follows:

```
function sweepRewardToken(
    address _token,
    address _to
) external onlyRole(DISTRIBUTOR_ROLE) {
    // ... existing checks ...

    // Remove token from rewardTokens array
    uint256 length = rewardTokens.length;
    for (uint256 i = 0; i < length; i++) {
        if (rewardTokens[i] == _token) {
            rewardTokens[i] = rewardTokens[length - 1];
            rewardTokens.pop();
            break;
```

```
            }
        }

        // Delete reward state
        delete rewards[_token];

        // ... existing transfer and event emission ...
    }
```

- After sweeping, the token must be re-added via `addRewardToken` before new distributions.
  - This ensures a clean state for new reward periods and prevents accounting corruption.
- Document this behavior to inform distributors that swept tokens require re-addition.

Status

Fixed

Low

Low 01 Sweep Can Be Executed At Exact Boundary

Location

PHIStaking.sol:264

Description

The `sweepRewardToken` function currently allows sweeping reward tokens when `block.timestamp ⩾ reward.periodFinish + SWEEP_DELAY`. Similarly, the `isSweepable` function uses the same ⩾ comparison. Using ⩾ means that sweeping is allowed exactly at the moment the sweep delay expires, not strictly after. This could lead to edge cases where the sweep is executed at the exact boundary, potentially causing confusion or off-by-one errors in time-sensitive logic. Additionally, the logic for determining sweepability is duplicated in multiple places, making the code harder to maintain and more error-prone.

Recommendation

Change the comparison in both `sweepRewardToken` and `isSweepable` from $\geqslant$ to $>$, ensuring that sweeping is only allowed strictly after the sweep delay has fully elapsed. Furthermore, consider overloading the `isSweepable` function to accept a `periodFinish` parameter, allowing for a more flexible and reusable check. This would simplify the `sweepRewardToken` function by delegating the sweepability logic to a single, well-defined place, improving code clarity and maintainability.

Status

Acknowledged

Informational

Info 01 Claiming Is Possible After Sweep Delay

Location

PHIStaking.sol:432

Description

The `checkpoint` function in the `PHIStaking` contract allows users to claim rewards at any time, including after the `SWEEP_DELAY` period has passed. The `SWEEP_DELAY` is intended as a grace period after the end of a reward distribution, after which any unclaimed rewards can be "swept" (i.e., withdrawn by the distributor). However, since sweeping is a manual process and not enforced by an automated keeper, there is a window where both claiming and sweeping are possible. This creates a race condition: if a user claims their rewards just before the sweep transaction is executed, they may receive rewards that the sweeper also attempts to withdraw, potentially leading to unexpected or unfair outcomes. This undermines the intended finality of the sweep process and could result in disputes or loss of funds for the distributor.

Recommendation

Disallow claiming rewards for a given reward token after its `SWEEP_DELAY` period has passed (i.e., after `reward.periodFinish + SWEEP_DELAY`). This can be enforced by adding a check in the `checkpoint` logic to prevent claims after the sweepable time. Additionally, consider automating the sweep process using a keeper or similar mechanism to ensure that sweeping occurs promptly after the delay, reducing the risk of race conditions and improving the robustness of the reward distribution system.

Status

Fixed

Info 02 Redundant Inheritance

Location

PHI.sol:17

Description

The `PHI` contract inherits from the `ERC20` contract directly, even though both `ERC20Bridgeable` and `ERC20Permit` which are also inherited already extend `ERC20`. This results in redundant inheritance of the same base contract. While Solidity's linearization prevents functional issues in this case, it adds unnecessary complexity to the inheritance hierarchy and can make the code harder to read and maintain. It may also increase the risk of ambiguity or errors if the base contract is updated or if multiple versions are used.

Recommendation

Remove the direct inheritance of `ERC20` from the `PHI` contract's inheritance list. Rely on the inheritance provided by `ERC20Bridgeable` and `ERC20Permit`, which already include `ERC20` functionality. This will simplify the contract's structure and avoid redundant code paths.

Status

Fixed

Info 03 Owner Not Whitelisted To Receive Tokens In Paused State

Location

PHI.sol:53

Description

When the `PHI` token contract is deployed, it is paused by default. While the `initialOwner` is added to the `isAllowedFrom` mapping (allowing them to send tokens even when paused), they are not added to the `isAllowedTo` mapping. This means the `initialOwner` can transfer tokens out, but cannot receive tokens while the contract is paused - except for the initial mint in the constructor. If the owner needs to receive tokens while the token is paused, the transfer will fail unless they are explicitly added to `isAllowedTo` or the senders are whitelisted. This could cause confusion or operational issues during the paused state.

Recommendation

Add the `initialOwner` to the `isAllowedTo` mapping in the constructor, just as is done for `isAllowedFrom`. This ensures the owner can both send and receive tokens while the contract is paused, improving usability and reducing the risk of failed transfers during the paused period.

Status

Fixed

Info 04 Reward Receiver From Struct Not Used

Location

PHIStaking.sol

Description

The `PHIStaking` contract allows users to set a custom `rewardReceiver` address in their `UserData` struct via the `setRewardsReceiver` function. This is intended to let users specify a default address to receive their staking rewards. However, in the current implementation, this `rewardReceiver` value is never actually used when rewards are distributed. Instead, reward claims (such as during withdrawals or explicit claims) always use the receiver address provided as a function argument, or default to the user's own address if none is provided. As a result, setting a `rewardReceiver` has no effect, and users cannot benefit from this intended customization.

Recommendation

Update the reward claiming logic so that, if no receiver address is provided (i.e., the argument is zero), the contract checks and uses the user's `rewardReceiver` as the destination for rewards. Only if both are unset should the contract default to sending rewards to the user's own address. This ensures the `rewardReceiver` field is functional and users can direct their rewards as intended.

Status

Fixed