Flare Network

Smart Contract Audit

coinspect

Flare

Flare TDE Updates

Smart Contract Audit

V221223 Prepared for Flare • December 2022

1. Executive Summary

In **December 2022, Flare** engaged Coinspect to perform a source code review of a set of modifications performed to the **Flare Network smart contracts**. The objective of the project was to evaluate the set of contracts that will be used for the Token Distribution Event (TDE).

The new features allow account delegations and include changes related to the handling of reward management and distribution. Also, an execution (relay) logic was added to enable trusted third parties to perform actions on behalf of other accounts. It is worth observing that users must trust these third parties to execute claims on their behalf as expected as the smart contracts reviewed do not guarantee or enforce this. As a result, user's rewards could be lost if not claimed by the executor as expected.

The following issues were identified during the initial assessment:

High Risk	Medium Risk	Low Risk
Open **0**	Open **0**	Open **0**
Fixed 0	Fixed 0	Fixed 5
Reported 0	Reported 0	Reported 5

Coinspect identified five low-risk issues and one informational issue. **FLR-26** shows how users are forced to pay executors fees multiple times. **FLR-27** illustrates how the implementation of the governance contract could be taken over. **FLR-28** shows how non standard tokens would remain locked inside the delegation accounts. **FLR-29** describes how users could manipulate withdrawal events emissions via reentrancy. **FLR-30** suggests using require instead of assert to check for valid

conditions that cannot be detected until execution time. The informational issue **FLR-31** recommends implementing a storage gap to prevent proxy collisions.

2. Assessment and Scope

The audit started on **December 12, 2022** and was conducted on the **flare_code_to_audit** branch of the git repository at https://gitlab.com/flarenetwork/flare-smart-contracts as of commit **03678e8f47eae988c654ef6858dcf14d06a31476** of **December 20, 2022**.

The main audited files have the following `sha256sum` hash:

```
7a66633326deb8df12f142e70abdd68dc57b08ee3128522b340024fbbc99774b  ClaimSetupManager.sol
f98370c7a4369a58986bea2486d0a13bea663524ba2ce003cd2f6ff25b7692eb  FtsoRewardManager.sol
dbe4e5c99777121679d0e67874a091f97538e1a85754016519da22d5d928a56a  ProxyGoverned.sol
8c7b0380d29c6ab5b45c3ff3e649cf03907b4ad07d5f2d323ae5ef3568a71ac1  FtsoRegistryProxy.sol
9966a2c8b3a02c75d13cf5423da738680ca86119f5d1be64447f47d8cc2d68b9  FlareAssetRegistry.sol
6bebbd18048a59670355034ea1276484a74b567f8cef595a428c07ed5131b04c  WNatRegistryProvider.sol
7bec42cfc55bc0db68f612907ff870d21e20c52710db16f1ea0c5b8813a262d9  FtsoRegistry.sol
149edc2a2d7d4f925bf4ea3df0766f14658ea29b9a2535772d9ccab78325534c  FtsoManager.sol
bee8e445edb09961149fe28a2290152f185e9e6f143d673a260acfebf36dc1f4  Ftso.sol
```

This audit focused on the changes introduced to the smart contract by the following merge requests:

- https://gitlab.com/flarenetwork/flare-smart-contracts/-/merge_requests/579
- https://gitlab.com/flarenetwork/flare-smart-contracts/-/merge_requests/581
- https://gitlab.com/flarenetwork/flare-smart-contracts/-/merge_requests/583

Overall the code quality was high, changes were well documented and easy to understand. However, Coinspect identified that assert was used instead of require on places that could revert under expected operative scenarios, instead of code or invariant errors **(FLR-30)**.

The main change introduced is a new flow for claiming `FTSO` rewards, where a token holder can claim rewards through a third party executor using the new `ClaimSetupManager` contract. Each user can create a delegation account that handles token transfers and voting power delegation. Coinspect observed how non standard tokens could remain locked inside the delegation accounts as boolean returns were expected **(FLR-28)**.

The reward system is epoch based meaning that the granularity of the calculation depends on the epoch length. Rewards for a user can be self-claimed or claimed by a trusted executor (claim relayer). Moreover, the same behavior occurs for FTSO

providers where they can designate an executor. Receivers can also enable an auto claim for a specified list of epochs which will be handled by previously set executors.

