FLR Finance LoansStable

Smart Contract Audit

February 21, 2022

Common Prefix

Overview

Introduction

Common Prefix was commissioned to perform a security audit on Flr Finance's `LoansStable` smart contracts, at commit hash b89ee65e3c8620a02a234f8a7980ceec688ee6df. The files inspected are the following:

```
RewardTokenStakingStorage.sol

StabilityPool.sol

StabilityPoolRewardTokenManagerStorage.sol

Dependencies/FlareLoansStableBase.sol

Dependencies/AggregatorV3Interface.sol

Dependencies/BaseMath.sol

Dependencies/FlareLoansStableBaseStorage.sol

Dependencies/FlareLoansStableMath.sol

NestManagerStorage.sol

NestManagerBase.sol

GasPool.sol

YUSDTokenStorage.sol

StabilityPoolStorage.sol

YUSDToken.sol

NestManagerMain.sol

StabilityPoolRewardTokenManager.sol

BorrowerOperationsWNat.sol

RewardTokenStaking.sol

ActivePoolStorage.sol

BorrowerOperations.sol
```

```
SortedNests.sol

DefaultPoolStorage.sol

SortedNestsStorage.sol

BorrowerOperationsStorage.sol

LoansStableSettings.sol

ActivePool.sol

StabilityPoolRewardTokenManagerUpgraded.sol

MultiNestGetter.sol

NestManagerLiquidations.sol

CollSurplusPool.sol

CollSurplusPoolStorage.sol

DefaultPool.sol

HintHelpers.sol

Proxy/*
```

Description of the protocol

FlareLoans is a decentralized lending protocol that allows owners of supported crypto assets to obtain liquidity against their collateral with a low collateral ratio (CR). The main operations of the protocol are, briefly, the following:

- A user can open a position by depositing an amount of collateral token. Positions are called `Nests`. By opening a new Nest one borrows freshly minted YUSD, the stablecoin created by the protocol. The tokenomics ensuring the stability of YUSD are handled by the protocol and are based on economic incentives. The owner of a Nest is required to maintain a minimum Collateral Ratio (CR), the threshold value being set at 110%. If a Nest's CR falls under this threshold, it can be liquidated. YUSD tokens can be exchanged and traded. The minimum amount of borrowing is defined at 2000YUSD. There is a liquidation fee of 100 YUSD upon opening a Nest, which is returned to the user upon closing the Nest, unless her position is liquidated.

- Any holder of YUSD can redeem her tokens against one or more Nests. When a redemption takes place the YUSDs involved cancel out an equal amount of debt from the participating Nests and the corresponding amount of collateral is transferred from the Nest to the redeemer. Participating Nests are the ones with the lowest CR, unless they are liquidable.

- YUSD holders can stake their tokens to the Stability Pool and become liquidity providers. Stability Pool's deposits are used during liquidations of undercollateralized Nests, or simply increase their CR. As a reward the liquidity providers get a share of the liquidated collateral but also receive reward tokens.

- Upon opening a Nest and redeeming YUSD tokens a (YUSD) fee is applied. A percentage of these fees is transferred to the RewardTokenStaking contract. Holders of reward tokens can stake them to this contract and accrue extra YUSD rewards, which a portion of the accrued fees.

- If the total CR of the system becomes lower than 150% the Recovery Mode is initiated. During the Recovery Mode borrowing and liquidation CR thresholds become stricter, so that Stability Providers are incentivized to increase their deposits and borrowers to improve their CR.

Similarities/Differences with Liquity

FlareLoans is based on defi protocol Liquity. Their differences are summarized as follows:

- Liquity supports only ETH as a collateral token. FlareLoans supports several tokens as collateral, although **not** altogether in one unified lending ecosystem. For each supported collateral token a different ecosystem, with its own copy of smart contracts, is constructed. The only common place is the YUSD token contract.
- FlareLoans contracts are upgradeable via a proxy pattern.
- [Technical] Higher accuracy computation of the product factor P (used for the computation of an SP provider's compounded deposit and the corresponding rewards). In FlareLoans, scale changes when P becomes less than 1e-9, compared to 1e-18 in Liquity. As a result, higher accuracy calculation of P is achieved, since scale-in occurs

more frequently. However, due to this change, the deviation between the “real” value of P and the one used by the protocol increases to the order of 1e-9 (compared to Liquity’s 1e-18). We note that this problem could be solved if a second change of scale for the deposits and rewards computation is taken into consideration.

