Smart Contract Security Audit Report

Table Of Contents

1 Executive Summary

On 2024.10.25, the SlowMist security team received the Prosper team's security audit application for Prosper Staking Pool, developed the audit plan according to the agreement of both parties and the characteristics of the project, and finally issued the security audit report.

The SlowMist security team adopts the strategy of "white box lead, black, grey box assists" to conduct a complete security test on the project in the way closest to the real attack.

The test method information:

Test method	Description
Black box testing	Conduct security tests from an attacker's perspective externally.
Grey box testing	Conduct security testing on code modules through the scripting tool, observing the internal running status, mining weaknesses.
White box testing	Based on the open source code, non-open source code, to detect whether there are vulnerabilities in programs such as nodes, SDK, etc.

The vulnerability severity level information:

Level	Description
Critical	Critical severity vulnerabilities will have a significant impact on the security of the DeFi project, and it is strongly recommended to fix the critical vulnerabilities.
High	High severity vulnerabilities will affect the normal operation of the DeFi project. It is strongly recommended to fix high-risk vulnerabilities.
Medium	Medium severity vulnerability will affect the operation of the DeFi project. It is recommended to fix medium-risk vulnerabilities.
Low	Low severity vulnerabilities may affect the operation of the DeFi project in certain scenarios. It is suggested that the project team should evaluate and consider whether these vulnerabilities need to be fixed.
Weakness	There are safety risks theoretically, but it is extremely difficult to reproduce in engineering.
Suggestion	There are better practices for coding or architecture.

2 Audit Methodology

The security audit process of SlowMist security team for smart contract includes two steps:

- Smart contract codes are scanned/tested for commonly known and more specific vulnerabilities using automated analysis tools.
- Manual audit of the codes for security issues. The contracts are manually analyzed to look for any potential problems.

Following is the list of commonly known vulnerabilities that was considered during the audit of the smart contract:

Serial Number	Audit Class	Audit Subclass
1	Overflow Audit	-
2	Reentrancy Attack Audit	-
3	Replay Attack Audit	-
4	Flashloan Attack Audit	-
5	Race Conditions Audit	Reordering Attack Audit
6	Permission Vulnerability Audit	Access Control Audit
		Excessive Authority Audit
7	Security Design Audit	External Module Safe Use Audit
		Compiler Version Security Audit
		Hard-coded Address Security Audit
		Fallback Function Safe Use Audit
		Show Coding Security Audit
		Function Return Value Security Audit
		External Call Function Security Audit

Serial Number	Audit Class	Audit Subclass
7	Security Design Audit	Block data Dependence Security Audit
		tx.origin Authentication Security Audit
8	Denial of Service Audit	-
9	Gas Optimization Audit	-
10	Design Logic Audit	-
11	Variable Coverage Vulnerability Audit	-
12	"False Top-up" Vulnerability Audit	-
13	Scoping and Declarations Audit	-
14	Malicious Event Log Audit	-
15	Arithmetic Accuracy Deviation Audit	-
16	Uninitialized Storage Pointer Audit	-

3 Project Overview

3.1 Project Introduction

The StakingPool project is designed to distribute rewards to users based on their contributions to a staking pool. The pool accepts one immutable token for staking and provides rewards in one immutable token. Rewards are provided periodically by an address with the REWARD_PROVIDER role. Users can stake tokens to earn rewards over time, and when they unstake, their tokens enter a bonding period before they can be claimed. Key functions within the contract are restricted by a whitelist, which is managed off-chain through a voting process and set by a script.

3.2 Vulnerability Information

The following is the status of the vulnerabilities found in this audit:

NO	Title	Category	Level	Status
N1	Stake token claim release issue	Design Logic Audit	High	Fixed
N2	Risk of excessive authority	Authority Control Vulnerability Audit	Medium	Acknowledged
N3	Preemptive Initialization	Reordering Vulnerability	Suggestion	Fixed
N4	_accrueRewards rewards accumulation issue	Design Logic Audit	Medium	Fixed
N5	Missing the parameter check	Others	Suggestion	Fixed
N6	getUpdatedUserData function returns incorrect data	Design Logic Audit	Low	Fixed
N7	Token compatibility reminder	Others	Information	Acknowledged
N8	Potential Loop Risk	Design Logic Audit	Information	Acknowledged

