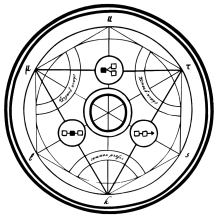
FLR Finance

FTSO Reward Manager Contract Audit

July 4, 2022

Common Prefix



Overview

Introduction

Common Prefix was commissioned to perform a security audit on Flr Finance's smart contracts FTSORewardManager. The contracts inspected are the following :

```
FlareFinanceFTSORewardManager.sol
FlareFinanceFTSORewardManagerProxy.sol
FlareFinanceFTSORewardManagerStorage.sol
WNatOwner.sol
WNatOwnerStorage.sol
```

Protocol description

Flare Network's Flare Time Series Oracle (FTSO) provides externally sourced data estimates to the Flare Network in a decentralized manner. Per the FTSO design several Data Providers are invited to participate in the FTSO procedure by submitting (voting for) truthful external (off-chain) data values to the Network's Oracles contracts. To ensure the robustness of the final data feed values, the Data Oracle processes all the submitted values, according to an algorithm which is described in detail here. In essence the algorithm discards the top and bottom 25% of submitted values and extracts the data estimate as the median of the remaining 50% of the feeds.

The value submission procedure can be thought of as a voting procedure where the voting power equals the corresponding Data Provider's SGB/FLR –governance token– (voting) balance. The higher the voting power of a Data Provider, the higher the weight that its suggested data value receives. Each native token carries one vote. For this design to be functional without disturbing the tokenomics of the native token, a wrapper contract for the native token enables the so-called "voting delegation" mechanism. This practically means that each native token holder can delegate her corresponding voting power (which equals native token balance) to any address while still keeping the ownership and management of her tokens. Holders of the Flare

Network's native token, SGB in the case of the Songbird testnet, are able to delegate the voting power of their funds to one or more Data Providers which participate in the FTSO procedure. The Data Providers use their own but also the delegated to them voting power to empower the data values they submitted to the Data Oracles (e.g. Price Oracles). The whole voting and data estimates extraction procedure happens in recurring rounds. Data Providers that contribute to this' procedure are compensated in each round with rewards in native token. In turn, native token holders who delegated their voting power to Data Providers, are compensated with a share of the accrued rewards. Only Data Providers, and their corresponding delegatorts, whose submitted values survived the 50% cut-off step are rewarded.

The total voting power of a Data Provider, and therefore the weight of its submitted values to the Oracle, depends on users' activity and, thus, changes constantly. It would be extremely gas inefficient for the FTSO protocol to synchronously track all these changes. Instead, a more conservatice approach is considered as follows. A number of price epochs take place within a single reward epoch (about a week), and during each price epoch the Data Providers submit their off-chain retrieved values. The voting power of each Data Provider is calculated only once per reward epoch and remains the same for each price epoch within the same reward epoch. Each user's stake, each Data Provider's total voting power as well as the participating users' total stakes are calculated before the start of each reward epoch. When the reward epoch starts, all of the voting-related data -which have been cached in all the previous blocks- are being retrieved from a block ("votePowerBlock") which is chosen randomly from the last 25% of blocks in the previous reward epoch (which we will call "votePowerBlock election period"). If a transaction changing the stake of an SGB holder happens during the votePowerBlock election period, the protocol cannot decide a priori if the new stake will be accounted for (if it happened before) in the reward epoch right after the votePowerBlock or (if it happened after the votePowerBlock) in the next next reward epoch instead. Therefore these transactions are indicated as pending and extra care is taken for them in the protocol.

FLRFinance provides a number of DeFi products that require funds locking. Such products are, for example, FLRLoans and Farming Pools. FLRFinance's contracts that accumulate locked SGB funds delegate voting power of these locked funds to FTSO Data Providers. This happens through the WNATOwner contract. All of FLR Finance's contracts who are supposed to delegate

voting power inherit from the `WNaTOwner` contract. Because of their participation in the FTSO procedures FLRFinance's contracts accumulate reward amounts as voting epochs pass by. The accrued rewards are distributed among the stakers - essentially, the users of FLR Finance protocols - in accordance to their locked stake. The accounting for the locked stakes and the corresponding reward shares is handled by the contract `FlareFinanceFTSORewardManager`. The `WNaTOwner` contract acts as an intermediary to FLR Finance and Flare Network in the FTSO process, including voting delegation and rewards claiming.

