Smart Contract Security Audit Report

Table Of Contents

1 Executive Summary

On 2025.01.17, the SlowMist security team received the FLock team's security audit application for Flock Phase2, 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

This is the FL alliance client section of the Flock protocol, mainly consisting of task contracts and task management contracts. Users can stake in the task contracts, where they will be selected as proposers and voters. Proposers submit model hashes, and voters vote on these model hashes. The protocol will distribute slashing penalties and rewards based on the final voting results.

3.2 Vulnerability Information

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

NO	Title	Category	Level	Status
N1	FlockTaskManager does not implement necessary functionality	Design Logic Audit	Medium	Fixed
N2	Unchecked transfer return value	Others	Suggestion	Fixed
N3	Risk of not checking whether the task has been finished when staking	Design Logic Audit	Medium	Fixed
N4	seedRoundParticipants have the risk of gas war	Race Conditions Vulnerability	High	Fixed
N5	Pseudo-random risks	Block data Dependence Vulnerability	Medium	Acknowledged
N6	Unable to verify whether the hash came from the client	Design Logic Audit	Critical	Fixed
N7	Allow voters to cast votes with a score of 0	Design Logic Audit	Low	Fixed
N8	Potential pitfalls of the updateClaimableReward function	Design Logic Audit	Medium	Fixed
N9	DoS risk if slash functions are not called for a long time	Denial of Service Vulnerability	Low	Fixed
N10	Gas optimization for slash functions	Gas Optimization Audit	Suggestion	Fixed
N11	PopularHash is at risk of a 51% attack	Design Logic Audit	Critical	Fixed
N12	TotalNumberOfRounds is not checked when finishing a round	Design Logic Audit	Medium	Fixed
N13	There is no order of precedence between contribute and vote	Design Logic Audit	High	Fixed
N14	Potential issue where the client cannot claim	Design Logic Audit	Medium	Fixed

NO	Title	Category	Level	Status
	rewards normally			

4 Code Overview

4.1 Contracts Description

Audit Version:

https://github.com/FLock-io/FL-Alliance-Client/tree/main/client

commit: 2a9ee24e245ed67bc19d8a094959655c0ee479aa

Fixed Version:

https://github.com/FLock-io/FL-Alliance-Client/tree/sc_audit_fix

commit: ec48ebf3cdef2a43976edd170c71df22dde0e9a1

Audit Scope:

```
./contracts/contracts
├── FlockTask.sol
├── FlockTaskManager.sol
└── FlockToken.sol
```

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:

FlockTask			
Function Name	Visibility	Mutability	Modifiers
<Constructor>	Public	Can Modify State	-
setSlashedPerStake	External	Can Modify State	onlyOwner

FlockTask			
setCallbackGasLimit	External	Can Modify State	onlyOwner
setMinStakeThreshold	External	Can Modify State	onlyOwner
setSecondsPerRound	External	Can Modify State	onlyOwner
setMaxNumberOfParticipants	External	Can Modify State	onlyOwner
setMinNumberOfParticipants	External	Can Modify State	onlyOwner
setBonusRatioForSlashCaller	External	Can Modify State	onlyOwner
setParticipantsDistributionRate	External	Can Modify State	onlyOwner
setFinished	External	Can Modify State	onlyOwner
initializeRewardPool	External	Can Modify State	-
fundRewardPool	External	Can Modify State	-
stake	External	Can Modify State	-
joinRound	External	Can Modify State	updateRound
seedRoundParticipants	External	Can Modify State	updateRound
selectRoundParticipants	External	Can Modify State	updateRound
contribute	Public	Can Modify State	updateRound
vote	External	Can Modify State	updateRound
updateClaimableReward	External	Can Modify State	updateRound
claimRewards	External	Can Modify State	updateRound
allContributed	External	-	-
slash	External	Can Modify State	updateRound
enoughParticipantsJoined	Public	-	-
getNumberOfProposers	Public	-	-

FlockTask			
getNumberOfParticipants	Public	-	-
isFinished	Public	-	-
getTotalStakes	External	-	-
hasRoundFinished	Public	-	-
mostPopularHash	Public	-	-
canBeSlashed	Public	-	-
finishRound	Internal	Can Modify State	-
shuffleParticipants	Internal	Can Modify State	-
distributeParticipants	Internal	Can Modify State	-
setNodeMetadata	External	Can Modify State	-
getTotalStakesForGoodParticipants	Public	-	-
getParticipants	External	-	-