4 Code Overview

4.1 Contracts Description

Audit Version:

https://github.com/hyplabs/animoca-staking-pool

commit: be927007da45c8ab6eea0629e0f251d03b37cde7

Fixed Version:

https://github.com/hyplabs/animoca-staking-pool

commit: de03d5bda66ecef6b19cd07c5073e0e23cbcb764

Audit scope:

- ./contract/StakingPool.sol

- ./contract/interfaces/IStakingPool.sol
- ./contract/storage/StakingPoolStorage.sol
- ./contract/types/Types.sol

The main network address of the contract is as follows:

Proxy: https://etherscan.io/address/0xc017EE0481C45905FECFCb4176e4eE16893e4D0b

Implementation: https://etherscan.io/address/0x9dee374a0ffab1a01b2bd1ce66812ee6b503a0e9

4.2 Visibility Description

The SlowMist Security team analyzed the visibility of major contracts during the audit, the result as follows:

StakingPool			
Function Name	Visibility	Mutability	Modifiers
<Constructor>	Public	Can Modify State	-
_authorizeUpgrade	Internal	Can Modify State	onlyRole
__StakingPool_init	External	Can Modify State	initializer
claim	External	Can Modify State	whenNotPaused
claimReleases	External	Can Modify State	whenNotPaused
claimReleases	External	Can Modify State	-
pause	External	Can Modify State	whenNotPaused onlyRole
provideRewards	External	Can Modify State	whenNotPaused onlyRole
setFeeWallet	External	Can Modify State	onlyRole
setProtocolFeeBP	External	Can Modify State	onlyRole
setStakerShareBP	External	Can Modify State	onlyRole
setTreasury	External	Can Modify State	onlyRole
setUnbondingDuration	External	Can Modify State	onlyRole

StakingPool			
setWhitelistRoot	External	Can Modify State	onlyRole
stake	External	Can Modify State	whenNotPaused
unpause	External	Can Modify State	whenPaused onlyRole
unstake	External	Can Modify State	whenNotPaused
getBoundReleaseAmount	External	-	-
getBoundReleases	External	-	-
getClaimableReleaseAmount	External	-	-
getClaimableReleases	External	-	-
getFeeWallet	External	-	-
getGlobalTS	External	-	-
getLatestRewardIndex	External	-	-
getProtocolFeeBP	External	-	-
getReward	External	-	-
getRewards	External	-	-
getStakerShareBP	External	-	-
getTreasury	External	-	-
getUnbondingDuration	External	-	-
getUpdatedGlobalTS	External	-	-
getUpdatedUserData	External	-	-
getUserData	External	-	-
getUserRewards	External	-	-
getWhitelistRoot	External	-	-

StakingPool			
_partitionRewards	Internal	Can Modify State	-
_accrueRewards	Internal	Can Modify State	-
_updateGlobalTS	Internal	Can Modify State	-
_updateAccountTS	Internal	Can Modify State	-
_update	Internal	Can Modify State	-
_getBoundReleases	Internal	-	-
_getClaimableReleases	Internal	-	-
_getFirstBoundReleaseIndex	Internal	-	-

4.3 Vulnerability Summary

[N1] [High] Stake token claim release issue

Category: Design Logic Audit

Content

In the `claimReleases(uint256 releasesToClaim)` function, users can specify the release amount `releasesToClaim` to index and accumulate the stake tokens. The while loop is the function's logic for traversing and accumulating the releases to be extracted. However, in this loop, the `++lastReleaseIndex` operation is first performed to increase the first cumulative index, which will result in the lack of calculation of the accumulation of the `releases[lastReleaseIndex]` amount that enters the loop for the first time. After obtaining the `release` amount, `release.amount` is accumulated in the total release amount that needs to be transferred. Then, the comparison between `release.releaseTime` and block.timestamp is started. Suppose the `release.releaseTime` at this time is greater than the current time block.timestamp, the loop is stopped. However, the `release.amount` of this index has been accumulated in the total release amount, which means that this part of the stake tokens that have not yet reached the unlocking release period are released in advance. This causes issues of missed calculations and early release in this loop logic.

