Smart Contract Security Audit Report

Table Of Contents

1 Executive Summary

On 2024.10.29, the SlowMist security team received the FLock team's security audit application for FLock Phase1, 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

FLock is a blockchain-based platform for decentralised AI. FLock eliminates obstacles that prevent active participation in AI systems, empowering communities to contribute models, data, or computing computing resources in a modular and decentralised way.

This audit covers the CoHosting smart contracts, which primarily consists of five modules: mini pool, pool manager, stake info, task manager, and flock token. The mini pool module provides token delegation and reward distribution services, while the pool manager module is responsible for creating mini pools. The stake info module records token staking information, and the task manager module manages staking services for tasks. The flock token is the protocol's native token.

3.2 Vulnerability Information

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

NO	Title	Category	Level	Status
N1	Potential risks of signature malleability	Replay Vulnerability	Low	Acknowledged
N2	Risk of whitelisted users front-running	Reordering Vulnerability	Critical	Fixed
N3	Risks of self-minting arbitrary amounts of tokens	Design Logic Audit	Critical	Acknowledged
N4	Risk of Signature Replay	Replay Vulnerability	Critical	Fixed
N5	Potential risk of not checking whether the signer is a 0 address	Design Logic Audit	Low	Fixed
N6	Using incorrect role for reward calculation	Design Logic Audit	Critical	Fixed
N7	Potential risk of funds being locked for task id 0	Design Logic Audit	High	Fixed
N8	Some Task parameters are not used	Others	Suggestion	Acknowledged
N9	Unable to guarantee that the mintDailyReward operation is performed only once per day	Others	Suggestion	Acknowledged
N10	The total staked amount is not deducted when removing user stake	Design Logic Audit	Suggestion	Acknowledged
N11	Upload reward does not check if the user belongs to the task	Design Logic Audit	Medium	Acknowledged
N12	Did not check if the sum of rewardPercentages of	Others	Low	Acknowledged

NO	Title	Category	Level	Status
	reward distribution is as expected			
N13	Risk of part of the reward being locked	Design Logic Audit	High	Fixed
N14	Redundant code	Others	Suggestion	Fixed
N15	Unable to withdraw withdrawal fee	Design Logic Audit	High	Acknowledged
N16	Risks of calling markTaskAsFinished directly to complete the task	Design Logic Audit	Medium	Fixed
N17	Risks of excessive privilege	Authority Control Vulnerability Audit	Medium	Fixed

4 Code Overview

4.1 Contracts Description

Audit Version:

https://github.com/FLock-io/co-hosting-smart-contracts

commit: 5e3d4c5c5d888a365b7439d216522a0d6d97b4e5

Fixed Version:

https://github.com/FLock-io/co-hosting-smart-contracts

commit: a334fe57170c094afa7050ddec0afde3b81ea9fd

Audit Scope:

```
./contracts
├── AdminUpgradeable.sol
├── FlockMiniPool.sol
├── FlockPoolManagerUpgradeable.sol
├── FlockStakeInfoUpgradeable.sol
├── FlockTaskManagerUpgradeable.sol
├── FlockTokenUpgradeable.sol
```

```
├── config
└── interfaces
```

The main network address of the contract is as follows:

The code was not deployed to the mainnet.

4.2 Visibility Description

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

AdminUpgradeable			
Function Name	Visibility	Mutability	Modifiers
__AdminUpgradeable_init	Internal	Can Modify State	onlyInitializing
isAdmin	Public	-	-
addAdmin	External	Can Modify State	onlyAdmin
renounceAdmin	External	Can Modify State	-

FlockMiniPool			
Function Name	Visibility	Mutability	Modifiers
<Constructor>	Public	Can Modify State	-
delegate	External	Can Modify State	-
undelegate	External	Can Modify State	-
claimRewards	External	Can Modify State	-
fundPool	External	Can Modify State	-
delegationOf	Public	-	-
getTotalDelegationAmount	External	-	-
claimable	Public	-	-