FlockTaskManager			
Function Name	Visibility	Mutability	Modifiers
<Constructor>	Public	Can Modify State	-
finishTask	Public	Can Modify State	onlyOwner
setFlockTokenAddress	Public	Can Modify State	onlyOwner
createTask	Public	Can Modify State	onlyOwner
getTasks	Public	-	-

FlockToken			
Function Name	Visibility	Mutability	Modifiers

FlockToken			
<Constructor>	Public	Can Modify State	ERC20
mint	Public	Can Modify State	onlyOwner
burn	Public	Can Modify State	onlyOwner

4.3 Vulnerability Summary

[N1] [Medium] FlockTaskManager does not implement necessary functionality

Category: Design Logic Audit

Content

In the protocol, FlockTasks are created by the FlockTaskManager contract, and the FlockTask contract's owner is set to the FlockTaskManager contract. While the FlockTask contract implements multiple functions that can only be called by the owner role, the FlockTaskManager contract only implements the capability to call `FlockTask.setFinished()`. It does not implement calls to other onlyOwner-modified functions in the FlockTask contract, such as setSlashedPerStake, setCallbackGasLimit, setMinStakeThreshold, and others.

Code location: contracts/federal_learning/FlockTask.sol#L141-L199

```
    function setSlashedPerStake(uint256 _slashedPerStake) external onlyOwner {
        slashedPerStake = _slashedPerStake;
    }

    function setCallbackGasLimit(uint32 _callbackGasLimit) external onlyOwner {
        callbackGasLimit = _callbackGasLimit;
    }

    function setMinStakeThreshold(uint256 _minStakeThreshold) external onlyOwner {
        minStakeThreshold = _minStakeThreshold;
    }

    function setSecondsPerRound(uint256 _secondsPerRound) external onlyOwner {
        secondsPerRound = _secondsPerRound;
    }

    function setMaxNumberOfParticipants(uint256 _maxNumberOfParticipants) external
onlyOwner {
        maxNumberOfParticipants = _maxNumberOfParticipants;
    }

    function setMinNumberOfParticipants(uint256 _minNumberOfParticipants) external
onlyOwner {
        minNumberOfParticipants = _minNumberOfParticipants;
    }

    function setBonusRatioForSlashCaller(uint256 _bonusRatioForSlashCaller) external
onlyOwner {
        bonusRatioForSlashCaller = _bonusRatioForSlashCaller;
    }

    function setParticipantsDistributionRate(uint256 _participantsDistributionRate)
external onlyOwner {
        participantsDistributionRate = _participantsDistributionRate;
    }
```

Solution

It is recommended to clarify the design intentions and either implement necessary management functions in the FlockTaskManager contract or remove unused functions from the FlockTask contract.

Status

Fixed

[N2] [Suggestion] Unchecked transfer return value

Category: Others

Content

In the FlockTask contract, when users stake or claim rewards, the contract performs Flock token transfers without checking the return values of these transfer operations.

Code location: contracts/federal_learning/FlockTask.sol#L227,L242,L381

```
    function fundRewardPool(uint256 _amount) external  {
        actualFLRewardForTask += _amount;
        FlockToken.transferFrom(msg.sender, address(this), _amount);
    }

    function stake(uint256 _amount) external {
        ...
        FlockToken.transferFrom(msg.sender, address(this), _amount);

        emit Staked(msg.sender, _amount);
    }

    function claimRewards() external updateRound {
        ...
        FlockToken.transfer(msg.sender, reward);
        emit Claimed(msg.sender, reward, currentRound, "finalReward");
    }
```

Solution

Although the Flock token complies with the EIP20 standard, it is still recommended to check the return values of token transfers to align with best practices.

Status

Fixed

[N3] [Medium] Risk of not checking whether the task has been finished when staking

Category: Design Logic Audit

Content

In the FlockTask contract, users can stake Flock tokens through the stake function and participate in rounds via the joinRound function. It's important to note that FlockTask has a finished state, which indicates task completion when either the number of rounds reaches totalNumberOfRounds or the contract's finished state is set to true. However, even after a FlockTask is marked as finished, users can still participate indefinitely through stake/joinRound functions, which contradicts the expected behavior. This means the isFinished state of FlockTask is not serving its intended purpose.

Solution:

Code location: contracts/federal_learning/FlockTask.sol#L482,L234,L250

```
    function stake(uint256 _amount) external {
        ...
    }

    function joinRound() external updateRound {
        ...
    }

    function isFinished() public view returns (bool) {
        return finished || (currentRound == totalNumberOfRounds);
    }
```

Solution

It is recommended to prevent any user participation once a FlockTask contract is marked as finished.