Executive Summary

The audited contracts lie under upgradable proxies. The Owner can upgrade the underlying implementation at any time. Apart from that there are a number of onlyOwner functions that set sensitive protocol parameters, most notably Minimum Collateral Ratio of a Nest (MCR), the Critical Collateral Ratio of the system (CCR), the minimum net debt that a Nest is required to hold, YUSD compensation for liquidators and fees distribution between Kakeibo and ReawardTokenStaking contract. Issue Low-2 refers to security threats due to the enhanced centralization inserted by the upgradeability mechanism.

We were provided some helpful documentation, though it is rather incomplete. Some of the functionality of the protocol is merely referred to (e.g. staking of the reward tokens). In many places Liquity’s terminology is used (e.g. LUSD instead of YUSD, LQTYStaking instead of RewardTokenStaking) but also Liquity’s parameters’ values (e.g. liquidation reserve equals 100 YUSD in LoansStable but is documented to be 200 YUSD in the docs). The documentation is also not clear or consistent about the tokens in the roles of collateral and reward.

The audited codebase was delivered with a rather poor set of tests. We strongly advise thorough testing to be performed. The purpose of this audit is complementary to that of a good testing suite with high code coverage. Thorough testing should provide a good level of confidence that the codebase functions as intended, while this audit aims to provide security confidence under the scope of blockchain-related functionality and threats.

Another point that is worth to be noted is that for the price feeds Flare Time Series Oracle will be used. However, this part is out of this audit’s scope.

We found no critical, high or medium severity issues, which denote an overall healthy protocol and implementation. However, the codebase seems to need quite a serious maintenance work and this is denoted by the large number of informational/suggestions items that we file in this report.

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	Debt offset() fails silently in the absence of YUSD deposits
Contract(s)	`StabilityPool`
Status	**Resolved**

Description

While in Recovery Mode a liquidation may trigger an amount of a Nest's debt to be handled by the Stability Pool (debt offset). In StabilityPool::offset, if the total YUSD deposits in the pool is zero then the function fails silently:

```
function offset(uint _debtToOffset, uint _collToAdd) external override {
       _requireCallerIsNestManager();
       uint totalYUSD = totalYUSDDeposits;

        // CP: fails silently if total YUSD deposits are zero
       if (totalYUSD == 0 || _debtToOffset == 0) { return; }

       // ...

       // CP: burns YUSD, among other
       _moveOffsetCollAndDebt(_collToAdd, _debtToOffset);
}
```

In this way the caller's side always assumes that the function execution completes successfully. This means that the protocol may come to proceed based on the assumption that some amount of YUSD gets burnt, while it's not. As a consequence, the tokenomics of the stablecoin could be threatened. While at the current version of the codebase each call to StabilityPool::offset calculates _debtToOffset amount in respect to the pool's total YUSD deposits, this is definitely an error-prone implementation. More specifically, if _debtToOffset, _collToAdd are non zero amounts and totalYUSDDeposits equals zero then the transaction should revert. On the other hand, if _debtToOffset, _collToAdd equal zero then the function can securely just return.

Recommendation

We suggest that StabilityPool::offset revert in case of zero YUSD deposits and non-zero parameters’ values, and simply return when the parameters’ values are zero.

LOW-2	High centralization makes it possible to drain users’ funds
Contract(s)	`[whole ecosystem]`
Status	**Dismissed**

Description

An adversary with privileges could replace any contract in the system with an arbitrary one. This could, in a worst case scenario, mean draining all user’s funds, both collateral and YUSD tokens. For the YUSD tokens specifically, the FlareLoans core-contracts are privileged to perform transfer/burn/mint of YUSD tokens, so that even if a single core contract is maliciously upgraded all of the tokens could be drained. More than that, because of the project’s architecture, where any sub-project of different collateral token still involves the YUSD token, and because of the project’s nature that is based on the price stability of YUSD, if any of the sub-project gets attacked by an insider then all of the rest will be highly affected, in the sense that upon an attack the YUSD token will most probably decrease tremendously in value. Consequently, the borrowers would be practically neither able to repay their loans, nor redeem their collateral tokens.