FlockMiniPool			
receiverTemporalDelegation	Public	-	-
totalTemporalDelegation	Public	-	-
getDelegators	External	-	-
interestPerYear	External	-	-
totalRewardForDelegator	External	-	-
addPastTemporalDelegation	Internal	Can Modify State	-
calculateTemporalDelegation	Internal	-	-
updateTotalDelegationInfoAndPastTemporalDelegation	Internal	Can Modify State	-
resetDelegationTimestampWithTemporalDelegation	Internal	Can Modify State	-
resetDelegationTimestamp	Internal	Can Modify State	-
adjustTotalDelegationWithTemporalDelegation	Internal	Can Modify State	-

FlockPoolManagerUpgradeable			
Function Name	Visibility	Mutability	Modifiers
initialize	Public	Can Modify State	initializer
setMinSigma	External	Can Modify State	onlyAdmin
setMaxSigma	External	Can Modify State	onlyAdmin
setProtocolFeePercentage	External	Can Modify State	onlyAdmin
setSigmaModificationCooldown	External	Can Modify State	onlyAdmin
setConfig	External	Can Modify State	onlyAdmin
getMiniPool	External	-	-
getPoolSigma	External	-	-

FlockPoolManagerUpgradeable			
isPool	Public	-	-
getPercentageBase	External	-	-
getProtocolFeePercentage	External	-	-
getPools	External	-	-
getUsers	External	-	-
getUserAt	External	-	-
getUsersLength	External	-	-
getDelegatorPools	External	-	-
createMiniPool	External	Can Modify State	onlyWhitelisted
setMiniPoolSigma	External	Can Modify State	-
addDelegatorToPool	External	Can Modify State	-

FlockStakeInfoUpgradeable			
Function Name	Visibility	Mutability	Modifiers
initialize	Public	Can Modify State	initializer
setConfig	External	Can Modify State	onlyAdmin
setMaxDelegationWeight	External	Can Modify State	onlyAdmin
initializeTotalNodeStakesPerTask	External	Can Modify State	onlyAdmin
initializeTotalValidatorStakesPerTask	External	Can Modify State	onlyAdmin
initialzeTotalActiveTaskStakes	External	Can Modify State	onlyAdmin
initializeTaskStakes	External	Can Modify State	onlyAdmin
addStakes	External	Can Modify State	-
removeStakes	External	Can Modify State	-

FlockStakeInfoUpgradeable			
clearStakes	External	Can Modify State	-
retallyDelegation	External	Can Modify State	-
getDelegationTokenAmountForUser	Public	-	-
getTotalUserStakeOverAllActiveTasks	Public	-	-
getTotalActiveTaskStakes	External	-	-
getTaskStakes	External	-	-
getTotalActiveTaskWeights	External	-	-
getTaskWeights	External	-	-
getTotalNodeStakesPerTask	External	-	-
getTotalValidatorStakesPerTask	External	-	-
retallyDelegationInternal	Internal	Can Modify State	-

FlockTaskManagerUpgradeable			
Function Name	Visibility	Mutability	Modifiers
initialize	Public	Can Modify State	initializer
setBaseStakeAmount	External	Can Modify State	onlyAdmin
setRewardTokenPerDay	External	Can Modify State	onlyAdmin
setLockedRewardPercentage	External	Can Modify State	onlyAdmin
setUnlockedRewardPercentage	External	Can Modify State	onlyAdmin
setRewardPercentageBase	External	Can Modify State	onlyAdmin
setMinNodeStakes	External	Can Modify State	onlyAdmin
setMaxDuration	External	Can Modify State	onlyAdmin
setValidatorMinStake	External	Can Modify State	onlyAdmin

FlockTaskManagerUpgradeable			
setBypassWhitelistForNode	External	Can Modify State	onlyAdmin
setBypassWhitelistForValidator	External	Can Modify State	onlyAdmin
setValidatorFeeWithdrawalPercentage	External	Can Modify State	onlyAdmin
setTotalStakes	External	Can Modify State	-
setGamma	External	Can Modify State	onlyAdmin
setConfig	External	Can Modify State	onlyAdmin
setFeePercentage	External	Can Modify State	onlyAdmin
getNodeStakes	External	-	-
getValidatorStakes	External	-	-
getAvailableRewardTasksForUser	External	-	-
isTaskCompleted	External	-	-
getTotalStakes	External	-	-
getNodeTasks	External	-	-
getValidatorTasks	External	-	-
createTask	External	Can Modify State	onlyAdmin
stakeTokensForTrainingNodes	External	Can Modify State	-
withdrawStakeTokensForTrainingNodes	External	Can Modify State	-
claimRewards	External	Can Modify State	-
stakeTokensForValidators	External	Can Modify State	-
withdrawStakeTokensForValidators	External	Can Modify State	-
transferRewardsToFLTasks	External	Can Modify State	onlyAdmin
mintDailyReward	External	Can Modify State	onlyAdmin

