Blockchain 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 - FL Alliance Client, 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" 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 party 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.

In black box testing and gray box testing, we use methods such as fuzz testing and script testing to test the robustness of the interface or the stability of the components by feeding random data or constructing data with a specific structure, and to mine some boundaries Abnormal performance of the system under conditions such as bugs or abnormal performance. In white box testing, we use methods such as code review, combined with the relevant experience accumulated by the security team on known blockchain security vulnerabilities, to analyze the object definition and logic implementation of the code to ensure that the code has the key components of the key logic. Realize no known vulnerabilities; at the same time, enter the vulnerability mining mode for new scenarios and new technologies, and find possible 0day errors.

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:

NO.	Audit Items	Result
1	SAST	Passed
2	Business logic security audit	Passed

3 Project Overview

3.1 Project Introduction

This is the Python client section of the Flock protocol. Users can interact with on-chain contracts through the client to perform operations such as token staking, proposal submission, voting, and model processing.

3.2 Coverage

Target Code and Revision:

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: 8924fcf56ad5f73058d8792eb99625f0eae91fb8

Audit Scope:

```
client
├── blockchain
│   └── utils.py
├── blockchain_client.py
├── client.py
├── contracts
│   ├── __init__.py
│   ├── contract.py
│   ├── flock_task.py
│   └── flock_token.py
├── dataset
│   ├── __init__.py
│   ├── data_converter.py
│   ├── dataset.py
│   └── schema.py
├── enums.py
├── exceptions.py
├── managers
│   ├── __init__.py
│   ├── container_manager.py
│   ├── s3_storage_manager.py
│   └── sync_manager.py
└── task.py
```

3.3 Vulnerability Information

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

NO	Title	Category	Level	Status
N1	Incorrect error message matching	Business logic security audit	Low	Fixed
N2	Redundant code	Business logic security audit	Suggestion	Fixed
N3	Incorrect parameter type	Business logic security audit	Medium	Fixed
N4	Risks of arbitrary uploads to S3 buckets	Business logic security audit	Critical	Fixed
N5	Client initialization does not check if the staked amount is sufficient	Business logic security audit	Low	Fixed
N6	Lack of necessary checks when instantiating a contract	Business logic security audit	Suggestion	Fixed
N7	The integrity of the model is not verified	Business logic security audit	High	Fixed
N8	Necessary error checking is not performed	Business logic security audit	Medium	Fixed
N9	Potential risk of denial of service for voting	Business logic security audit	Medium	Fixed
N10	Risk of model poisoning	Business logic security audit	Critical	Fixed

4 Findings

4.1 Vulnerability Summary

[N1] [Low] Incorrect error message matching

Category: Business logic security audit

Content

In the FlockTask class, the update_reward function is used to call the on-chain `FlockTask.updateClaimableReward()` function for reward settlement. It uses try-except for error handling and

attempts to determine if a user has staked by matching the error message "No stake". However, in reality, the updateClaimableReward function throws the error message "No score to claim" when a user has insufficient stake, which doesn't match the error message being handled in the except block.

Similarly, in the contribute function of the FlockTask class, when the caught error message is "Not a proposer", the program throws a `RoundEndedError`. However, in the actual contract, it's the error message "Round has already finished" that indicates the current round has ended.

In the vote function of the FlockTask class, the error messages being handled have no correlation with the errors thrown by the on-chain contract. The caught error message "Not a proposer" is not defined in the on-chain vote function.

Furthermore, both the vote and contribute functions in the FlockTask class use the RoundEndedError class when throwing errors. However, in reality, not all reverts on-chain are due to rounds ending.

Code location: client/contracts/flock_task.py#L30

```
    def update_reward(self) -> dict:
        ...
        try:
            return self.bc.transact(self.contract.functions.updateClaimableReward())
        except CallRevertError as e:
            if "No stake" in e.message:
                raise InsufficientStake from e
            elif "No previous round" in e.message:
                return {}
            else:
                raise e
```

client/contracts/flock_task.py#L97

```
    def contribute(self, _hash: str) -> dict:
        try:
            return self.bc.transact(self.contract.functions.contribute(_hash))
        except CallRevertError as e:
            if "Not a proposer" in e.message:
                raise RoundEndedError from e
            else:
                raise e
```

```
        except TransactionRevertError as e:
            raise RoundEndedError from e
```

client/contracts/flock_task.py#L86-L91

```
    def vote(self, _hash: str, score: int, initial_loss: float, agg_loss: float) ->
dict:
        # Convert float to int
        initial_loss = self._convert_loss(initial_loss)
        agg_loss = self._convert_loss(agg_loss)

        try:
            return self.bc.transact(self.contract.functions.vote(_hash, score,
initial_loss, agg_loss))
        except CallRevertError as e:
            if "Not a proposer" in e.message:
                raise RoundEndedError from e
            else:
                raise e
        except TransactionRevertError as e:
            raise RoundEndedError from e
```

Solution

When handling error messages, they should match the error messages thrown by the on-chain contract.