The reward manager allows an inlet of tokens coming from the inflation manager that will be later assigned as the distributed rewards. Also, unclaimed rewards for past epochs will be also burned while injecting rewards via inflation. Once the rewards are accounted for and assigned to a user, they could be either transferred as wrapped tokens or directly as native to the receiver. The rewarding contract implements a modifier that ensures rewards balance which is checked before performing any claim. In addition, the rewarding manager contract has the privilege to set and modify key parameters.

Users can enable delegation accounts through `ClaimSetupManager`. Executors can register in this manager and later be set by users to perform relayed rewards claimings. Executors set a relaying fee upon registration (which can be later updated) and users have to pay that fee when assigning an executor. The fee cannot be changed for the current epoch and has an offset (cooldown) on which it can be changed to prevent frontrunning attacks. Moreover, the execution fees have upper and lower boundaries enforced by the contract to prevent outstanding charges. Delegation accounts are also managed by this contract allowing owners of those accounts to handle the rewards previously claimed. The rewarding system relies on the `onlyOwnerOrExecutor` modifier to control actions that can only be performed by the owner or a previously set executor. Coinspect identified that users cannot set newer executors without having to pay again the fee to previously set executors **(FLR-26)**.

It is worth mentioning that users pay executors upon assignment instead of once they perform the relayed call. Those who decide to pay an executor, must know that there's nothing stopping them from not relaying their rewards claiming calls. As a result, user's rewards would be lost if not claimed by the executor as expected.

The FTSO's are registered under the `FTSORegistry` contract which is upgradable and allows the manager to add and remove oracles. Moreover, prices could be retrieved via its implementation. This contract is deployed behind a proxy (`FTSORegistryProxy`) which is a governed custom proxy contract. Coinspect found

that the implementation of the base contract's Governance was prone to a take-over because it allowed a public initialization **(FLR-27)**. Also, related to the withdrawal process, **FLR-29** shows how events could be manipulated with reentrancy.

The `ProxyGoverned` is a proxy contract inherited by the `FtsoRegistryProxy` contract. This proxy (and its childs, e.g. `Governed`) have several functions that are access controlled. Any call to an access controlled function made by a non privileged account will fail instead of being forwarded. The Transparent Proxy Pattern suggests to forward any call coming from a non privileged account, instead of triggering a reversal. There are several functions controlled by the `onlyGovernance` modifier that will not be forwarded and will revert. Following the mentioned pattern is an alternative to prevent this. For the storage layout, both the proxy and the implementation inherit from the `GovernedBase` contract. This allows the implementation to use the same governance set in the proxy and manage all governance from the same place. But this also fixes the `GovernedBase` account, making this part of the code really hard to update in the future. Also, Coinspect recommended using a storage gap to prevent collisions **(FLR-31)**.

3. Summary of Findings

Id	Title	Total Risk	Fixed
FLR-26	Multiple fee payments for previously set executors	Low	✔
FLR-27	Governance implementation takeover	Low	✔
FLR-28	Non standard tokens can get locked in DelegationAccount contracts	Low	✔
FLR-29	Withdrawal events manipulation with reentrancy	Low	✔
FLR-30	Assert misusage	Low	✔
FLR-31	ProxyGoverned storage layout lacks safety gap	Info	!

4. Detailed Findings

FLR-26 Multiple fee payments for previously set executors

Total Risk	Impact	Location
Low	Low	`ClaimSetupManager.sol`
Fixed ✔	Likelihood Low	

Description

Users pay the executors their fee upon assignment. Because the setting mechanism replaces all the already set executors, users are forced to re-pay the `executorFee` to previously set executors even if they want to manage only one of them.

The `_setClaimExecutors` implementation only admits passing a list of executors, replacing previous registrations:

```
function _setClaimExecutors(address[] memory _executors) internal {
    // replace executors
    ownerClaimExecutorSet[msg.sender].replaceAll(_executors);
    emit ClaimExecutorsChanged(msg.sender, _executors);
    uint256 totalExecutorsFee = 0;
    for (uint256 i = 0 ; i < _executors.length; i++) {
        uint256 executorFee = getExecutorCurrentFeeValue(_executors[i]);
        if (executorFee > 0) {
            totalExecutorsFee += executorFee;
            /* solhint-disable avoid-low-level-calls */
            //slither-disable-next-line arbitrary-send-eth
            (bool success, ) = _executors[i].call{value: executorFee}(""); //nonReentrant
            /* solhint-enable avoid-low-level-calls */
            require(success, ERR_TRANSFER_FAILURE);
        }
    }
  // THE FUNCTION CONTINUES HERE ...
 }
```

Users cannot modify or add just one executor to relay their reward claims and are forced to pass again the whole list of executors, paying them the fee:

1) Alice adds a list of executors: `[addr(1), addr(2), addr(3)]`
2) Then she decides to add the executor `addr(4)`.
3) Alice is forced to pass the following array, paying again to the first three executors: `[addr(1), addr(2), addr(3), addr(4)]`

As a result, Alice paid twice to the first three executors in order to add a fourth one.