FlockTaskManagerUpgradeable			
markTaskAsFinished	Public	Can Modify State	onlyAdmin
uploadRewardResultsForTrainingNodes	Public	Can Modify State	onlyAdmin
uploadRewardResultsForValidators	Public	Can Modify State	onlyAdmin
uploadRewardResultsOnlyForValidators	Public	Can Modify State	onlyAdmin
distributeRewardToDelegators	Internal	Can Modify State	-
uploadFinalRewardResultsForTrainingNodes	Public	Can Modify State	onlyAdmin
afterDelegationUpdate	External	Can Modify State	-

FlockTokenUpgradeable			
Function Name	Visibility	Mutability	Modifiers
initialize	External	Can Modify State	initializer
setSigner	External	Can Modify State	onlyAdmin
isWhitelisted	External	-	-
isWeaklyWhitelisted	External	-	-
isAtLeastWeaklyWhitelisted	External	-	-
setDailyMintLimit	External	Can Modify State	onlyAdmin
addToWhitelist	External	Can Modify State	onlyAdmin
addToWeaklyWhitelist	External	Can Modify State	onlyAdmin
removeFromWeaklyWhitelist	External	Can Modify State	onlyAdmin
addToWhitelistAndMintToken	External	Can Modify State	onlyAdmin
removeFromWhitelist	External	Can Modify State	onlyAdmin
mint	External	Can Modify State	onlyAdmin
burn	External	Can Modify State	onlyAdmin

FlockTokenUpgradeable			
itoa	Internal	-	-
checkSignature	Internal	-	-
selfMint	External	Can Modify State	-

FlockConfig			
Function Name	Visibility	Mutability	Modifiers
initialize	External	Can Modify State	initializer
setAddress	Public	Can Modify State	onlyAdmin
copyFromOtherConfig	External	Can Modify State	onlyAdmin
getAddress	External	-	-

4.3 Vulnerability Summary

[N1] [Low] Potential risks of signature malleability

Category: Replay Vulnerability

Content

In the FlockTokenUpgradeable contract, the checkSignature function is used to check the validity of a signature to determine if the signature owner is the contract's signer. However, it is important to note that when performing the signature check, the validity of the `s` value is not checked. Due to the malleability of signatures still allowed by the ecrecover interface, this renders the usedSignatures check useless, allowing for multiple mints.

Code location: contracts/FlockTokenUpgradeable.sol#L198

```
    function checkSignature(string calldata message, bytes32 _r, bytes32 _s, uint8
_v) internal view returns (bool) {
        // Prefix the message
        bytes memory prefixedMessage = abi.encodePacked("\x19Ethereum Signed
Message:\n", itoa(bytes(message).length), message);

        // Compute the message digest
```

```
        bytes32 digest = keccak256(prefixedMessage);
        address _signer = ecrecover(digest, _v, _r, _s);

        return signer == _signer;
    }
```

Solution

It is recommended to use OpenZeppelin's ECDSA library for signature verification.

Status

Acknowledged

[N2] [Critical] Risk of whitelisted users front-running

Category: Reordering Vulnerability

Content

In the FlockTokenUpgradeable contract, whitelisted users can mint tokens using valid signature data through the selfMint function. Unfortunately, when performing signature verification, the message used is passed in by the user, rather than being constructed within the contract, and it does not verify the relationship between the caller and the message they provide. This allows any user to pass in any valid and unused message along with the corresponding signature to mint tokens. As a result, malicious users can continuously front-run other users' transactions to steal tokens that should belong to other whitelisted users.