Recommendation

In order to mitigate the centralization issues we suggest implementing a timelock functionality for any upgrades performed by the Owner of the contracts, so that the users have time to react if needed.

Informational/Suggestions

INFO-1	Code simplification
Contract(s)	`ActivePool, NestManagerBase`
Status	**Resolved**

Description

In `ActivePool::processCollateralTokenFee` the fee amounts `rewardTokenStakingFeeRewards` and kakeiboFee are computed as follows:

```
function processCollateralTokenFee(uint collateralTokenFee) external override
returns (uint rewardTokenStakingFeeReward) {

       rewardTokenStakingFeeReward = collateralTokenFee *
loansStableSettingsCache.rewardTokenStakingFeeRate() / 1e18;
       _sendCollateralToken(rewardTokenStakingAddress, rewardTokenStakingFeeReward,
true);

       _sendCollateralToken(kakeiboAddress, collateralTokenFee *
loansStableSettingsCache.kakeiboFeeRate() / 1e18, true);
}
```

However, these two fee amounts are complementary in respect to the `collateralTokenFee`, so there is no need to essentially perform the same calculation twice.

Also in NestManagerBase::_removeNestOwner the definition of local variable length can be omitted:

```
uint length = NestOwnersArrayLength;
uint idxLast = length - 1
```

and simply have:

```
uint idxLast = NestOwnersArrayLength - 1;
```

Recommendation

We suggest calculating `kakeiboFeeRate` by simply subtracting `rewardTokenStakingFeeReward` out of `collateraTokenFee` for simplicity, accuracy, readability and gas savings:

```
_sendCollateralToken(kakeiboAddress, collateralTokenFee -
rewardTokenStakingFeeReward, true);
```

We also suggest removing the redundant local variable length in `NestManagerBase::_removeNestOwner`.

INFO-2	Move struct/enum/event declarations to storage contract
Contract(s)	`BorrowerOperations, BorrowerOperationsStorage`
Status	**Resolved**

Description

In `BorrowerOperations.sol` several structs, an enum and an event are declared:

```
struct LocalVariables_adjustNest {...}

struct LocalVariables_openNest {...}

struct ContractsCache {...}

enum BorrowerOperation {...}

event NestUpdated(...);
```

However, there is a separate contract (`BorrowerOperationsStorage.sol`) dedicated to the state of BorrowerOperations and these should better be declared therein.

Recommendation

We suggest moving these declarations to `BorrowerOperationsStorage.sol` contract, for clarity but also for consistency to the architecture of the other contracts in the project.

INFO-3	Remove legacy commented out code
Contract(s)	`BorrowerOperations`
Status	**Resolved**

Description

In BorrowerOperations::openNest and BorrowerOperations::addColl some code of legacy functionality regarding the handling of nest payments in wrapped native assets remains commented out.

```
function openNest(uint256 amount, uint256 _maxFeePercentage, uint256 _YUSDAmount,
address _upperHint, address _lowerHint) external override {
    //WNAT.transferFrom(msg.sender, address(this), amount);
    _openNest(amount, _maxFeePercentage, _YUSDAmount, _upperHint, _lowerHint, true);
}

function addColl(uint256 amount, address _upperHint, address _lowerHint) external
override {
    //WNAT.transferFrom(msg.sender, address(this), amount);
    _adjustNest(amount, msg.sender, 0, 0, false, _upperHint, _lowerHint, 0, true);
}
```

Recommendation

We suggest deleting these commented out lines of code for clarity and readability.

INFO-4	Remove ‘TODO’ comments
Contract(s)	`BorrowerOperationsStorage, CollSurplusPoolStorage,ActivePoolStorage, StabilityPoolRewardTokenManager, StabilityPool, StabilityPoolRewardTokenManagerUpgraded,`
Status	**Largely Resolved**

Description

In BorrowerOperationsStorage.sol, CollSurplusPoolStorage.sol and ActivePoolStorage.sol there is a forgotten ‘TODO’ comment:

```
bool immutable public isTokenCollateral; //TODO:  immutable
```

There are a number of ‘TODO’ comments in several other contracts. Some of them are legacy comments, while others are actual pending items.
For example, several such comments exist in StabilityPoolRewardTokenManager.sol:

```
// TODO remove withdraw
// TODO remove balanceOf  // CP: for gas optimization, pending
// TODO remove stake
// TODO make update reward function
// TODO rename getRewardToClaim // CP: legacy
```

Recommendation

We suggest cleaning up all the ‘TODO’ items.