Recommendation

Allow users to add and remove single executors.

Status

Fixed.

Executors are now only paid the first time they are registered, in commit `cd10aac03a3e7f8f235a9e7941dac197f97ad46f` (`merge request !581`).

FLR-27 Governance implementation takeover

Total Risk	Impact	Location
Low	Low	`FTSORegistry.sol`
Fixed ✔	Likelihood Medium	

Description

The `FTSORegistry` contract inherits `GovernedBase` and uses the `address(0)` as a deployment parameter. As a consequence, the governance implementation contract will remain uninitialized allowing anyone to proceed with its initialization passing a custom governance contract.

The registry contract has the following constructor:

```
constructor() GovernedBase(address(0)) AddressUpdatable(address(0)) {
        /* empty block */
 }
```

Moreover, the `GovernedBase` contract has the following deployment and initialization logic:

```
constructor(address _initialGovernance) {
     if (_initialGovernance != address(0)) {
         initialise(_initialGovernance);
     }
 }

 function initialise(address _initialGovernance) public virtual {
     require(initialised == false, "initialised != false");
     initialised = true;
     initialGovernance = _initialGovernance;
     emit GovernanceInitialised(_initialGovernance);
 }
```

The governance contracts will point to the same address once they are switched to production mode via `BaseGovernance.switchToProductionMode()`, changing the retrieved governance address:

```
   function governance() public view returns (address) {
          return productionMode ? governanceSettings.getGovernanceAddress() :
initialGovernance;
    }
```

Although the governance contracts will point to the same address while in production mode, **depending on how the retrieved governance address is used by the caller while on initial stages uncertain scenarios may arise.**

Recommendation

Initialize the base governance contract with a known address.

Status

Fixed.

The deployment now initializes the implementation with the dead address:

```
constructor() GovernedBase(DEAD_ADDRESS) AddressUpdatable(address(0)) {
        /* empty block */
    }
```

Fix located in commit `051ef3241af5f7f879b91b4f175f27947f1e1570` (`merge request !581`).

FLR-28	Non standard tokens can get locked in DelegationAccount contracts

Total Risk	Impact	Location
Low	Low	`DelegationAccount.sol`
Fixed ✔	Likelihood Low	

Description

Non standard ERC tokens that do not have a boolean return will remain locked inside the `DelegationAccount` contract.

The delegation accounts handle token transfer with the following function:

```
   function transferExternalToken(WNat _wNat, IERC20 _token, uint256 _amount)
external override onlyManager {
          require(address(_token) != address(_wNat), "Transfer from wNat not
allowed");
        bool success = _token.transfer(owner, _amount);
        require(success, ERR_TRANSFER_FAILURE);
        emit ExternalTokenTransferred(_token, _amount);
    }
```

In the event of receiving ERC tokens that do not return a boolean value on transfer (such as BNB and USDT in other chains), the `require` statement checking the state of the transfer will never be true, reverting the external token transfer.

Recommendation

Use a safe transfer library when performing transfers of arbitrary tokens.

Status

Fixed.

Arbitrary tokens are now handled with `safeTransfer` in commit `051ef3241af5f7f879b91b4f175f27947f1e1570` (`merge request !581`).

FLR-29 Withdrawal events manipulation with reentrancy

Total Risk	Impact	Location
Low	Low	DelegationAccount.sol
Fixed ✔	Likelihood Low	

Description

The withdrawal methods for both native and ERC tokens emit an event after performing the transfer. As a result, users could reenter the withdrawal method and the event emitted will only account the amount for a single call.

```
    function withdraw(WNat _wNat, uint256 _amount) external override onlyManager {
        bool success = _wNat.transfer(owner, _amount);
        require(success, ERR_TRANSFER_FAILURE);
        emit WithdrawToOwner(_amount);
    }

       function transferExternalToken(WNat _wNat, IERC20 _token, uint256 _amount)
external override onlyManager {
        require(address(_token) != address(_wNat), "Transfer from wNat not allowed");
        bool success = _token.transfer(owner, _amount);
        require(success, ERR_TRANSFER_FAILURE);
        emit ExternalTokenTransferred(_token, _amount);
    }
```

Users cannot interact with the mentioned functions directly. They first need to create one via `ClaimSetupManager.enableDelegationAccount()` or `ClaimSetupManager.setAutoClaiming()`. Both calls end up executing the internal `_getOrCreateDelegationAccountData()` function that clones a Delegation Account and initializes its owner as the `msg.sender`.