Code location: contracts/FlockTokenUpgradeable.sol#L211

```
    function selfMint(uint256 _amount, string calldata message, bytes32 _r, bytes32
_s, uint8 _v) external {
        require(whitelisted[msg.sender], "Caller is not whitelisted");
        bytes32 signatureHash = keccak256(abi.encodePacked(_r, _s, _v));
        require(!usedSignatures[signatureHash], "Signature already used");
        require(checkSignature(message, _r, _s, _v), "Invalid signature");

        ...
    }
```

Solution

It is recommended to construct the `message` within the contract, including `msg.sender`, rather than relying on user-provided data.

Status

Fixed

[N3] [Critical] Risks of self-minting arbitrary amounts of tokens

Category: Design Logic Audit

Content

In the FlockTokenUpgradeable contract, when whitelisted users mint tokens through the selfMint function, there is no verification whether the minted token amount matches the quantity authorized in the signer's signature. As a result, after obtaining a valid signature, whitelisted users can mint any amount of tokens up to the dailyMintLimit.

Code location: contracts/FlockTokenUpgradeable.sol#L207

```
    function selfMint(uint256 _amount, string calldata message, bytes32 _r, bytes32
_s, uint8 _v) external {
        require(whitelisted[msg.sender], "Caller is not whitelisted");
        bytes32 signatureHash = keccak256(abi.encodePacked(_r, _s, _v));
        require(!usedSignatures[signatureHash], "Signature already used");
        require(checkSignature(message, _r, _s, _v), "Invalid signature");

        ...
        require(amountMintedToday[msg.sender] + _amount <= dailyMintLimit, "Daily mint
limit exceeded");
        amountMintedToday[msg.sender] += _amount;
        usedSignatures[signatureHash] = true;
        _mint(msg.sender, _amount);
    }
```

Solution

If this is not the intended design, it is recommended to include the user-provided `_amount` parameter in the `message` that requires signature verification.

Status

Acknowledged; After communicating with the project team, the project team stated that this is the expected design, which allows whitelisted users to mint up to the `dailyMintLimit` amount of tokens.

[N4] [Critical] Risk of Signature Replay

Category: Replay Vulnerability

Content

In the selfMint function of the FlockTokenUpgradeable contract, it encodes the r/s/v values of the signature using keccak256 and performs usedSignatures verification to prevent users from reusing the same signature data. However, it should be noted that the `signatureHash` does not include any domain separators such as chain ID or contract address. This allows signatures obtained through other means by the signing users to be used for token minting. Furthermore, if the FlockTokenUpgradeable contract is deployed across multiple chains with the same signer, users can utilize data from other chains to perform multi-chain minting.

Code location: contracts/FlockTokenUpgradeable.sol#L192-L223

```
    function selfMint(uint256 _amount, string calldata message, bytes32 _r, bytes32
_s, uint8 _v) external {
        require(whitelisted[msg.sender], "Caller is not whitelisted");
        bytes32 signatureHash = keccak256(abi.encodePacked(_r, _s, _v));
        require(!usedSignatures[signatureHash], "Signature already used");
        ...
    }
```

Solution

It is recommended to follow OpenZeppelin's EIP712 implementation by using necessary domain separators when constructing signature data, and including the message as part of the signatureHash composition.

Status

Fixed

[N5] [Low] Potential risk of not checking whether the signer is a 0 address

Category: Design Logic Audit

Content

In the FlockTokenUpgradeable contract, the signer address is set during contract initialization or through the setSigner function, but it does not check whether the new address is a zero address. Since the signer is involved in token minting permissions, if it is incorrectly set to a zero address, any whitelisted user can mint tokens arbitrarily without requiring valid signature data.

Code location: contracts/FlockTokenUpgradeable.sol#L31-L44

```
    function initialize(uint256 _dailyMintLimit, string calldata _tokenName, string
calldata _tokenSymbol, address _signer) external initializer {
        __AdminUpgradeable_init(msg.sender);
        __ERC20_init(_tokenName, _tokenSymbol);
        dailyMintLimit = _dailyMintLimit;
        signer = _signer;
    }

    function setSigner(address _signer) external onlyAdmin {
        signer = _signer;
    }
```

Solution

It is recommended to either check that the input `_signer` is not a zero address in both initialize and setSigner functions or use OpenZeppelin's ECDSA library for signature verification, which will reject signature data with a zero address signer.