Status

Fixed

[N2] [Suggestion] Redundant code

Category: Business logic security audit

Content

In `Dataset::__init__`, although it accepts a schema parameter, it is not actually used in the function.

`Dataset::_validate_data` directly uses `self.data` for validation instead of the passed-in data parameter.

Code location: client/dataset/dataset.py#L16

```
    def __init__(self, dataset_fh: IO, schema: dict) -> None:
        # self.schema = schema
        self.data = self._load_dataset(dataset_fh)
        # self._validate_data(self.data, schema)
        self.train_data, self.test_data = self._split_dataset(self.data)
```

```
def _validate_data(self, data: list[dict], schema: dict) -> None:
    ...
    try:
        validate(self.data)
        logger.info("Dataset validated.")
    ...
```

Solution

If this is not intended for future use, it is recommended to remove these redundant parameters.

Status

Fixed; The project team removed this module.

[N3] [Medium] Incorrect parameter type

Category: Business logic security audit

Content

In `Dataset::_load_dataset`, the received file_handle parameter is typed as `str`. However, in `__init__`, the passed dataset_fh is of type IO, resulting in a type mismatch.

Code location: client/dataset/dataset.py#L22

```
def __init__(self, dataset_fh: IO, schema: dict) -> None:
    # self.schema = schema
    self.data = self._load_dataset(dataset_fh)
    # self._validate_data(self.data, schema)
    self.train_data, self.test_data = self._split_dataset(self.data)

def _load_dataset(self, file_handle: str) -> list[dict]:
    return json.load(file_handle)
```

Solution

It is recommended to modify the file_handle type in `Dataset::_load_dataset` function to IO.

Status

Fixed; The project team removed this module.

[N4] [Critical] Risks of arbitrary uploads to S3 buckets

Category: Business logic security audit

Content

In the S3StorageManager class, the upload_parameters function is used to upload data to S3 buckets. Unfortunately, any user can use the `S3_SIGNER_URL` to upload files with arbitrary content. Malicious users could upload large quantities of files to exhaust the S3 bucket's storage space, or use it as a distribution point for malware.

Code location: client/managers/s3_storage_manager.py#L35-L50

```
    def upload_parameters(self, parameters: bytes) -> str:
        ...
        response = requests.post(self.S3_SIGNER_URL, json={"filename": _hash})
        if response.status_code == 400:
            if response.json()["error"] == "File already exists!":
                logger.info(f"Parameters with hash {_hash} already exists")
                return _hash
        elif response.status_code != 200:
            raise RuntimeError(f"Failed to get pre-signed URL: {response.json()}")

        presigned_url = response.json()["url"]
        response = requests.put(presigned_url, data=parameters)
        return _hash
```

Solution

It is strongly recommended to implement identity authentication during upload operations to prevent these risks.

Status

Fixed

[N5] [Low] Client initialization does not check if the staked amount is sufficient

Category: Business logic security audit

Content

In `Client::__init__`, necessary parameters for client operation are initialized, including the stake_amount parameter. When users stake through stake_if_needed, if stake_amount is greater than 0, it will be used as the amount to stake. However, during initialization, stake_amount is not checked against FlockTask's minimum staking requirement (minStakeThreshold). If the user sets an amount less than minStakeThreshold, staking will fail.

Code location: client/client.py#L73

```
    def __init__(
        self,
        ...
    ) -> None:
        ...
        self.stake_amount = int(stake * 10**18)
        self.dataset_path = dataset

        container_name = f"flock_model_{self.port}"
        self.container_manager = ContainerManager(
            dataset_path=self.dataset_path,
            container_name=container_name,
            port=self.port,
            use_gpu=bool(gpu),
            env_vars=env_vars,
        )
```

Solution

It is recommended to verify in `Client::__init__` whether the initialized stake_amount is greater than minStakeThreshold.

Status

Fixed

[N6] [Suggestion] Lack of necessary checks when instantiating a contract

Category: Business logic security audit

Content

In `Client::_initialize_contracts`, when the required `contracts.json` file exists and global variables token_address and task_address are not set, the program reads necessary parameters from `contracts.json`. Otherwise, it directly uses the global variables token_address and task_address to instantiate contracts, but it does not check whether both token_address and task_address are non-empty before doing so.

