StakingPool

Smart Contract Audit Report
Prepared for GuildFi

Date Issued:	Dec 17, 2021
Project ID:	AUDIT2021052
Version:	v1.0
Confidentiality Level:	Public

inspeX
CYBERSECURITY PROFESSIONAL SERVICE

Report Information

Project ID	AUDIT2021052
Version	v1.0
Client	GuildFi
Project	StakingPool
Auditor(s)	Pongsakorn Sommalai Puttimet Thammasaeng
Author	Pongsakorn Sommalai
Reviewer	Suvicha Buakhom
Confidentiality Level	Public

Version History

Version	Date	Description	Author(s)
1.0	Dec 17, 2021	Full report	Pongsakorn Sommalai

Contact Information

Company	Inspex
Phone	(+66) 90 888 7186
Telegram	t.me/inspexco
Email	audit@inspex.co

Table of Contents

1. Executive Summary

As requested by GuildFi, Inspex team conducted an audit to verify the security posture of the StakingPool smart contracts between Dec 14, 2021 and Dec 16, 2021. During the audit, Inspex team examined all smart contracts and the overall operation within the scope to understand the overview of StakingPool smart contracts. Static code analysis, dynamic analysis, and manual review were done in conjunction to identify smart contract vulnerabilities together with technical & business logic flaws that may be exposed to the potential risk of the platform and the ecosystem. Practical recommendations are provided according to each vulnerability found and should be followed to remediate the issue.

1.1. Audit Result

In the initial audit, Inspex found 2 high, 1 medium, 1 low, 2 very low-severity issues. With the project team's prompt response in resolving the issues found by Inspex, all issues were resolved in the reassessment. Therefore, Inspex trusts that StakingPool smart contracts have high-level protections in place to be safe from most attacks.

1.2. Disclaimer

This security audit is not produced to supplant any other type of assessment and does not guarantee the discovery of all security vulnerabilities within the scope of the assessment. However, we warrant that this audit is conducted with goodwill, professional approach, and competence. Since an assessment from one single party cannot be confirmed to cover all possible issues within the smart contract(s), Inspex suggests conducting multiple independent assessments to minimize the risks. Lastly, nothing contained in this audit report should be considered as investment advice.

2. Project Overview

2.1. Project Introduction

GuildFi is an interconnected ecosystem of games, communities, and NFT assets for maximizing yields and enabling interoperability across the Metaverse.

The Staking Pool contracts are the GuildFi platform's time-weighted staking pool. It is used for distributing the reward token to the user. By locking their token in this contract, the users can amplify the reward related to locking duration.

Scope Information:

Project Name	StakingPool
Website	https://guildfi.com/
Smart Contract Type	Ethereum Smart Contract
Chain	Ethereum
Programming Language	Solidity

Audit Information:

Audit Method	Whitebox
Audit Date	Dec 14, 2021 - Dec 16, 2021
Reassessment Date	Dec 17, 2021

The audit method can be categorized into two types depending on the assessment targets provided:

1. **Whitebox**: The complete source code of the smart contracts are provided for the assessment.
2. **Blackbox**: Only the bytecodes of the smart contracts are provided for the assessment.

2.2. Scope

The following smart contracts were audited and reassessed by Inspex in detail:

Initial Audit: (Commit: 6e802a5a18545756ec6713a8c68c426954a2f37a)

Contract	Location (URL)
StakingPool	https://github.com/GuildFi/GF-STAKING/blob/6e802a5a18/contracts/StakingPool.sol
StakingPoolManager	https://github.com/GuildFi/GF-STAKING/blob/6e802a5a18/contracts/StakingPoolManager.sol
View	https://github.com/GuildFi/GF-STAKING/blob/6e802a5a18/contracts/View.sol
TokenSaver	https://github.com/GuildFi/GF-STAKING/blob/6e802a5a18/contracts/base/TokenSaver.sol

Reassessment: (Commit: 13638ab89c094c7d2c346070d67c8942348668be)