Related official documentation by Flare Network:
https://docs.flare.network/user/delegation/delegation-in-detail/#vote-power-block-selection
https://docs.flare.network/user/delegation/reward-claiming/

Security Opinion

The audited codebase has limited attack surface, since the two core contracts - `FlareFinanceFTSORewardManager` and `WNaTOwner` mainly contain functions of restricted access to trusted entities.

More specifically, `WNaTOwner` interacts with Flare Network's FTSO-related contracts for delegating voting power to Data Providers and claiming rewards. Functions for voting delegation are accessible only by the contract's owner. There are two functions for claiming rewards; claimRewardFromDataProviders(uint epochId) for claiming from the FTSO Data Providers, which is accessible only by the owner, and claimRewardAndSendToFTSORewardManager(uint epochId) for claiming from the FTSO RewardManager contract, which only claims the rewards and immediately transfers them to the FlareFinanceFTSORewardManager contract.

Contract `FlareFinanceFTSORewardManager` is mainly accessible via whitelisted entities; owner, AllowedCaller which is supposed to be FLRFinance DeFi products' contracts, AllowedRewardNotifier which is supposed to be the `WNaTOwner` contract.

Of course, some trust assumptions are needed by the users of the protocol that the trusted entities, which are configured by the contract's owner, consequently the owner itself behaves in accordance to the users' interest and not against them in any way.

It's worth noting that the rewards distributed to the users may not be 100% precise to their corresponding stake across the epochs, due to some subtleties of the FTSO voting process which result in a certain level of complexity when accounting for the exact stakes of each user in every single epoch. For a more detailed explanation of such inaccuracies refer to issue L1 below. We believe that such inaccuracies should be low enough and imply insignificant fairness issues to the reward distribution process. Another case of a slight inaccuracy is described in issue L2.

Disclaimer

Note that this audit does not give any warranties on the bug-free status of the given smart contracts, i.e. the evaluation result does not guarantee the nonexistence of any further findings of security issues. This audit report is intended to be used for discussion purposes only. Functional correctness should not rely on human inspection but be verified through thorough testing. We always recommend proceeding with several independent audits and a public bug bounty program to ensure the security of the project.

Findings Severity Breakdown

The findings are classified under the following severity categories according to the impact and the likelihood of an attack.

Level	Description
Critical	Logical errors or implementation bugs that are easily exploited and may lead to any kind of loss of funds
High	Logical errors or implementation bugs that are likely to be exploited and may have disadvantageous economic impact or contract failure
Medium	Issues that may break the intended contract logic or lead to DoS attacks
Low	Issues harder to exploit (exploitable with low probability), issues that lead to poor contract performance, clumsy logic or seriously error-prone implementation
Informational	Advisory comments and recommendations that could help make the

	codebase clearer, more readable and easier to maintain

Findings

Critical

None found.

High

None found.

Medium

None found.

Low

LOW-1	Pending withdrawal transactions may increase the reward rate in an inconsistent way
Contract(s)	`FlareFinanceFTSORewardManager.sol`
Status	**OPEN**

Description

A stake/withdraw transaction will be stored as pending if it is submitted within the vote power block election period. Consider that a transaction is submitted within reward epoch i. The staked/withdrawn amount of such a transaction is going to be accounted for the user's balance

either in epoch i+1 or i+2, depending on the randomly chosen vote power block of epoch i+1. As a result, the user's balance will be eventually properly updated on any following user's interaction with the protocol at a future epoch. However, the protocol needs to account the transacted amount for the total supply of stakes, so as to be able to calculate the new reward rate upon a reward amount update. What currently happens for both stake() and withdraw() is that the transacted amount is included in epoch i+1, no matter if it will be "moved" one epoch later in some cases.

```
/// stake(address user, uint amount):
// CP: stakeEpoch = getNextEpoch()
//all stakes from Thursday to Saturday also goes to the next epoch despite vote
//power block, this may lead to small mismatching in total numbers that should be
//equalized in long run
_updateTotalSupplyByEpoch(amount, stakeEpoch, true, true);

/// _updateRewardAmount(uint totalAmount, uint epoch):
uint rate =
    totalAmount * 1e18 / _getTotalSupplyAndCopyFromPreviousEpochIfNeeded(epoch);
```

In the case of stake() this is a conservative design: behave like we received the funds as early as possible (epoch i+1). In case they are eventually included in a later epoch (i+2), the rate will be slightly lower for epoch i+1, meaning that users will get slightly lower reward for that epoch.