Code location:

contract/StakingPool.sol#L156-L184

```
    function claimReleases(uint256 releasesToClaim)
        external
        returns (uint256 amount)
    {
        …
        while (lastReleaseIndex < finalIndex) {
            ++lastReleaseIndex;
            release = releases[lastReleaseIndex];

            amount += release.amount;

            if (release.releaseTime > block.timestamp) {
                break;
            }
        }

        $.userData[_msgSender()].lastReleaseIndex = lastReleaseIndex;

        IERC20(STAKE_TOKEN).safeTransfer(_msgSender(), amount);

        emit ClaimedReleases(_msgSender(), amount);
    }
```

Solution

It is recommended to follow the release method of `function claimReleases()` and first perform the `if (release.releaseTime > block.timestamp)` judgment in the loop before accumulating the release amount and incrementing the index.

Status

Fixed

[N2] [Medium] Risk of excessive authority

Category: Authority Control Vulnerability Audit

Content

1.In the contract, MANAGER_ROLE can modify the `unbondingDuration` lock period through the setUnbondingDuration function. And WHITELIST_ROOT_SETTER_ROLE can modify the `whitelistRoot` by modifying the setWhitelistRoot function to affect the verification of MerkleProof.

Code location:

contract/StakingPool.sol#L283-L300

```
    function setUnbondingDuration(uint256 duration)
        external
        onlyRole(MANAGER_ROLE)
    {
        Storage.layout().unbondingDuration = duration;

        emit UnbondingDurationSet(duration);
    }

    function setWhitelistRoot(bytes32 root)
        external
        onlyRole(WHITELIST_ROOT_SETTER_ROLE)
    {
        Storage.layout().whitelistRoot = root;

        emit WhitelistRootSet(root);
    }
```

2.The UUPSUpgradeable MANAGER_ROLE relevant authority can upgrade the contract, leading to the risk of over-privileged in this role.

Solution

In the short term, transferring owner ownership to multisig contracts is an effective solution to avoid single-point risk. But in the long run, it is a more reasonable solution to implement a privilege separation strategy and set up multiple privileged roles to manage each privileged function separately. The authority involving user funds should be managed by the community, and the authority involving emergency contract suspension can be managed by the EOA address. This ensures both a quick response to threats and the safety of user funds.

Status

Acknowledged; Response from the project team: A multi-signature wallet will be used to manage permissions, but the use of a timelock will not be mandatory.

[N3] [Suggestion] Preemptive Initialization

Category: Reordering Vulnerability

Content

By calling the __StakingPool_init function to initialize the contract, there is a potential issue that malicious attackers preemptively call the initialize function to initialize.

Code location:

contract/StakingPool.sol#L70-L95

```
    function __StakingPool_init(
        ...
    ) external initializer {
        ...
    }
```

Solution

It is suggested that the initialization operation can be called in the same transaction immediately after the contract is created to avoid being maliciously called by the attacker.

Status

Fixed; Response from the project team: The deployStakingPool.s.sol script will be used for contract deployment and initialization.

[N4] [Medium] _accrueRewards rewards accumulation issue

Category: Design Logic Audit

Content

The _accrueRewards function loops through the reward records (rewards array), accumulates the time weight (accruedTS) according to the user's staked balance (balance) and time span (elapsed), and uses the reward rate (scaledRate) to calculate the reward the user deserves.

However, when there are records with scaledRate = 0 in the rewards array, calculation anomalies will occur. Since lastUpdate is not updated when scaledRate = 0, the next calculation will repeat the calculation of this period. The repeated calculation of the time period causes accruedTS to be accumulated multiple times. When a non-zero scaledRate is encountered later, the incorrectly accumulated accruedTS is used to calculate the reward, resulting in inaccurate calculation of the reward amount.

Code location:

contract/StakingPool.sol#L615-L649, L542-L579, L491-L530

```
    function _accrueRewards(address account) internal {
        Storage.Layout storage $ = Storage.layout();
        UserData memory data = $.userData[account];

        uint256 currentIndex = $.rewards.length - 1;
        uint256 lastIndex = data.lastIndex;
        uint256 balance = balanceOf(account);

        RewardData memory rewardData;
        uint256 accruedTS;
        while (lastIndex != currentIndex) {
            ++lastIndex;
            rewardData = $.rewards[lastIndex];

            uint256 elapsed = rewardData.timestamp - data.lastUpdate;

            accruedTS = balance * elapsed;
            data.accruedTS += accruedTS;

            if (
                rewardData.scaledRate != 0
                    && data.accruedTS > data.deductionTS
            ) {
                data.accruedRewards += (
                    data.accruedTS - data.deductionTS
                ).mulDiv(rewardData.scaledRate, SCALE);
                data.deductionTS = data.accruedTS;
                data.lastUpdate = rewardData.timestamp;
                data.lastIndex = lastIndex;
            }
        }

        $.userData[account] = data;
        _updateAccountTS(account);
    }
```

Solution

It is recommended to update lastUpdate and lastIndex in each loop, regardless of whether scaledRate is 0, to ensure the accuracy of time calculation.