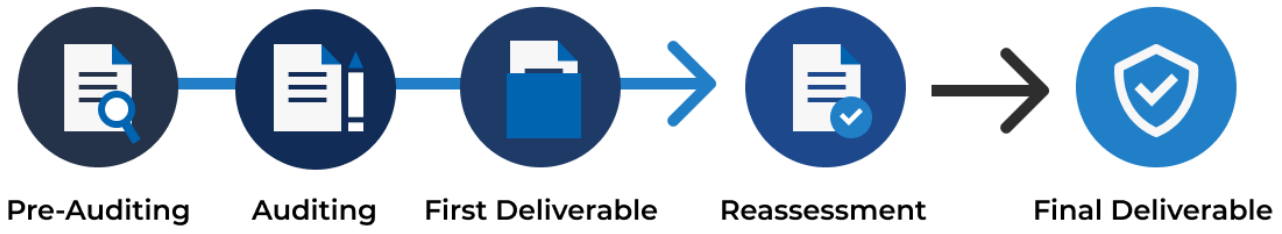
Contract	Location (URL)
StakingPool	https://github.com/GuildFi/GF-STAKING/blob/13638ab89c/contracts/StakingPool.sol
StakingPoolManager	https://github.com/GuildFi/GF-STAKING/blob/13638ab89c/contracts/StakingPoolManager.sol
View	https://github.com/GuildFi/GF-STAKING/blob/13638ab89c/contracts/View.sol
TokenSaver	https://github.com/GuildFi/GF-STAKING/blob/13638ab89c/contracts/base/TokenSaver.sol

The assessment scope covers only the in-scope smart contracts and the smart contracts that they inherit from.

3. Methodology

Inspex conducts the following procedure to enhance the security level of our clients' smart contracts:

1. **Pre-Auditing**: Getting to understand the overall operations of the related smart contracts, checking for readiness, and preparing for the auditing
2. **Auditing**: Inspecting the smart contracts using automated analysis tools and manual analysis by a team of professionals
3. **First Deliverable and Consulting**: Delivering a preliminary report on the findings with suggestions on how to remediate those issues and providing consultation
4. **Reassessment**: Verifying the status of the issues and whether there are any other complications in the fixes applied
5. **Final Deliverable**: Providing a full report with the detailed status of each issue



3.1. Test Categories

Inspex smart contract auditing methodology consists of both automated testing with scanning tools and manual testing by experienced testers. We have categorized the tests into 3 categories as follows:

1. **General Smart Contract Vulnerability (General)** - Smart contracts are analyzed automatically using static code analysis tools for general smart contract coding bugs, which are then verified manually to remove all false positives generated.
2. **Advanced Smart Contract Vulnerability (Advanced)** - The workflow, logic, and the actual behavior of the smart contracts are manually analyzed in-depth to determine any flaws that can cause technical or business damage to the smart contracts or the users of the smart contracts.
3. **Smart Contract Best Practice (Best Practice)** - The code of smart contracts is then analyzed from the development perspective, providing suggestions to improve the overall code quality using standardized best practices.

3.2. Audit Items

The following audit items were checked during the auditing activity.

General
Reentrancy Attack
Integer Overflows and Underflows
Unchecked Return Values for Low-Level Calls
Bad Randomness
Transaction Ordering Dependence
Time Manipulation
Short Address Attack
Outdated Compiler Version
Use of Known Vulnerable Component
Deprecated Solidity Features
Use of Deprecated Component
Loop with High Gas Consumption
Unauthorized Self-destruct
Redundant Fallback Function
Insufficient Logging for Privileged Functions
Invoking of Unreliable Smart Contract
Use of Upgradable Contract Design
Advanced
Business Logic Flaw
Ownership Takeover
Broken Access Control
Broken Authentication
Improper Kill-Switch Mechanism

Improper Front-end Integration
Insecure Smart Contract Initiation
Denial of Service
Improper Oracle Usage
Memory Corruption
Best Practice
Use of Variadic Byte Array
Implicit Compiler Version
Implicit Visibility Level
Implicit Type Inference
Function Declaration Inconsistency
Token API Violation
Best Practices Violation

3.3. Risk Rating

OWASP Risk Rating Methodology[1] is used to determine the severity of each issue with the following criteria:

- **Likelihood**: a measure of how likely this vulnerability is to be uncovered and exploited by an attacker.
- **Impact**: a measure of the damage caused by a successful attack

Both likelihood and impact can be categorized into three levels: **Low**, **Medium**, and **High**.