Once the delegation account for the owner is deployed, it can be managed via the `ClaimSetupManager` contract:

```
  function withdraw(uint256 _amount) external override {
      _getDelegationAccount(msg.sender).withdraw(wNat, _amount);
  }
```

```
  function transferExternalToken(IERC20 _token, uint256 _amount) external override {
                _getDelegationAccount(msg.sender).transferExternalToken(wNat,  _token,
_amount);
    }
```

The owner of the delegation account can be a contract that:

1) Has 10 HTokens (ERC777 hookable tokens) in the Delegation Contract (Contract D).
2) Calls `claimSetupManager.transferExternalToken(1 Ntoken)`
3) Via the token hook logic made in the owner contract, reenters again with step 2)
4) As long as there is token balance in the delegation account contract, the step 3) is repeated.
5) The event emitted would be: `ExternalTokenTransferred(HToken, 1 token)` where the owner transferred 10 tokens instead.

Recommendation

Emit the event before making the transfer.

Status

Fixed.

In commit `051ef3241af5f7f879b91b4f175f27947f1e1570` (`merge request !581`). The event is now emitted before performing the transfer. Also, the `nonReentrant` modifier in `ClaimSetupManager.transferExternalToken()` was added.

FLR-30 Assert misusage

Total Risk	Impact	Location
Low	Low	`FtsoRewardManager.sol`
Fixed ✔	Likelihood Low	

Description

The `require` statement does not use up the remaining gas whereas the `assert` statement wasting all the remaining gas sent for the call. Several locations of the project use the `assert` statement to handle errors that could happen because of several operative situations (for example, if a non privileged account calls a function). As a result, an error of type `Panic(uint256)` is triggered instead of an error type `Error`.

The Solidity documentation states:

"The `assert` function should only be used to test for internal errors, and to check invariants. If this happens there is a bug in your contract which you should fix.

The `require` function should be used to ensure valid conditions that cannot be detected until execution time. This includes conditions on inputs or return values from calls to external contracts."

For example, in the `FtsoRewardManager` contract:

```
  function _checkExecutorAndAllowedRecipient(address _rewardOwner, address _recipient) private
view {
        if (msg.sender == _rewardOwner) {
            return;
        }
           assert(claimSetupManager.checkExecutorAndAllowedRecipient(msg.sender, _rewardOwner,
_recipient));
    }
```

It is a possible (yet not a code bug) scenario for a user to trigger a claim without being the owner or executor.

Recommendation

Replace `assert` for `require` in any other place where it applies, according to the Solidity Docs.

Status

Fixed.

In commit `051ef3241af5f7f879b91b4f175f27947f1e1570` (`merge request !581`). Now `ClaimSetupManager.checkExecutorAndAllowedRecipient()` does not return a boolean value and the assert was removed from `FtsoRewardManager._checkExecutorAndAllowedRecipient()` relying on the require statements located in `checkExecutorAndAllowedRecipient()`.

FLR-31	ProxyGoverned storage layout lacks safety gap		
Total Risk **Info**	Impact -	Location `ProxyGoverned.sol`	
Fixed !	Likelihood -		

Description

An upgrade of the `ProxyGoverned` contract that does not take into account the previously established storage layout might corrupt data breaking the proxy and leading to uncertain scenarios.

The `ProxyGoverned` contract inherits from `Governed` (thus `GovernedBase`) their variables and consequently the storage layout. The implementation's layout should match the proxy's appending new variables to the end of the layout. Moreover, a storage gap is commonly suggested to prevent the impact mentioned before.

Recommendation

Implement a safety storage gap to prevent storage collisions. Append newer variables to the end of the layout while updating the implementation.

Status

Won't fix.

The Flare Team stated that they are not willing to upgrade that contract: "*We think we should leave the code as is. `FtsoRegistryProxy` contract will never change (that's why we need proxy in this case) and the `FtsoRegistry` uses the storage variables from `GovernedBase` anyway, so we cannot just change the storage layout when we redeploy. So we think there shouldn't be any need for the storage gap.*"

5. Disclaimer

The information presented in this document is provided "as is" and without warranty. The present security audit does not cover any off-chain systems or frontends that communicate with the contracts, nor the general operational security of the organization that developed the code.