Status

Fixed

[N4] [High] seedRoundParticipants have the risk of gas war

Category: Race Conditions Vulnerability

Content

In the FlockTask contract, any user can call the seedRoundParticipants function with unverified "random" to sort participants. Since seedRoundParticipants can only be called once per round, participants might engage in gas wars for their own benefit, attempting to manipulate the participant ordering to secure advantageous positions. This makes the participant ordering susceptible to manipulation and violates the principle of fairness.

Code location: contracts/federal_learning/FlockTask.sol#L271

```
    function seedRoundParticipants(uint256 _random) external updateRound {
        ...
        shuffleParticipants(participants[currentRound], _random);
        distributeParticipants(currentRound);
    }
```

Solution

Given that the selectRoundParticipants function already exists, allowing arbitrary users to perform seedRoundParticipants operations with arbitrary "random" is not a good approach. It is recommended to keep only the selectRoundParticipants function for participant ordering.

Status

Fixed

[N5] [Medium] Pseudo-random risks

Category: Block data Dependence Vulnerability

Content

In the FlockTask contract, when participants are randomly ordered through the selectRoundParticipants function, the contract obtains random numbers from the rand function in the LibFlockTask library. Unfortunately, the rand function exclusively uses on-chain data for random number generation, making the results predictable. Participants could predict the random numbers to position themselves favorably in the random ordering. However, since any user can call the selectRoundParticipants function, this effectively reduces the success rate of participant manipulation.

Code location: contracts/libs/LibFlockTask.sol#L27

```
    function rand() public view returns (uint256) {
        return uint256(keccak256(abi.encodePacked(block.timestamp, block.difficulty,
((uint256(keccak256(abi.encodePacked(block.coinbase)))) / (block.timestamp)),
block.gaslimit, ((uint256(keccak256(abi.encodePacked(msg.sender)))) /
(block.timestamp)), block.number)));
    }
```

Solution

Using Chainlink's VRF would be the best solution for implementation.

Status

Acknowledged; After communicating with the project team, they said they trust the sequencer, but would consider using a third-party VRF in the future.

[N6] [Critical] Unable to verify whether the hash came from the client

Category: Design Logic Audit

Content

In the FlockTask contract, task participants can vote for a specified hash through the vote function. The contract does not validate the legitimacy of the hash submitted by participants, meaning participants can construct and vote for hashes without actually running the client. Participating users can even observe on-chain voting patterns and selectively vote for favorable hashes to obtain rewards. This severely undermines the interests and motivation of honest participants.

Code location: contracts/federal_learning/FlockTask.sol#L312

```
    function vote(string memory _hash, int8 _score, int256 _initialLoss, int256
_aggLoss) external updateRound {
        ...
    }
```

Solution

If this is not intended by design, proof of work verification should be implemented for participants to mitigate these risks.

Status

Fixed

[N7] [Low] Allow voters to cast votes with a score of 0

Category: Design Logic Audit

Content

In the FlockTask contract, the vote function only allows voters to cast scores between -1 and 1, which means voters can submit a score of 0. For voters who cast a score of 0, when checked through canBeSlashed, the correctness of their voting score becomes irrelevant, as votedIncorrectScore will always return false. Therefore, as long as a voter who cast a score of 0 votes for the same hash as the final popularHash, they can receive rewards while avoiding being slashed.

Code location: contracts/federal_learning/FlockTask.sol#L314

```
    function vote(string memory _hash, int8 _score, int256 _initialLoss, int256
_aggLoss) external updateRound {
        require(participantRoles[currentRound][msg.sender] == uint(Role.VOTER), "Not a
voter");
        require(_score >= -1 && _score <= 1, "Invalid score");
        ...
    }
```

Solution

It is recommended to only allow scores of 1 and -1 to be entered.

Status

Fixed

[N8] [Medium] Potential pitfalls of the updateClaimableReward function

Category: Design Logic Audit

Content

In the FlockTask contract, participants can trigger the reward distribution mechanism through the updateClaimableReward function. Each time a round of rewards is distributed, the totalRewardedRounds variable increments by 1. Reward distribution is only possible when the current round number is greater than totalRewardedRounds and slashing has been completed. There are two scenarios for triggering updateClaimableReward:

1.Immediately calling updateClaimableReward after each round completion

2.Delaying updateClaimableReward calls for multiple completed rounds and then calling it multiple times at once to settle all completed rounds

Notably, these two different reward settlement methods produce different reward amounts even for the same number of rounds, with the second method generating more rewards than the first.