However, in the case of withdraw() a problematic scenario arises: behave like we gave away the funds as early as possible (epoch i+1). In case they were actually accounted for in a later epoch (i+2), the rate will be slightly higher than it should be. This means that if all users claim their rewards for epoch i+1 there will be some users gaining higher rewards but also some failing claim transactions due to lack of funds.

Recommendation

We suggest adopting a conservative design also for the case of withdrawals, accounting the withdrawn amounts in the total supply in the latest epoch possible (i+2).

LOW-2	User may be unable to claim her rewards of one epoch
Contract(s)	`FlareFinanceFTSORewardManager.sol`

Status	**OPEN**

Description

In case a user withdraws her stake and rewards from an epoch and this epoch has earned only a part of its rewards, the user will not be able to claim the rest of it in the future since `isUserRewardsClaimedForEpoch[user][epoch]` is set true within `_claimReward` function. So there is a chance that a user loses her rewards for an epoch.

Recommendation

We noticed that mapping `epochPaidRewards` is updated on reward claiming and could be of help to this issue but is currently never actually used (except for logging). We suggest using this information to overcome the slight inaccuracy occurring in such cases.

Informational/Suggestions

INFO-1	Dead code
Contract(s)	`FlareFinanceFTSORewardManager.sol`
Status	**OPEN**

Description

Function FlareFinanceFTSORewardManager::updateRewardAmount is never called by the WNATOwner contract. Considering the current design, it also seems error-prone because it automatically considers that the notified reward amount concerns the previous epoch.

Recommendation

We suggest reverting the function `FlareFinanceFTSORewardManager::updateRewardAmount` because its functionality is not supported by the WNATOwner contract.

INFO-2	Code Simplification
Contract(s)	`FlareFinanceFTSORewardManager.sol`
Status	**OPEN**

Description

Function signature of `FlareFinanceFTSORewardManager::userStakesForEpoch` defines the return variable but is never used:

```
function userStakesForEpoch(address user, uint epoch) public view returns (uint stakeForEpoch) {
    // CP: use the return variable instead
    (int stk, ) = userStakesForEpochWithLastStakeEpoch(user, epoch);
    return uint(stk);
}
```

Recommendation

We suggest simplifying the body of the function body by using the named return variable.

INFO-3	Code Simplification & Gas Optimization
Contract(s)	`FlareFinanceFTSORewardManager.sol`
Status	**OPEN**

Description

Function `FlareFinanceFTSORewardManager::userStakesForEpochWithLastStakeEpoch` performs some calculations twice for a specific epoch:

```
function userStakesForEpochWithLastStakeEpoch(address user, uint epoch) public view
returns (int stakeForEpoch, uint lastStakeEpoch) {
    int currentEpochStake = userActualStakesForEpoch[user][epoch];
    if (currentEpochStake == -1){
       lastStakeEpoch = epoch;
    } else if (currentEpochStake > 0) {
        stakeForEpoch = currentEpochStake;
        lastStakeEpoch = epoch;
    // CP: the following for loop also takes care of 'epoch'
    // CP: the above branches if/else-if can be removed
    } else if (epoch != 0) {
        for (int i = int(epoch); i >= int(userFirstEpoch[user]); i--) {
            stakeForEpoch = userActualStakesForEpoch[user][uint(i)];
            if (stakeForEpoch == -1) {
                stakeForEpoch = 0;
                lastStakeEpoch = uint(i);
                break;
            } else if (stakeForEpoch > 0) { //return stakeForEpoch
                lastStakeEpoch = uint(i);
                break;
            }
        }
    }
}
```

Recommendation

We suggest simplifying the function `userStakesForEpochWithLastStakeEpoch` for clarity and gas savings.