Severity is the overall risk of the issue. It can be categorized into five levels: **Very Low**, **Low**, **Medium**, **High**, and **Critical**. It is calculated from the combination of likelihood and impact factors using the matrix below. The severity of findings with no likelihood or impact would be categorized as **Info**.

Impact \ Likelihood	Low	Medium	High
Low	Very Low	Low	Medium
Medium	Low	Medium	High
High	Medium	High	Critical

4. Summary of Findings

From the assessments, Inspex has found 6 issues in three categories. The following chart shows the number of the issues categorized into three categories: **General**, **Advanced**, and **Best Practice**.

The statuses of the issues are defined as follows:

Status	Description
Resolved	The issue has been resolved and has no further complications.
Resolved *	The issue has been resolved with mitigations and clarifications. For the clarification or mitigation detail, please refer to Chapter 5.
Acknowledged	The issue's risk has been acknowledged and accepted.
No Security Impact	The best practice recommendation has been acknowledged.

The information and status of each issue can be found in the following table:

ID	Title	Category	Severity	Status
IDX-001	Uninitialized lastRewardBlock Contract State	Advanced	**High**	Resolved
IDX-002	Improper Token Transfer in StakingPool	Advanced	**High**	Resolved
IDX-003	Centralized Control of State Variable	General	**Medium**	Resolved
IDX-004	Denial of Service with Huge Array Size	General	**Low**	Resolved
IDX-005	Outdated Compiler Version	General	**Very Low**	Resolved
IDX-006	Improper Access Control in StakingPoolManager	Advanced	**Very Low**	Resolved

* The mitigations or clarifications by GuildFi can be found in Chapter 5.

5. Detailed Findings Information

5.1. Uninitialized lastRewardBlock Contract State

ID	IDX-001
Target	StakingPoolManager
Category	Advanced Smart Contract Vulnerability
CWE	CWE-908: Use of Uninitialized Resource
Risk	**Severity:** High **Impact:** Medium The miscalculation of the blockPassed variable in `distributeRewards()` function leads to higher reward distribution than expected. **Likelihood:** High This issue will happen on the first execution of `distributeRewards()` function without any condition.
Status	Resolved This issue has already been resolved as recommended in the commit 13638ab89c094c7d2c346070d67c8942348668be.

5.1.1. Description

In the `distributeRewards()` function, the `blockPassed` variable is calculated by `getMuliplier()` function that use `lastRewardBlock` as `_from` and `block.number` as `_to` parameters as shown below:

StakingManager.sol

```
138 function distributeRewards() public onlyRewardDistributor {
139   uint256 blockPassed = getMultiplier(lastRewardBlock, block.number,
    rewardEndBlock);
```

When the `distributeRewards()` function is executed on the first time, the `lastRewardBlock` state is not initiated. Thus, the `lastRewardBlock` state is set to `0` by default, and the return value of `getMultiplier()` function is calculated by subtracting the `_to` parameter with `_from` parameter in line 133. Since the `_from` is `0` and `_to` is `block.number`, the value of the `blockPassed` variable returned from the `getMultiplier()` function will be `block.number` as shown in the following source code.

StakingManager.sol

```
123 /// @notice Return reward multiplier over the given _from to _to block.
124 function getMultiplier(
125   uint256 _from,
126   uint256 _to,
```

```
127     uint256 _endBlock
128   ) public pure returns (uint256) {
129     if ((_from >= _endBlock) || (_from > _to)) {
130       return 0;
131     }
132     if (_to <= _endBlock) {
133       return _to - _from;
134     }
135     return _endBlock - _from;
136   }
```

The `blockPassed` variable from the previous step is used to calculate the `totalRewardAmount` variable that will be used for distributing the reward to each pool as shown below:

StakingManager.sol

```
145   uint256 totalRewardAmount = rewardPerBlock * blockPassed;
146   lastRewardBlock = block.number >= rewardEndBlock ? rewardEndBlock :
      block.number;
147
148   // return if pool length == 0
149   if (pools.length == 0) {
150     return;
151   }
152
153   // return if accrued rewards == 0
154   if (totalRewardAmount == 0) {
155     return;
156   }
157
158   reward.safeTransferFrom(rewardSource, address(this), totalRewardAmount);
159
160   for (uint256 i = 0; i < pools.length; i++) {
161     Pool memory pool = pools[i];
162     uint256 poolRewardAmount = (totalRewardAmount * pool.weight) / totalWeight;
163     // Ignore tx failing to prevent a single pool from halting reward
      distribution
164     address(pool.poolContract).call(
165       abi.encodeWithSelector(pool.poolContract.distributeRewards.selector,
      poolRewardAmount)
166     );
167   }
```

As a result, because the `blockPassed` variable is incorrectly calculated to `block.number` on the first execution of the `distributeRewards()` function, the `distributeRewards()` function will distribute more reward to each pool than expected.