Status

Fixed

[N6] [Critical] Using incorrect role for reward calculation

Category: Design Logic Audit

Content

In the FlockTaskManagerUpgradeable contract, the admin role can distribute rewards to validators for their tasks through the uploadRewardResultsForValidators function. However, when calculating taskRewardForValidatorAndDelegator, it incorrectly uses the NODE role's task weight instead of the intended VALIDATOR role's weight.

Code location: contracts/FlockTaskManagerUpgradeable.sol#L667

```
    function uploadRewardResultsForValidators(uint256 taskId, address[] calldata
validatorsAddrs, uint256[] calldata rewardPercentages) public onlyAdmin {
        ...
        uint256 taskRewardForValidatorAndDelegator = (rewardTokenPerDayForArena *
config.getStakeInfo().getTaskWeights(taskId, uint256(ROLE.NODE))) /
config.getStakeInfo().getTotalActiveTaskWeights(uint256(ROLE.NODE));
```

```
        ...
    }
```

Solution

The calculation of taskRewardForValidatorAndDelegator should use the task weight associated with the VALIDATOR role instead.

Status

Fixed

[N7] [High] Potential risk of funds being locked for task id 0

Category: Design Logic Audit

Content

In the FlockTaskManagerUpgradeable contract, the admin role can create tasks with specified taskIds through the createTask function, but the contract does not restrict taskId from being zero. If the admin role creates a task with taskId of 0, it cannot be marked as 'completed' through the markTaskAsFinished function, resulting in locked staked funds that cannot be withdrawn from this task.

Code location: contracts/FlockTaskManagerUpgradeable.sol#L452

```
    function createTask(uint256 taskId, uint256 expectedDuration, string memory
taskName) external onlyAdmin {
        ...
        tasks[taskId] = Task({id: taskId, creator: msg.sender, startTime:
block.timestamp, expectedDuration: expectedDuration, stakeAmount: stakeAmount,
isCompleted: false, taskName: taskName});

        ...
    }

    function markTaskAsFinished(uint256 taskId) public onlyAdmin {
        require(tasks[taskId].id != 0, "Task does not exist");
        ...
    }
```

Solution

It is recommended to add a check in the createTask function to ensure the input taskId is greater than zero.

Status

Fixed

[N8] [Suggestion] Some Task parameters are not used

Category: Others

Content

In the FlockTaskManagerUpgradeable contract, the admin role is empowered to create tasks via the createTask function and establish parameters such as start time, expected duration, and stake amount. However, it is important to note that these parameters are not being effectively utilized; specifically, the actual task duration does not adhere to the specified expected duration, and there is no validation against expected duration upon task completion. Likewise, the stake amount staked by the creator remains unutilized throughout the lifecycle of the task.

Code location: contracts/FlockTaskManagerUpgradeable.sol#L147-L155

```
struct Task {
    uint256 id;
    bool isCompleted;
    address creator;
    uint256 startTime;
    uint256 expectedDuration;
    uint256 stakeAmount;
    string taskName;
}
```

Solution

It is advisable to clarify design expectations in order to ensure that all parameters are employed as intended.

Status

Acknowledged; After communicating with the project team, the project team stated that these data will be used in the testnet, and these historical parameters will be cleared when the mainnet is launched in the future.

[N9] [Suggestion] Unable to guarantee that the mintDailyReward operation is performed only once per day

Category: Others

Content

In the FlockTaskManagerUpgradeable contract, the admin mints daily Flock token rewards to this contract through the mintDailyReward function. However, it's important to note that the contract neither restricts mintDailyReward operations to once per day nor mandates daily mintDailyReward executions. Multiple executions or missed mintDailyReward operations can result in reward losses.

Code location: contracts/FlockTaskManagerUpgradeable.sol#L605

```
    function mintDailyReward() external onlyAdmin {
        config.getFlockToken().mint(address(this), rewardTokenPerDay);
        emit MintDailyReward(rewardTokenPerDay);
    }
```

Solution

It is recommended that the project team maintains a keeper program to ensure successful execution of mintDailyReward once daily, or alternatively, utilizes services like ChainLink Automation.