Code location: contracts/federal_learning/FlockTask.sol#L330

```
    function updateClaimableReward() external updateRound {
        require(nodeScore[msg.sender] > 0, "No score to claim");
        require(currentRound > 0, "No previous round");
        require(alreadySlashedRounds > currentRound - 1, "Slash not completed for
rounds");

        if (totalRewardedRounds < currentRound){
            uint256 _round = currentRound - 1;
            uint256 scale = 10 ** 18;
            uint256 roundsLeft = totalNumberOfRounds - _round;
            uint256 roundRewardAmount = rewardPool / roundsLeft;
            uint256 roundTotalStakesForGoodParticipants =
getTotalStakesForGoodParticipants(_round);
            ...
            totalRewardedRounds += 1;
        }
    }
```

Solution

It is recommended to clarify the design intentions to ensure the actual implementation aligns with the intended reward distribution model.

Status

Fixed

[N9] [Low] DoS risk if slash functions are not called for a long time

Category: Denial of Service Vulnerability

Content

In the FlockTask contract, any user can call the slash function and receive rewards. The slash function iterates through all unslashed rounds to perform slashing one by one. When a user is slashed, their nodeScore is deducted by the corresponding amount. Therefore, when the slash function slashes multiple rounds at once, a participant's nodeScore might be reduced to 0, and if the next slash iteration attempts to slash again, it will cause nodeScore to underflow, resulting in contract revert and potentially rendering the slash function unusable.

Code location: contracts/federal_learning/FlockTask.sol#L422,L436

```
function slash() external updateRound {
        ...
                if (totalScorePerRoundNegative) {
                    uint256 slashedAmount = (roundStakedTokens[alreadySlashedRounds]
[proposerAddress] * slashedPerStake) / 10 ** 18;
                    nodeScore[proposerAddress] -= slashedAmount;
                    totalNodeScore -= slashedAmount;

                    ...
                }
            }

            for (uint256 w = 0; w < selectedVoters[alreadySlashedRounds].length; w++)
{
                ...

                if (voterCanBeSlashed) {
                    uint256 slashedAmount = (roundStakedTokens[alreadySlashedRounds]
[voterAddress] * slashedPerStake) / 10 ** 18;
                    nodeScore[voterAddress] -= slashedAmount;
                    totalNodeScore -= slashedAmount;

                   ...
    }
```

Solution

Although the likelihood of this risk occurring is low due to the incentive of slash rewards, the project team should remain vigilant and ensure that slashing operations are performed promptly for each round.

Status

Fixed

[N10] [Suggestion] Gas optimization for slash functions

Category: Gas Optimization Audit

Content

In the FlockTask contract's slash function, when slashing is successful, the caller can receive a portion of the slashed amount as a reward, with the remaining slashed amount being accumulated in the rewardPool. It's worth noting that totalSlashedAmount is not always greater than 0 for each slash operation, therefore checking if slashReward is greater than 0 after slashing before distributing rewards would effectively save gas.

Code location: contracts/federal_learning/FlockTask.sol#L444-L448

```
    function slash() external updateRound {
        ...
        for (; alreadySlashedRounds < currentRound; alreadySlashedRounds++) {
            ...

            uint256 slashReward = totalSlashedAmount * bonusRatioForSlashCaller /
10000;
            nodeScore[msg.sender] += slashReward;
            totalNodeScore += slashReward;
            rewardPool += (totalSlashedAmount - slashReward);
            totalSlashReward += slashReward;
        }

        emit Rewarded(msg.sender, totalSlashReward, currentRound, "slasher");
    }
```

Solution

It is recommended to check if slashReward is greater than 0 after each round of slashing, and only proceed with reward distribution if it is positive.

Status

Fixed

[N11] [Critical] PopularHash is at risk of a 51% attack

Category: Design Logic Audit

Content

In the FlockTask contract, mostPopularHash is used to select the hash with the highest number of votes as the popularHash. The popularHash and its vote count are related to slashing and reward distribution. Unfortunately, there is no entry barrier for task participants, and the contract cannot verify the validity of hashes submitted by voters. This means malicious users can participate in tasks by staking the minStakeThreshold amount of tokens using multiple addresses. These dishonest addresses can manipulate their constructed hash to receive the highest number of votes, allowing them to slash other participants and profit from it.