5.1.2. Remediation

Inspex suggests adding `_rewardStartBlock` in the constructor to initiate the `lastRewardBlock` state in order to set the first block that reward will be distributed as shown in the following example:

StakingPoolManager.sol

```
52  constructor(
53    address _reward,
54    address _rewardSource,
55    uint256 _rewardEndBlock,
56    uint256 _rewardStartBlock
57  ) {
58    require(_reward != address(0), "bad _reward");
59    require(_rewardSource != address(0), "bad _rewardSource");
60    require(_rewardStartBlock > block.number && _rewardStartBlock <
    _rewardEndBlock, "! bad period");
61    reward = IERC20(_reward);
62    rewardSource = _rewardSource;
63    rewardEndBlock = _rewardEndBlock;
64    lastRewardBlock = _rewardStartBlock;
65  }
```

5.2. Improper Token Transfer in StakingPool

ID	IDX-002
Target	TokenSaver
Category	Advanced Smart Contract Vulnerability
CWE	CWE-912: Hidden Functionality
Risk	**Severity: High** **Impact: High** The user with the`TOKEN_SAVER_ROLE` role can drain any token from the `StakingPool` contract. **Likelihood: Medium** It is likely that the contract owner will have the `TOKEN_SAVER_ROLE` role.
Status	Resolved This issue has already been resolved as recommended in the commit 13638ab89c094c7d2c346070d67c8942348668be.

5.2.1. Description

The `StakingPool` contract inherits from the `BasePool` contract in line 11 as shown below:

StakingPool.sol

```
11 contract StakingPool is BasePool, ITimeLockPool {
12   using Math for uint256;
13   using SafeERC20 for IERC20;
```

Then, the `BasePool` abstract contract also inherits from the `TokenSaver` contract as follows:

BasePool.sol

```
15 abstract contract BasePool is ERC20Votes, AbstractRewards, IBasePool,
   TokenSaver {
16   using SafeERC20 for IERC20;
17   using SafeCast for uint256;
18   using SafeCast for int256;
```

Additionally, the `TokenSaver` contract contains the `saveToken()` function that allows any wallet with the `TOKEN_SAVER_ROLE` role to transfer any token from this contract to any address as shown in the following source code:

TokenSaver.sol

```
24 function saveToken(
25   address _token,
26   address _receiver,
27   uint256 _amount
28 ) external onlyTokenSaver {
29   IERC20(_token).safeTransfer(_receiver, _amount);
30   emit TokenSaved(_msgSender(), _receiver, _token, _amount);
31 }
```

The result indicates that any wallet with the `TOKEN_SAVER_ROLE` role can drain other users' reward or staked token from this contract.

5.2.2. Remediation

Inspex suggests removing this function from the `StakingPool` contract by removing the `TokenSaver` contract from the `BasePool` parent contract as shown in the following example:

BasePool.sol

```
15 abstract contract BasePool is ERC20Votes, AbstractRewards, IBasePool {
16   using SafeERC20 for IERC20;
17   using SafeCast for uint256;
18   using SafeCast for int256;
```

5.3. Centralized Control of State Variable

ID	IDX-003
Target	StakingPoolManager
Category	General Smart Contract Vulnerability
CWE	CWE-284: Improper Access Control
Risk	**Severity:** Medium **Impact:** Medium The controlling authorities can change the critical state variables to gain additional profit. Thus, it is unfair to the other users. **Likelihood:** Medium There is nothing to restrict the changes from being done; however, this action can only be done by the contract owner.
Status	Resolved This issue has already been resolved as recommended in the commit 13638ab89c094c7d2c346070d67c8942348668be.