Status

Acknowledged

[N10] [Suggestion] The total staked amount is not deducted when removing user stake

Category: Design Logic Audit

Content

In the protocol, when users withdraw staked funds through the task management contract after task completion, the removeStakes function in the StakeInfo contract is called to update the user's staking amount and recalculate task weights via retallyDelegationInternal. It's worth noting that when clearing task data, the totalNodeStakesPerTask/totalValidatorStakesPerTask values for the task are not reset to 0.

Code location: contracts/FlockStakeInfoUpgradeable.sol#L192-L196

```
    function addStakes(uint256 taskId, uint256 stakes, uint256 role, address user)
external {
        ...
        if (role == ROLE_NODE) {
            totalNodeStakesPerTask[taskId] += stakes;
        } else if (role == ROLE_VALIDATOR) {
            totalValidatorStakesPerTask[taskId] += stakes;
        }
```

```
        ...
    }

    function removeStakes(uint256 taskId, uint256 stakes, uint256 role, address user)
external {
        require(msg.sender == config.mainManagerAddress(), "Only Main Manager can call
this function");
        config.getMainManager().setTotalStakes(user,
config.getMainManager().getTotalStakes(user) >= stakes ?
config.getMainManager().getTotalStakes(user) - stakes : 0);
        retallyDelegationInternal(user);
        emit ParticipantUnstaked(user, stakes, role, taskId);
    }
```

Solution

If this is not the intended design, it is recommended to thoroughly clean up all data associated with the task upon completion.

Status

Acknowledged; After communicating with the project team, the project team stated that this was the expected design.

[N11] [Medium] Upload reward does not check if the user belongs to the task

Category: Design Logic Audit

Content

In the FlockTaskManagerUpgradeable contract, the admin role can distribute rewards to various roles through the uploadRewardResultsForTrainingNodes, uploadRewardResultsForValidators, and uploadRewardResultsOnlyForValidators functions. The admin specifies the taskId for reward distribution, trainingNodesAddrs for reward recipients, and rewardPercentages for allocation. However, the functions do not verify whether the addresses in the trainingNodesAddrs list actually participated in the specified task. If the admin mistakenly includes addresses of roles that did not participate in the task, these addresses would receive unwarranted rewards.

Code location: contracts/FlockTaskManagerUpgradeable.sol#L631,L661,L692

```
    function uploadRewardResultsForTrainingNodes(uint256 taskId, address[] calldata
trainingNodesAddrs, uint256[] calldata rewardPercentages) public onlyAdmin {
```

```
        ...
    }

    function uploadRewardResultsForValidators(uint256 taskId, address[] calldata
validatorsAddrs, uint256[] calldata rewardPercentages) public onlyAdmin {
        ...
    }

    function uploadRewardResultsOnlyForValidators(uint256 taskId, address[] calldata
validatorsAddrs, uint256[] calldata rewardPercentages) public onlyAdmin {
        ...
    }
```

Solution

It is recommended to implement verification checks during reward uploads to ensure that all addresses in the trainingNodesAddrs list participated in the task being rewarded.

Status

Acknowledged; After communicating with the project team, the project team stated that the relationship between users and tasks will be handled on the backend.

[N12] [Low] Did not check if the sum of rewardPercentages of reward distribution is as expected

Category: Others

Content

In the FlockTaskManagerUpgradeable contract, when the admin role uploads rewards for distribution, they include a rewardPercentages list to ensure rewards are allocated according to predetermined percentages for each address. However, it should be noted that the contract does not verify whether the sum of rewardPercentages equals 100%. If the admin inadvertently uploads reward percentages that exceed the expected total, it will result in incorrect reward distribution.