INFO-5	Misleading contract and functions naming
Contract(s)	BorrowerOperations, BorrowerOperationsWNat
Status	**Resolved**

Description

In BorrowerOperations.sol, BorrowerOperationsWNat.sol some functions seem to serve the same purpose but give to the user the choice to pay collateral either in native token or in its wrapped form. For example, the operation to open a new nest:

```
// CP: BorrowerOperationsWNat.sol
function openNestWNat(uint256 _maxFeePercentage, uint256 _YUSDAmount, address
_upperHint, address _lowerHint) external payable override {
    _openNest(msg.value, _maxFeePercentage, _YUSDAmount, _upperHint, _lowerHint,
false);
}

// CP: BorrowerOperations.sol
function openNest(uint256 amount, uint256 _maxFeePercentage, uint256 _YUSDAmount,
address _upperHint, address _lowerHint) external override {
    _openNest(amount, _maxFeePercentage, _YUSDAmount, _upperHint, _lowerHint, true);
}

function _openNest(uint256 amount, uint256 _maxFeePercentage, uint256 _YUSDAmount,
address _upperHint, address _lowerHint, bool isWnatNest) internal {...}
```

There seems to be some contradiction in the given names of these two 'openNest' functions: while the openNestWNat would be expected to be used in the case that the user pays in the wrapped version of the native token, the function is payable and isWnatNest parameter is set to false. At the same time openNest seems to handle the wrapped token payment method (and, in fact, any other ERC20 token that may be set as collateral). By extension, the contract name BorrowerOperationsWNat seems to be misleading.

Recommendation

We suggest reconsidering the BorrowerOperationsWNat and its functions naming in respect to the above observations.

Alleviation

The FLR Finance team renamed BorrowerOperationsWNat.sol to BorrowerOperationsNativeToken.sol. The function names of this contract were similarly altered, for example openNestWNat() has been renamed to openNestNativeToken().

INFO-6	Remove redundant lines of code
Contract(s)	`BorrowerOperations, ActivePoolProxy, CollSurplusPool, RewardTokenStaking`
Status	**Resolved**

Description

In BorrowerOperations.sol there are a few unnecessary code lines:

```
function _openNest(uint256 amount, uint256 _maxFeePercentage, uint256 _YUSDAmount,
address _upperHint, address _lowerHint, bool isWnatNest) internal {
  LocalVariables_openNest memory vars;
// ...
// CP: unnecessary line
  vars.YUSDFee;
}

function _getNewNestAmounts(...) internal pure returns (uint256, uint256) {

// CP: the following two assignments are essentially dead code - Look at INFO-7
  uint256 newColl = _coll;
  uint256 newDebt = _debt;

  newColl = _isCollIncrease ? _coll + _collChange : _coll - _collChange;
  newDebt = _isDebtIncrease ? _debt + _debtChange : _debt - _debtChange;

  return (newColl, newDebt);

}
```

In ActivePoolProxy.sol the import of Ownable.sol is redundant as it is already imported in ActivePoolStorage.sol. The same applies in CollSurplusPool.sol, StabilityPool.sol and RewardTokenStaking.sol.

```
import "../../utils/Ownable.sol"; // CP: redundant import
contract ActivePoolProxy is ActivePoolStorage {}
```

Recommendation

We suggest deleting unnecessary assignments and imports.

INFO-7	Return variables for code simplification
Contract(s)	`BorrowerOperations, NestManagerMain, NestManagerLiquidations, RewardTokenStaking`
Status	**Resolved**

Description

There is some inconsistency in the way that function return values are handled throughout the codebase. In some functions the return variable is named while in others not. Also, there are many cases where named return variables are not used optimally in regards to simplifying the code.