5.3.1. Description

Critical state variables can be updated any time by the controlling authorities. Changes in these variables can cause impacts to the users, so the users should accept or be notified before these changes are effective.

However, there is currently no constraint to prevent the authorities from modifying these variables without notifying the users.

The controllable privileged state update functions are as follows:

File	Target	Function	Modifier
StakingPoolManager.sol (L:64)	StakingPoolManager	addPool()	onlyGov
StakingPoolManager.sol (L:82)	StakingPoolManager	removePool()	onlyGov
StakingPoolManager.sol (L:98)	StakingPoolManager	adjustWeight()	onlyGov
StakingPoolManager.sol (L:111)	StakingPoolManager	setRewardEndBlock()	onlyGov
StakingPoolManager.sol (L:117)	StakingPoolManager	setRewardPerBlock()	onlyGov

5.3.2. Remediation

In the ideal case, the critical state variables should not be modifiable to keep the integrity of the smart contract. However, if modifications are needed, Inspex suggests limiting the use of these functions via the following options:

- Implementing a community-run governance to control the use of these functions
- Using a Timelock contract to delay the changes for a reasonable amount of time

5.4. Denial of Service with Huge Array Size

ID	IDX-004
Target	StakingPool
Category	General Smart Contract Vulnerability
CWE	CWE-400: Uncontrolled Resource Consumption
Risk	**Severity:** Low **Impact:** Medium The `getTotalDeposit()` and `getDepositsOf()` functions become unusable. **Likelihood:** Low It is very unlikely that the array size of the `depositsOf` state will be large enough to make the `getTotalDeposit()` and `getDepositsOf()` functions become unusable.
Status	Resolved This issue has already been resolved as recommended in the commit 13638ab89c094c7d2c346070d67c8942348668be.

5.4.1. Description

For the case of `getTotalDeposit()` function, looping is performed over the `depositsOf[_account]` state array size in line 76 in order to calculate the total deposit value.

StakingPool.sol

```
73 function getTotalDeposit(address _account) public view returns (uint256) {
74   uint256 total;
75   for (uint256 i = 0; i < depositsOf[_account].length; i++) {
76     total += depositsOf[_account][i].amount;
77   }
78
79   return total;
80 }
```

Since the `depositsOf[_account]` is a dynamic array, when the size of `depositsOf[_account]` is larger than approximately 1,500 items, the `getTotalDeposit()` function will become unusable.

For the case of `getDepositsOf()` function, although the Solidity code is not using any loop, the EVM accesses the state to get each item in order to build the `depositsOf[_account]` array as the return value of `getDepositsOf()` function in line 83 as shown below:

StakingPool.sol

```
82 function getDepositsOf(address _account) public view returns (Deposit[] memory)
   {
83   return depositsOf[_account];
84 }
```

When the size of `depositsOf[_account]` is larger than approximately 1,500 items, the `getTotalDeposit()` function will also become unusable.

5.4.2. Remediation

Case 1: The `getDepositsOf()` function

Inspex suggests implementing the pagination mechanism to limit the output array size and gas usage, for example:

StakingPool.sol

```
82 function getDepositsOf(address _account, uint256 skip, uint256 limit) public
   view returns (Deposit[] memory) {
83   Deposit[] memory _depositsOf = new Deposit[](limit);
84   uint256 depositsOfLength = depositsOf[_account].length;
85
86   if (skip >= depositsOfLength ) return _depositsOf;
87
88   for (uint256 i = skip; i < (skip + limit).min(depositsOfLength); i++) {
89     _depositsOf[i-skip] = depositsOf[_account][i];
90   }
91
92   return _depositsOf;
93 }
```

Case 2: The `getTotalDeposit()` function

First, Inspex recommends creating the `totalDeposit` state with `mapping(address => uint256)` data type as shown below:

StakingPool.sol

```
20 mapping(address => uint256) public totalDeposit;
```

Then, we suggests adding `totalDeposit` state with input amount in `deposit()` function and subtracting it in `withdraw()` function as follows:

StakingPool.sol

```
47  function deposit(
48    uint256 _amount,
49    uint256 _duration,
50    address _receiver
51  ) external override {
52    require(_amount > 0, "bad _amount");
53    // Don't allow locking > maxLockDuration
54    uint256 duration = _duration.min(maxLockDuration);
55    // Enforce min lockup duration to prevent flash loan or MEV transaction
    ordering
56    duration = duration.max(MIN_LOCK_DURATION);
57
58    depositToken.safeTransferFrom(_msgSender(), address(this), _amount);
59
60    depositsOf[_receiver].push(
61      Deposit({ amount: _amount, start: uint64(block.timestamp), end:
    uint64(block.timestamp) + uint64(duration) })
62    );
63    totalDeposit[_receiver] += _amount;
64
65    uint256 mintAmount = (_amount * getMultiplier(duration)) / 1e18;
66
67    _mint(_receiver, mintAmount);
68    emit Deposited(_amount, duration, _receiver, _msgSender());
69  }
```

StakingPool.sol

```
104 function withdraw(uint256 _depositId, address _receiver) external {
105   require(_depositId < depositsOf[_msgSender()].length, "!exist");
106   Deposit memory userDeposit = depositsOf[_msgSender()][_depositId];
107   require(block.timestamp >= userDeposit.end, "too soon");
108
109   // No risk of wrapping around on casting to uint256 since deposit end always
    > deposit start and types are 64 bits
110   uint256 shareAmount = (userDeposit.amount *
    getMultiplier(uint256(userDeposit.end - userDeposit.start))) / 1e18;
111   // remove Deposit
112   totalDeposit[_msgSender()] -= _amount;
113   depositsOf[_msgSender()][_depositId] =
    depositsOf[_msgSender()][depositsOf[_msgSender()].length - 1];
114   depositsOf[_msgSender()].pop();
115
116   // burn pool shares
117   _burn(_msgSender(), shareAmount);
118
```

```
119   // return tokens
120   depositToken.safeTransfer(_receiver, userDeposit.amount);
121   emit Withdrawn(_depositId, _receiver, _msgSender(), userDeposit.amount);
122 }
123
124
```

Finally, in `getTotalDeposit()` function, it is recommended to directly return the `totalDeposit` state as shown in the following example:

StakingPool.sol

```
74 function getTotalDeposit(address _account) public view returns (uint256) {
75   return totalDeposit[_account];
76 }
```

5.5. Outdated Compiler Version

ID	IDX-005
Target	StakingPoolManager StakingPool View TokenSaver
Category	General Smart Contract Vulnerability
CWE	CWE-1104: Use of Unmaintained Third Party Components
Risk	**Severity:** Very Low **Impact:** Low From the list of known Solidity bugs, direct impact cannot be caused from those bugs themselves. **Likelihood:** Low From the list of known Solidity bugs, it is very unlikely that those bugs would affect these smart contracts.
Status	**Resolved** This issue has already been resolved as recommended in the commit 13638ab89c094c7d2c346070d67c8942348668be.

5.5.1. Description

The Solidity compiler version specified in the smart contracts was outdated. This version has publicly known inherent bugs[2] that may potentially be used to cause damage to the smart contracts or the users of the smart contracts.

StakingPool.sol

```
1 // SPDX-License-Identifier: MIT
2 pragma solidity 0.8.7;
```

StakingPoolManager.sol

```
1 // SPDX-License-Identifier: MIT
2 pragma solidity 0.8.7;
```

View.sol

```
1 // SPDX-License-Identifier: MIT
2 pragma solidity 0.8.7;
```

TokenSaver.sol

```
1 // SPDX-License-Identifier: MIT
2 pragma solidity 0.8.7;
```

5.5.2. Remediation

Inspex suggests upgrading the Solidity compiler to the latest stable version.

During the audit activity, the latest stable version of Solidity compiler in this major is version 0.8.10.

5.6. Improper Access Control in StakingPoolManager

ID	IDX-006
Target	StakingPoolManger
Category	Advanced Smart Contract Vulnerability
CWE	CWE-284: Improper Access Control
Risk	**Severity:** Very Low **Impact:** Low The access control is designed in such a way that the `GOV_ROLE` role needs the `REWARD_DISTRIBUTOR_ROLE` role to make all functionalities work correctly. This breaks the separation of duties (SoD) security concept. **Likelihood:** Low It is very unlikely that there is a user that has the `GOV_ROLE` role that does not require the `REWARD_DISTRIBUTOR_ROLE` role.
Status	Resolved This issue has already been resolved as recommended in the commit 13638ab89c094c7d2c346070d67c8942348668be.