Code location: contracts/FlockTaskManagerUpgradeable.sol#L631,L661,L692

```
    function uploadRewardResultsForTrainingNodes(uint256 taskId, address[] calldata
trainingNodesAddrs, uint256[] calldata rewardPercentages) public onlyAdmin {
        ...
    }

    function uploadRewardResultsForValidators(uint256 taskId, address[] calldata
validatorsAddrs, uint256[] calldata rewardPercentages) public onlyAdmin {
```

```
        ...
    }

    function uploadRewardResultsOnlyForValidators(uint256 taskId, address[] calldata validatorsAddrs, uint256[] calldata rewardPercentages) public onlyAdmin {
        ...
    }
```

Solution

It is recommended to implement a verification check during reward uploads to ensure the sum of rewardPercentages matches the expected total.

Status

Acknowledged

[N13] [High] Risk of part of the reward being locked

Category: Design Logic Audit

Content

In the FlockTaskManagerUpgradeable contract, users can claim allocated rewards through the claimRewards function. The contract charges a fee (feePercentage), but these fees remain directly in the contract without being tracked in any global variable for future withdrawal. This may result in a portion of rewards being locked within the FlockTaskManagerUpgradeable contract with no way to retrieve them.

Code location: contracts/FlockTaskManagerUpgradeable.sol#L506

```
    function claimRewards() external {
        uint256 rewardAmount = userRewards[msg.sender]-userRewards[msg.sender]*feePercentage/rewardPercentageBase;
        ...
    }
```

Solution

If this is not the intended design, it is recommended to implement a fee withdrawal interface in the contract to mitigate this risk.

Status

Fixed; After communicating with the project, the project party stated that it will redesign the expense processing logic based on actual business needs in the future.

[N14] [Suggestion] Redundant code

Category: Others

Content

In the FlockTaskManagerUpgradeable contract, there are several redundant global variables, such as taskStakesWithPower, totalNodeStakesPerTask, totalValidatorStakesPerTask, and totalDelegatorStakesPerTaskPerValidator. While these variables are defined, they are never utilized in the contract. Additionally, although the LibMath library is imported, it has never been used.

Code location:

contracts/FlockTaskManagerUpgradeable.sol

contracts/libs/LibMath.sol

Solution

It is recommended to clarify the design intentions and remove unnecessary code to maintain code cleanliness.

Status

Fixed

[N15] [High] Unable to withdraw withdrawal fee

Category: Design Logic Audit

Content

In the FlockTaskManagerUpgradeable contract, validators can withdraw their staked funds through the withdrawStakeTokensForValidators function after task completion. During the withdrawal process, a certain percentage is charged as a withdrawal fee, which is recorded in the validatorWithdrawalFeeInVault variable. Unfortunately, the contract lacks any interface to withdraw these collected fees, meaning the withdrawal fees will remain locked within the contract with no way to retrieve them.

Code location: contracts/FlockTaskManagerUpgradeable.sol#L563

```
    function withdrawStakeTokensForValidators(uint256 taskId) external {
        ...
```

```
        uint256 withdrawalFee = (validatorWithdrawalFeePercentage * stakedAmount) /
rewardPercentageBase;
        validatorWithdrawalFeeInVault += withdrawalFee;
        ...
    }
```

Solution

It is recommended to implement an interface for withdrawing the validatorWithdrawalFeeInVault fees to prevent this lock-up risk.

Status

Acknowledged; After communicating with the project team, the project team stated that validatorWithdrawalFeePercentage will be set to 0 when the mainnet is launched to address the above risks.

[N16] [Medium] Risks of calling markTaskAsFinished directly to complete the task

Category: Design Logic Audit

Content

In the FlockTaskManagerUpgradeable contract, the admin role can mark a specific task as completed by calling the markTaskAsFinished function. Additionally, the uploadFinalRewardResultsForTrainingNodes function also calls markTaskAsFinished to complete tasks and distribute locked rewards. Unfortunately, the markTaskAsFinished function checks the isCompleted status of the current task, which means if the admin role has previously called markTaskAsFinished to complete a task, it becomes impossible to distribute locked rewards through the uploadFinalRewardResultsForTrainingNodes function.