Here are some cases that we believe that named return variables could help simplify the code:

```
/// BorrowerOperations.sol
function _getNewICRFromNestChange(...) internal pure returns (uint256) {

//...

// CP: by naming the return variables the following two lines can be reduced to:
// CP: newICR = FlareLoansStableMath._computeCR(newColl, newDebt, _price);
  uint256 newICR = FlareLoansStableMath._computeCR(newColl, newDebt, _price);
  return newICR;
}

function _getNewNestAmounts(...) internal pure returns (uint256, uint256) {

// CP: the following two lines could be omitted by naming the return variables
  uint256 newColl = _coll;
  uint256 newDebt = _debt;

  newColl = _isCollIncrease ? _coll + _collChange : _coll - _collChange;
  newDebt = _isDebtIncrease ? _debt + _debtChange : _debt - _debtChange;
  return (newColl, newDebt); //CP: also could be omitted
}
```

```
/// NestManagerMain.sol
function _updateStakeAndTotalStakes(address _borrower) internal returns (uint) {
    // CP: uint declaration would be unnecessary if return variable is named
    uint newStake = _computeNewStake(nests[_borrower].coll);
    uint oldStake = nests[_borrower].stake;
    nests[_borrower].stake = newStake;

    totalStakes = totalStakes - oldStake + newStake;
    emit TotalStakesUpdated(totalStakes);
    return newStake; // CP: could be omitted
}

// CP: code can be simpler if return variable is named after 'stake'
function _computeNewStake(uint _coll) internal view returns (uint) {
    uint stake; // CP: could be omitted
    if (totalCollateralSnapshot == 0) {
        stake = _coll;
    } else {
        assert(totalStakesSnapshot > 0);
        stake = _coll * totalStakesSnapshot / totalCollateralSnapshot;
    }
    return stake; // CP: could be omitted
}

function addNestOwnerToArray(address _borrower) external override returns (uint
index) {
    _requireCallerIsBorrowerOperations();
    // CP: Assign to, currently unused, 'index' and omit the return statement
    return _addNestOwnerToArray(_borrower);
}

function _addNestOwnerToArray(address _borrower) internal returns (uint128 index) {
       //...
       return index; // CP: can be omitted
}

function _redeemCollateralFromNest(...) internal returns (SingleRedemptionValues
memory singleRedemption) {
    // ...
    return singleRedemption; // CP: can be omitted
```

```
/// NestManagerLiquidations.sol
function _liquidateRecoveryMode(...)
    internal
    returns (LiquidationValues memory singleLiquidation)
{
    // ...
    return singleLiquidation; // CP: can be omitted
}



/// RewardTokenStaking.sol
function _getPendingCollateralTokenGain(address _user) internal view returns
(uint256) {
    // ...
    uint256 collateralTokenGain = (stakes[_user] * (F_collateralToken -
F_collateralTokenSnapshot)) /DECIMAL_PRECISION; // CP: could omit type declaration
    return collateralTokenGain; // CP: could be omitted
}

function _getPendingYUSDGain(address _user) internal view returns (uint256) {
    // ...
    uint256 yusdGain = (stakes[_user] * (F_yusd - F_yusdSnapshot)) /
DECIMAL_PRECISION; // CP: type declaration of yusdGain could be omitted
    return yusdGain; // CP: could be omitted
}
```

Recommendation

We suggest simplifying the code as described.

INFO-8	Function modifiers for access control
Contract(s)	`ActivePool, BorrowerOperations, CollSurplusPool, DefaultPool, NestManagerMain, RewardTokenStaking, SortedNests, StabilityPool, YSDToken`
Status	**Resolved**

Description

There are a great number of functions that perform sensitive operations and are permitted to be accessed only by specific contracts. Access control is applied by require() statements within the body of those functions. Take for example ActivePool::processCollateralTokenFee:

```
function   processCollateralTokenFee(uint   collateralTokenFee)   external   override
returns (uint rewardTokenStakingFeeReward) {
    // CP: turn to a modifier
    _requireCallerIsNestManager();
    // …
}

function _requireCallerIsNestManager() internal view {
require(
      msg.sender == nestManagerAddress,
      "ActivePool: Caller is not NestManager");
}
```