Status

Fixed; In the provideRewards function, a new check has been added to ensure that the amount of rewards provided cannot be equal to 0.

[N5] [Suggestion] Missing the parameter check

Category: Others

Content

1.In the constructor and __StakingPool_init functions, there is a missing zero address check for the address.

Code location:

contract/StakingPool.sol#L53-L60, L70-L95

```
    function __StakingPool_init(
        ...
    ) external initializer {
        ...
        $.feeWallet = feeWallet;
        $.treasury = treasury;
        ...
    }

    constructor(
        address stakeTokenAddress,
        address rewardTokenAddress
    ) {
        _disableInitializers();
        STAKE_TOKEN = stakeTokenAddress;
        REWARD_TOKEN = rewardTokenAddress;
    }
```

2.In the constructor, the STAKE_TOKEN and REWARD_TOKEN addresses are initialized, but there is no check whether the two addresses are the same.

Code location:

contract/StakingPool.sol#L53-L60

```
    constructor(
        address stakeTokenAddress,
        address rewardTokenAddress
    ) {
        _disableInitializers();
        STAKE_TOKEN = stakeTokenAddress;
        REWARD_TOKEN = rewardTokenAddress;
    }
```

3.In the `claimReleases(uint256 releasesToClaim)` function, users can specify the release amount `releasesToClaim` to index and accumulate the stake tokens. However, this function does not check whether finalIndex is greater than the length of the releases array of the user account. If it exceeds, it will cause array access to be out of bounds.

Code location:

contract/StakingPool.sol#L156-L184

```
    function claimReleases(uint256 releasesToClaim)
        external
        returns (uint256 amount)
    {
        ...
        uint256 lastReleaseIndex =
            $.userData[_msgSender()].lastReleaseIndex;
        uint256 finalIndex = lastReleaseIndex + releasesToClaim;
        ...
    }
```

4.In the __StakingPool_init function, there is a lack of a check to enforce the maximum upper limit on the protocolFeeBP parameter.

Code location:

contract/StakingPool.sol#L53-L60, L70-L95

```
    function __StakingPool_init(
        ...
    ) external initializer {
        ...
        $.protocolFeeBP = protocolFeeBP;
        ...
    }
```

Solution

1.It is recommended to add the 0 address check.

2.It is recommended to add the check that STAKE_TOKEN and REWARD_TOKEN addresses are not equal

3.It is recommended to add `require(lastReleaseIndex + releasesToClaim <= releases.length);` array bounds check

4.It is recommended to implement a check to enforce the maximum upper limit on the protocolFeeBP parameter.

Status

Fixed

[N6] [Low] getUpdatedUserData function returns incorrect data

Category: Design Logic Audit

Content

When the user calls the getUpdatedUserData function, they will receive the user data that is about to be updated. However, the value of lastUpdate in the returned updated user data is the time of the most recent reward distribution, not the current timestamp; whereas, at the end of the _accrueRewards function, it will call the _updateAccountTS function to update the lastUpdate parameter in the user data to block.timestamp. This conflict will confuse the user or the frontend after calling the getUpdatedUserData function.

Code location:

contract/StakingPool.sol#L491-530

```
    function getUpdatedUserData(address account)
        external
        view
        returns (UserData memory updatedData)
    {
        ...

        while (lastIndex != currentIndex) {
            ...

            if (
                rewardData.scaledRate != 0
                    && data.accruedTS > data.deductionTS
            ) {
                data.accruedRewards += (
                    data.accruedTS - data.deductionTS
                ).mulDiv(rewardData.scaledRate, SCALE);
                data.deductionTS = data.accruedTS;
                data.lastUpdate = rewardData.timestamp;
                data.lastIndex = lastIndex;
            }
        }

        data.accruedTS +=
            balanceOf(account) * (block.timestamp - data.lastUpdate);
        updatedData = data;
    }
```

Solution

It is recommended to set the value of the lastUpdate parameter in the returned updatedData to block.timestamp.