5.6.1. Description

In the StakingPoolManger contract, the `distributeRewards()` function has the `onlyRewardDistributor` modifier in line 138. It allows any address with `REWARD_DISTRIBUTOR_ROLE` role to execute this function as shown below:

StakingPoolManager.sol

```
138 function distributeRewards() public onlyRewardDistributor {
139   uint256 blockPassed = getMultiplier(lastRewardBlock, block.number,
    rewardEndBlock);
```

According to the design of the reward distribution, the reward must be distributed before changing the state of the reward configuration.

For example, when the `setRewardPerBlock()` is called, the `distributeRewards()` function will be called before changing the `rewardPerBlock` state at line 118-119 as shown in the following source code:

StakingPoolManager.sol

```
117 function setRewardPerBlock(uint256 _rewardPerBlock) external onlyGov {
118   distributeRewards();
119   rewardPerBlock = _rewardPerBlock;
120   emit SetRewardsPerBlock(_rewardPerBlock);
```

```
121 }
```

Therefore, when the `setRewardPerBlock()` function is called, the `distributeRewards()` function will also be executed. Thus, both `onlyGov` and `onlyRewardDistributor` modifiers are executed, so the user needs both `GOV_ROLE` and `REWARD_DISTRIBUTOR_ROLE` roles to execute the `setRewardPerBlock()` function.

As a result, this breaks the separation of duties (SoD) security concept. The `GOV_ROLE` should not be allowed to execute the functions of the `REWARD_DISTRIBUTOR_ROLE` role.

The following functions are affected by this issue:

File	Contract	Function	Modifier
StakingPoolManager.sol (L:64)	StakingPoolManager	addPool()	onlyGov
StakingPoolManager.sol (L:82)	StakingPoolManager	removePool()	onlyGov
StakingPoolManager.sol (L:98)	StakingPoolManager	adjustWeight()	onlyGov
StakingPoolManager.sol (L:117)	StakingPoolManager	setRewardPerBlock()	onlyGov

5.6.2. Remediation

Inspex suggests separating functionalities of the `GOV_ROLE` and `REWARD_DISTRIBUTOR_ROLE` roles from each other.

In this case, first, we recommends changing the visibility of `distributeRewards()` function from `public` to `internal`, changing its name to `_distributeRewards()`, and removing `onlyRewardDistributor` modifier as shown below:

StakingPoolManager.sol

```
138 function _distributeRewards() internal {
139   uint256 blockPassed = getMultiplier(lastRewardBlock, block.number,
    rewardEndBlock);
140
141   if (blockPassed == 0) {
142     return;
143   }
```

Then, we suggests implementing the new `distributeRewards()` function with `external` visibility in order to directly call `_distributeRewards()` function as follows:

StakingPoolManager.sol

```
183 function distributeRewards() external onlyRewardDistributor {
184   _distributeRewards();
185 }
```

Finally, it is recommended to replace all `distributeRewards()` function callings to the `_distributeRewards()` function as shown in the following example:

StakingPoolManager.sol

```
117 function setRewardPerBlock(uint256 _rewardPerBlock) external onlyGov {
118   _distributeRewards();
119   rewardPerBlock = _rewardPerBlock;
120   emit SetRewardsPerBlock(_rewardPerBlock);
121 }
```

6. Appendix

6.1. About Inspex

Inspex is formed by a team of cybersecurity experts highly experienced in various fields of cybersecurity. We provide blockchain and smart contract professional services at the highest quality to enhance the security of our clients and the overall blockchain ecosystem.

Follow Us On:

Website	https://inspex.co
Twitter	@InspexCo
Facebook	https://www.facebook.com/InspexCo
Telegram	@inspex_announcement

6.2. References

[1] "OWASP Risk Rating Methodology." [Online]. Available: https://owasp.org/www-community/OWASP_Risk_Rating_Methodology. [Accessed: 08-May-2021]

[2] "List of Known Bugs" [Online]. Available: https://docs.soliditylang.org/en/v0.8.7/bugs.html. [Accessed: 17-Dec-2021]

inspeX

CYBERSECURITY PROFESSIONAL SERVICE