Here is the list with the access control functions:

```
// ActivePool.sol
function _requireCallerIsBorrowerOperationsOrDefaultPool() internal view {}

function _requireCallerIsBOorNestMorSP() internal view {}

function _requireCallerIsNestManager() internal view {}

function _requireCallerIsBOorNestM() internal view {}
```

```
// BorrowerOperations.sol
function _requireCallerIsStabilityPool() internal view {}


// CollSurplusPool.sol
function _requireCallerIsBorrowerOperations() internal view {}
function _requireCallerIsNestManager() internal view {}
function _requireCallerIsActivePool() internal view {}


// DefaultPool.sol
function _requireCallerIsActivePool() internal view {}
function _requireCallerIsNestManager() internal view {}


// NestManagerMain.sol
function _requireCallerIsBorrowerOperations() internal view {}


// RewardTokenStaking.sol
function _requireCallerIsNestManager() internal view {}
function _requireCallerIsBorrowerOperations() internal view {}
function _requireCallerIsActivePool() internal view {}
// SortedNests.sol
function _requireCallerIsNestManager() internal view {}
function _requireCallerIsBOorNestM(INestManager _nestManager) internal view {}
// StabilityPool.sol
function _requireCallerIsActivePool() internal view {}
function _requireCallerIsNestManager() internal view {}

// YSDToken.sol
function _requireCallerIsBorrowerOperations() internal view {}
function _requireCallerIsBOorNestMorSP() internal view {}
function _requireCallerIsStabilityPool() internal view {}

function _requireCallerIsNestMorSP() internal view {}
```

However it is a best practice to use function modifiers for access control checks, since it makes the code much more readable.

Recommendation

We suggest turning all these functions to modifiers for clarity and readability.

INFO-9	Asymmetric implementation of similar functionality
Contract(s)	`BorrowerOperations`, `StabilityPool`, `StabilityPoolStorage`
Status	**Largely Resolved - Partially Dismissed**

Description

A. In BorrowerOperations contract, whereas for closing a nest the inner nest-related arrangements are delegated to the NestManagerMain contract:

```
function _closeNest(bool isWNatTransfer) internal {
    // ...
    nestManagerCached.removeStake(msg.sender);
    nestManagerCached.closeNest(msg.sender);
    emit NestUpdated(msg.sender, 0, 0, 0, BorrowerOperation.closeNest)
    // ...
}
```

for the case of opening a nest, all the relative arrangements are taken care of inside BorrowerOperations::_openNest by calling the required NestManagerMain's operations one-by-one:

```
function  _openNest(uint256  amount,  uint256  _maxFeePercentage,  uint256  _YUSDAmount,
address _upperHint, address _lowerHint, bool isWnatNest) internal {
    // ...
    contractsCache.nestManager.setNestStatus(msg.sender, 1);
    contractsCache.nestManager.increaseNestColl(msg.sender, amount);
    contractsCache.nestManager.increaseNestDebt(msg.sender, vars.compositeDebt);
    contractsCache.nestManager.updateNestRewardSnapshots(msg.sender);
    vars.stake = contractsCache.nestManager.updateStakeAndTotalStakes(msg.sender);
    sortedNests.insert(msg.sender, vars.NICR, _upperHint, _lowerHint);
    vars.arrayIndex = contractsCache.nestManager.addNestOwnerToArray(msg.sender);
    emit NestCreated(msg.sender, vars.arrayIndex)
    // ...
}
```

We don't see a reason justifying such an asymmetry. On the contrary, it could be error-prone regarding future upgrades of the contracts while it also negatively affects the readability of BorrowerOperations::_openNest, especially when it comes to hard-coding a enum field as in `contractsCache.nestManager.setNestStatus(msg.sender, 1);`

B. In StabilityPool.sol all three functions provideToSP(), withdrawFromSP() and withdrawCollateralTokenGain() claim rewards for msg.sender. However, the implementation is slightly different in each one of them:

```
/// SP::depositToSP()
stabilityPoolRewardTokenManager.claimReward(msg.sender, frontEnd, kickbackRate);

/// SP::withdrawFromSP
stabilityPoolRewardTokenManager.claimReward(msg.sender, frontEnd, kickbackRate);
if (rewardUser > 0) depositorRewardTokenPaid[msg.sender] += rewardUser;
if (rewardFrontend > 0) frontRewardTokenPaid[frontEnd] += rewardFrontend;

/// SP::withdrawCollateralTokenGain
(rewardUser, rewardFrontend) = safeReward ?
stabilityPoolRewardTokenManager.claimRewardSafe(msg.sender, frontEnd, kickbackRate)
: stabilityPoolRewardTokenManager.claimReward(msg.sender, frontEnd, kickbackRate)
```