Code location: contracts/FlockTaskManagerUpgradeable.sol#L616,L753

```
    function markTaskAsFinished(uint256 taskId) public onlyAdmin {
        require(tasks[taskId].id != 0, "Task does not exist");
        require(!tasks[taskId].isCompleted, "Task already completed");

        ...
    }

    function uploadFinalRewardResultsForTrainingNodes(uint256 taskId, address[]
calldata trainingNodesAddrs, uint256[] calldata rewardPercentages) public onlyAdmin {
        markTaskAsFinished(taskId);
```

```
        ...
    }
```

Solution

It is recommended to modify the markTaskAsFinished implementation so that if the specified task is already completed, it simply returns instead of forcing a revert, ensuring compatibility with the uploadFinalRewardResultsForTrainingNodes function.

Status

Fixed

[N17] [Medium] Risks of excessive privilege

Category: Authority Control Vulnerability Audit

Content

In the FlockToken contract, the owner role can burn tokens from any user's account using the burn function. Similarly in the FlockTokenUpgradeable contract, the admin role can mint/burn arbitrary amounts of tokens.

In the FlockStakeInfoUpgradeable contract, the admin role can set arbitrary staking amounts through initializeTotalNodeStakesPerTask/initializeTotalValidatorStakesPerTask/initialzeTotalActiveTaskStakes/initializeTaskStakes functions.

All of these create risks of excessive privileges.

Code location:

contracts/FlockTokenUpgradeable.sol#L137-L148

```
    function mint(address _account, uint256 _amount) external onlyAdmin {
        _mint(_account, _amount);
    }

    function burn(address _account, uint256 _amount) external onlyAdmin {
        _burn(_account, _amount);
    }
```

contracts/FlockStakeInfoUpgradeable.sol#L125-L163

```
    function initializeTotalNodeStakesPerTask(uint256 taskId, uint256
totalNodeStakes) external onlyAdmin {
```

```
        totalNodeStakesPerTask[taskId] = totalNodeStakes;
    }

    function initializeTotalValidatorStakesPerTask(uint256 taskId, uint256
totalValidatorStakes) external onlyAdmin {
        totalValidatorStakesPerTask[taskId] = totalValidatorStakes;
    }

    function initialzeTotalActiveTaskStakes(uint256 _totalActiveTaskStakes) external
onlyAdmin {
        totalActiveTaskStakes = _totalActiveTaskStakes;
        totalActiveTaskStakesOnDelegationOverNode = _totalActiveTaskStakes;
        totalActiveTaskStakesOnDelegationOverValidator = _totalActiveTaskStakes;
        totalActiveTaskWeights = _totalActiveTaskStakes;
        totalActiveTaskWeightsOnDelegationOverNode = _totalActiveTaskStakes;
        totalActiveTaskWeightsOnDelegationOverValidator = _totalActiveTaskStakes;
    }

    function initializeTaskStakes(uint256 taskId,uint256 stakes) external onlyAdmin {
        taskStakes[taskId] = stakes;
        taskStakesOnDelegationOverNode[taskId] = stakes;
        taskStakesOnDelegationOverValidator[taskId] = stakes;
        taskWeights[taskId] = stakes;
        taskWeightsOnDelegationOverNode[taskId] = stakes;
        taskWeightsOnDelegationOverValidator[taskId] = stakes;
    }
```

Solution

In the short term, transferring privileged roles to a timelock contract, and then having the timelock contract managed by a multisig contract can effectively mitigate single point of failure risks and reduce excessive privilege risks.

In the long term, transferring privileged roles to DAO governance can effectively resolve excessive privilege risks and gain more trust from community users. To better handle emergencies, the protocol can add an emergency pause role managed by the project team to quickly stop losses in emergency situations.

Status

Fixed; The project team has addressed this issue in PR41 by removing the initialization function from the FlockStakeInfoUpgradeable contract. They have also implemented a maximum minting cap for FLock tokens and eliminated the functionality that allowed arbitrary token burning of user tokens.

5 Audit Result

Audit Number	Audit Team	Audit Date	Audit Result
0X002411110002	SlowMist Security Team	2024.10.29 - 2024.11.11	Low 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 4 critical risks, 3 high risks, 3 medium risks, 3 low risks, and 4 suggestions. All the findings were fixed or acknowledged. The code was not deployed to the mainnet.

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.

Official Website

www.slowmist.com

E-mail

team@slowmist.com

Twitter

@SlowMist_Team

Github

https://github.com/slowmist