Code location: client/client.py#L109

```
    def _initialize_contracts(self) -> None:
        ...
        if os.path.exists("data/contracts.json") and not (self.token_address and
self.task_address):
            ...
```

```
        else:
            ...
```

Solution

It is recommended to verify that the global variables token_address and task_address are non-empty even when the `contracts.json` file does not exist.

Status

Fixed

[N7] [High] The integrity of the model is not verified

Category: Business logic security audit

Content

In `Client::start_model_container`, when an image doesn't exist, the program downloads the necessary image from the S3 bucket using the `_download_model_tar_and_build` function based on the hash, and directly starts a container with this image without performing any integrity checks. If the file integrity is compromised, this could cause the program's final results to deviate from expectations.

Code location: client/client.py#L123,L154

```
    def start_model_container(self) -> None:
        ...
        if not image:
            image = self._download_model_tar_and_build(_hash)

        self.container_manager.start_container(image)

    def _download_model_tar_and_build(self, _hash: str) ->
docker.models.images.Image:
        ...
            image = self.container_manager.build_image(path, _hash)
        return image
```

Solution

It is recommended to compare the hash of the downloaded file's content with the expected hash to verify file integrity after downloading from the S3 bucket.

Status

Fixed

[N8] [Medium] Necessary error checking is not performed

Category: Business logic security audit

Content

In the Client, when the client starts, it performs commit_metadata operation before staking and starting the container. However, the commit_metadata operation doesn't check if the transaction is successful and doesn't handle any errors, which could lead to the process continuing to run even when commit_metadata fails.

Similarly, in the loop function, when an error occurs after the program performs `join_round`, it assumes the task has started and waits for the current round to end through `wait_for_round_finish`. However, the `join_round` operation can also fail due to insufficient stake, maximum participants reached, and other errors. These errors don't necessarily mean the current round has started, so continuing to wait for the round to end through `wait_for_round_finish` is not feasible. This could cause the loop to become blocked at the `join_round` operation.

Code location:

client/client.py#L170

```
    def start(self) -> None:
        """
        Start the federated learning process.
        """
        self.commit_metadata()
        self.stake_if_needed()
        self.start_model_container()
        self.loop()

    def commit_metadata(self) -> None:
        dataset_size = os.path.getsize(self.dataset_path)
        metadata = {"dataset_size": dataset_size}
        self.flock_task.set_node_metadata(json.dumps(metadata))
```

client/client.py#L288-L293

```
    def loop(self) -> None:
        ...
            try:
                self.flock_task.join_round()
            except RoundAlreadyStarted:
                logger.info("Round already started, waiting for the next round to
join.")
                self.sync_manager.wait_for_round_finish(_round)
                Continue
        ...
```

Solution

It is recommended to implement necessary error catching and handle different errors differently to improve the program's robustness.

Status

Fixed

[N9] [Medium] Potential risk of denial of service for voting

Category: Business logic security audit

Content

In `Client::run_voter_flow`, the program checks whether all PROPOSERs in the current task round have completed their contributions through `wait_for_contributions`. However, it's important to note that if a PROPOSER never contributes, this will cause `wait_for_contributions` to either wait indefinitely or timeout. In reality, since there is no sequential relationship between contribute and vote operations in the smart contract, VOTERs can still vote and the round can end normally even if not all PROPOSERs have completed their contributions. However, this creates a DoS vulnerability for the client.

Code location: client/client.py#L359

```
    def run_voter_flow(self, _round: int) -> None:
        ...
        self.sync_manager.wait_for_contributions(_round)
        ...
```

Solution

As identified in the smart contract issues, there should be clear rules established for the sequential relationship between contribute and vote operations, along with necessary handling mechanisms.

Status

Fixed

[N10] [Critical] Risk of model poisoning

Category: Business logic security audit

Content

As identified in the smart contract audit regarding the 51% voting attack risk, VOTERs can collude to vote a malicious hash as the most popular one. Since the smart contract cannot verify the client's training workload, this type of attack is extremely easy to execute. Once such an attack occurs, clients will download and use the poisoned model for training, causing the training results to deviate from expectations.

Solution

It is recommended to implement proof-of-work verification to ensure that both contributed hashes and voted hashes are results of honest training by the clients.

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).

5 Audit Result

Audit Number	Audit Team	Audit Date	Audit Result
0X002501240002	SlowMist Security Team	2025.01.17 - 2025.01.24	Passed

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, 1 high risk, 3 medium risks, 2 low risks, and 2 suggestions. All

the findings were fixed.

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