INFO-4	Gas optimization
Contract(s)	`FlareFinanceFTSORewardManager.sol`
Status	**OPEN**

Description

Function FlareFinanceFTSORewardManager::_processPendingTransactions is responsible to resolve any pending transactions of previous epochs, by placing them in the proper voting epoch, let's call them epochs i and i+1, according to whether they were submitted before or after of the corresponding vote power block. This is implemented by identifying the first transaction placed in a block which follows the vote power block, accounting the stake of all the pending transactions up the identified one to epoch i and the rest of them to epoch i+1.
When `userLastEpochWithPendingTransactions` is incremented to denote the epoch i+1, code that is intended to perform the stakes accounting for epoch i keeps being executed. This issue doesn't imply a bug at the moment, though it is error-prone. At the moment it essentially results in increased gas costs.

```
/// FlareFinanceFTSORewardManager::_processPendingTransactions
if (epochPendingTransactions[i].block > epochVotePowerBlock) {
    // CP: the following 'if' branch will be executed for every
    // CP: block > epochVotePowerBlock
    // CP: although it is intended to be executed only for the very first one
    if (i > 0) {
        userActualStakesForEpoch[user][userLastEpochWithPendingTransactions] =
            epochCollectedBalance > 0 ?
            epochCollectedBalance : -1;
        emit UpdateEpochWithPendingTransactions(userLastEpochWithPendingTransactions,
epochCollectedBalance);
    }
    if (!transactionEpochWasIncrementedAfterVotePowerBlock) {
        userLastEpochWithPendingTransactions++;
        transactionEpochWasIncrementedAfterVotePowerBlock = true;
    }
}
```

Recommendation

We suggest altering the if branch condition from `f (epochPendingTransactions[i].block > epochVotePowerBlock)` to `f (epochPendingTransactions[i].block > epochVotePowerBlock && !transactionEpochWasIncrementedAfterVotePowerBlock)` for gas savings and clarity.

INFO-5	Gas optimization
Contract(s)	`FlareFinanceFTSORewardManager.sol`
Status	**OPEN**

Description

In function FlareFinanceFTSORewardManager::_getEpochBalanceOf user's first stake epoch is read from storage in each loop iteration:

```
/// FlareFinanceFTSORewardManager::_getEpochBalanceOf
for (int i = int(epoch); i >= int(userFirstEpoch[user]); i--) { ... }
```

Recommendation

We suggest defining and using a local variable instead, for gas savings.

INFO-6	Move proxy-impl contracts' state variables in common to the storage contract
Contract(s)	`FlareFinanceFTSORewardManager.sol`, `FlareFinanceFTSORewardManagerProxy.sol`, `FlareFinanceFTSORewardManagerStorage.sol`
Status	**OPEN**

Description

State variable target is defined in both contracts `FlareFinanceFTSORewardManager.sol` and `FlareFinanceFTSORewardManagerProxy`, although they both inherit `FlareFinanceFTSORewardManagerStorage.sol` contract and it would be more suitable for it to be declared therein.

Recommendation

We suggest moving the definition of variable target to `FlareFinanceFTSORewardManagerStorage.sol` contract for clarity.

INFO-7	Code Simplification
Contract(s)	`FlareFinanceFTSORewardManager.sol`
Status	**OPEN**

Description

Functions `FlareFinanceFTSORewardManager::stake` and `FlareFinanceFTSORewardManager::withdraw` call `_processPendingTransactions(user)` before performing any accounting, so that all of the user's pending transactions are resolved. Despite this fact, the balance of the user is retrieved as follows:

```
// CP: parameter checkPendingTransactions == true
uint userBalance = _getBalanceOf(user, true);
```

_getBalanceOf is asked to look into pending transactions though they have already been taken care of _processPendingTransaction.

Recommendation

We suggest calling _getBalanceOf with parameter `checkPendingTransactions` set to false for clarity and gas savings.

About Common Prefix

Common Prefix is a blockchain research, development, and consulting company consisting of a small number of scientists and engineers specializing in many aspects of blockchain science. We work with industry partners who are looking to advance the state-of-the-art in our field to help them analyze and design simple but rigorous protocols from first principles, with provable security in mind.

Our consulting and audits pertain to theoretical cryptographic protocol analyses as well as the pragmatic auditing of implementations in both core consensus technologies and application layer smart contracts.