Status

Fixed

[N7] [Information] Token compatibility reminder

Category: Others

Content

In the StakingPool contract, users can stake/unstake their tokens through the stake, unstake and claimReleases functions after unbondingDuration. It will directly record the amount parameter passed by the minted token of the StakingPool, and transfer the stake tokens to the contract through the safeTransferFrom function. Similarly, the unstake function will first burn the minted tokens corresponding to the 1:1 amount and then transfer the tokens back to the user through the claimReleases function or directly through safeTransfer after the lock ends. However, suppose this STAKE_TOKEN is a deflationary token. In that case, each transfer will make the STAKE_TOKEN in the contract less and less, so that the user will not be able to withdraw the previously deposited amount of tokens.

Code location:

contract/StakingPool.sol#L150, L181, L324, L356

```
    IERC20(STAKE_TOKEN).safeTransferFrom(
            _msgSender(), address(this), amount
        );
    IERC20(STAKE_TOKEN).safeTransfer(_msgSender(), amount);
```

Solution

It is recommended to record the difference before and after the user's transfer as the actual amount or not use the deflationary tokens as the STAKE_TOKEN.

Status

Acknowledged; Response from the project team: if the intention was for any pool with any token combination to be launched, it would be implemented certainly. Given that the launch is on ETH main net and the STAKE tokens are known *NOT* to be taxed/deflationary, it shall be omitted.

[N8] [Information] Potential Loop Risk

Category: Design Logic Audit

Content

In the _accrueRewards function, the amount of rewards accumulated by the user is calculated using a while loop to iterate through the rewards array, which will run from the index of the last reward the user claimed to the index of the latest reward distribution. If a user does not perform any new actions (such as redeeming, minting, transferring, or claiming) for a long time after their last reward calculation, and if many rounds of reward distribution occur during this period, this could result in a DoS risk due to the large size of the array being iterated over by the while loop.

Code location:

contract/StakingPool.sol#L670-690

```
    function _accrueRewards(address account) internal {
        ...

        while (lastIndex != currentIndex) {
            ++lastIndex;
            rewardData = $.rewards[lastIndex];

            uint256 elapsed = rewardData.timestamp - data.lastUpdate;

            accruedTS = balance * elapsed;
            data.accruedTS += accruedTS;

            if (
                rewardData.scaledRate != 0
                    && data.accruedTS > data.deductionTS
            ) {
                data.accruedRewards += (
                    data.accruedTS - data.deductionTS
                ).mulDiv(rewardData.scaledRate, SCALE);
                data.deductionTS = data.accruedTS;
                data.lastUpdate = rewardData.timestamp;
                data.lastIndex = lastIndex;
            }
        }

        ...
    }
```

Solution

N/A

Status

Acknowledged

5 Audit Result

Audit Number	Audit Team	Audit Date	Audit Result
0X002410300001	SlowMist Security Team	2024.10.25 - 2024.10.29	Medium Risk

Summary conclusion: The SlowMist security team uses a manual and SlowMist team's analysis tool to audit the project, during the audit work we found 1 high risk, 2 medium risk, 1 low risk, 2 suggestion and 2 information. All the findings were fixed or acknowledged. The code has been deployed to the mainnet. The current risk level is temporarily rated as medium simply because the centralized control permissions have not yet been managed.

6 Statement

SlowMist issues this report with reference to the facts that have occurred or existed before the issuance of this report, and only assumes corresponding responsibility based on these.

For the facts that occurred or existed after the issuance, SlowMist is not able to judge the security status of this project, and is not responsible for them. The security audit analysis and other contents of this report are based on the documents and materials provided to SlowMist by the information provider till the date of the insurance report (referred to as "provided information"). SlowMist assumes: The information provided is not missing, tampered with, deleted or concealed. If the information provided is missing, tampered with, deleted, concealed, or inconsistent with the actual situation, the SlowMist shall not be liable for any loss or adverse effect resulting therefrom. SlowMist only conducts the agreed security audit on the security situation of the project and issues this report. SlowMist is not responsible for the background and other conditions of the project.