Code location: contracts/federal_learning/FlockTask.sol#L509

```
    function mostPopularHash(uint256 _round) public view returns (string memory) {
        string memory popularHash = "";
        uint256 maxVoteCount = 0;
        for (uint256 i = 0; i < voters[_round].length; i++) {
            string memory voteHash = votes[_round][voters[_round][i]].voteHash;
            uint256 voteCount = votesHashCount[_round][voteHash];
            if (voteCount > maxVoteCount) {
                maxVoteCount = voteCount;
                popularHash = voteHash;
            }
        }
        return popularHash;
    }
```

Solution

N/A

Status

Fixed; After communicating with the project team, the project team said that we have added the commit-and-reveal solutions to mitigate the situation that VOTERs collude to vote a malicious hash as the most popular one. And we assume that honest participants can control at least 51% of the stake (e.g., through whitelisting etc).

[N12] [Medium] TotalNumberOfRounds is not checked when finishing a round

Category: Design Logic Audit

Content

In the FlockTask contract, the finishRound function is used to end the current round and advance to the next round, incrementing the currentRound variable. However, it does not check whether the new round number is still less than totalNumberOfRounds. Theoretically, when the number of rounds reaches the preset totalNumberOfRounds value, the current task should be completed, but this logic is not implemented in the finishRound function.

Code location: contracts/federal_learning/FlockTask.sol#L561

```
    function finishRound() internal {
        string memory popularHash = mostPopularHash(currentRound);
        string memory nextHash = globalModelHash[currentRound];
        int256 popularHashScore = hashScore[currentRound][popularHash];

        if (popularHashScore > 0) {
            nextHash = popularHash;
        }

        roundResult[currentRound] = RoundResult({popularHash: popularHash, voteScore:
popularHashScore, nextHash: nextHash});
        emit RoundFinished(currentRound, selectedProposers[currentRound],
selectedVoters[currentRound]);

        currentRound++;
        globalModelHash[currentRound] = nextHash;
    }
```

Solution

It is recommended to check if the current round number has reached totalNumberOfRounds when advancing to a new round. When totalNumberOfRounds is reached, the contract's finished state should be set to true and further participation should be prohibited.

Status

Fixed

[N13] [High] There is no order of precedence between contribute and vote

Category: Design Logic Audit

Content

In the FlockTask contract, once role assignments are completed for a round, PROPOSERs can contribute hashes, and VOTERs can vote immediately without waiting for PROPOSERs to finish contributing. This means VOTERs' votes can be completely unrelated to PROPOSERs' contributed hashes, and these two operations have no sequential relationship. This contradicts the client program's flow where voting must wait until all PROPOSERs have completed their contributions.

Code location: contracts/federal_learning/FlockTask.sol#L298-L325

```
    function contribute(string memory _hash) public updateRound {
        ...
    }

    function vote(string memory _hash, int8 _score, int256 _initialLoss, int256
 _aggLoss) external updateRound {
        ...
    }
```

Solution

It is recommended to only allow voting after all PROPOSERs have completed their contributions. To avoid DOS risk from PROPOSERs not contributing, a contribution deadline should be added. PROPOSERs who fail to contribute before the deadline would be slashed.

Status

Fixed

[N14] [Medium] Potential issue where the client cannot claim rewards normally

Category: Design Logic Audit

Content

In the FlockTask contract, users can claim rewards through the claimRewards function when either the number of rounds reaches totalNumberOfRounds or the finished status is true. However, it's important to note that the `isFinished` function uses `currentRound == totalNumberOfRounds` to check totalNumberOfRounds, and rounds can still continue beyond totalNumberOfRounds. This means that when finished is not set to true, the time window for users to claim rewards might be very short, and not all users may successfully claim their rewards.

Code location: contracts/federal_learning/FlockTask.sol#L483

```
function isFinished() public view returns (bool) {
    return finished || (currentRound == totalNumberOfRounds);
}
```

Solution

If this is not the intended design, it is recommended to limit the maximum number of rounds to totalNumberOfRounds.

Status

Fixed

5 Audit Result

Audit Number	Audit Team	Audit Date	Audit Result
0X002501240001	SlowMist Security Team	2025.01.17 - 2025.01.24	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 2 critical risks, 2 high risks, 6 medium risks, 2 low risks, and 2 suggestions. All the findings were fixed or acknowledged. The code was not deployed to the mainnet. Since the protocol's random number generation depends on the sequencer, there remain certain risks.

6 Statement