More specifically, in the second case the following mappings get updated:

```
mapping (address => uint) public frontRewardTokenPaid;
mapping (address => uint) public depositorRewardTokenPaid
```

However, these mapping are nowhere else used within the whole FlareLoans codebase.
In addition, in depositToSP() claiming the rewards by calling claimRewardsSafe() is not a choice.

Recommendation

A. We suggest creating an openNest function in NestManagerMain contract for better code organization and responsibilities distinction among contracts, but also for readability.
B. We suggest removing unused mappings `frontRewardTokenPaid` and `depositorRewardTokenPaid`.

INFO-10	Dead code
Contract(s)	`BorrowerOperations, DefaultPool, CollSurplusPool, RewardTokenStaking, StabilityPool`
Status	**Largely Resolved - Partially Dismissed**

Description

A. In BorrowerOperations.sol function _requireCallerIsBorrower():

```
function _requireCallerIsBorrower(address _borrower) internal view {
    require(msg.sender == _borrower, "BorrowerOps: Caller must be the borrower for a
withdrawal");

}
```

seems to be legacy code since it is nowhere called from.

B. DefaultPool::convertToWNatTokenSurplus seems to be dead code, since no other function than DefaultPool::receive is payable and receive() already converts any incoming amount to WNat. The same applies also for CollSurplusPool::convertToWNatNativeTokenSurplus and RewardTokenStaking:convertToWNatNativeTokenSurplus.

```
// DefaultPool.sol
function convertToWNatNativeTokenSurplus() external onlyOwner {
    uint balance = address(this).balance;
    require(balance > 0, "DefaultPool: No native token surplus");
    WNAT.deposit{value: balance}();
}
```

C. In StabilityPool.sol there is an onlyOwner approve() function that seems to serve no purpose:

```
function approve(address spender, uint amount) public onlyOwner returns (bool) {
    return collateralToken.approve(spender, amount);
}
```

There is also a forgotten line of code in StabilityPool::provideToSP:

```
function provideToSP(uint _amount, address _frontEndTag, bool isWNatTransfer)
external override {
    stabilityPoolRewardTokenManager.claimReward(msg.sender, frontEnd, kickbackRate);
    (msg.sender);   // CP: dead code
}
```

D. In StabilityPool.sol both functions provideToSP() and withdrawFromSP() first call `StabilityPoolRewardTokenManager::updateRewardAmount()` and afterwards claim the rewards for the user:

```
function provideToSP(uint _amount, address _frontEndTag, bool isWNatTransfer)
external override {
    // ...
    stabilityPoolRewardTokenManager.updateRewardAmount(msg.sender);
    // ...
    stabilityPoolRewardTokenManager.claimReward(msg.sender, frontEnd, kickbackRate);
}

function withdrawFromSP(uint _amount, bool safeReward, bool isWNatTransfer) external
override {
    stabilityPoolRewardTokenManager.updateRewardAmount(msg.sender);
    // ...
    (uint rewardUser, uint rewardFrontend) = safeReward ?
     stabilityPoolRewardTokenManager.claimRewardSafe(msg.sender, frontEnd,
kickbackRate):
     stabilityPoolRewardTokenManager.claimReward(msg.sender, frontEnd,
kickbackRate);
}
```

However, updateRewardAmount() only executes the code of modifier StabilityPoolRewardTokenManager::updateReward, which is also a modifier of the claim-rewards functions. This makes the calls to updateRewardAmount() redundant.

Recommendation

A, B. We suggest removing these functions.

C. We suggest removing this line of code

D. We suggest omitting the call to `StabilityPoolRewardTokenManager::updateRewardAmount()` from both `SP::provideToSP` and `SP::withdrawFromSP`.

INFO-11	Code duplication and error-prone enum handling
Contract(s)	`BorrowerOperations, NestManagerLiquidations`
Status	**Resolved**

Description

A. Some operations in BorrowerOperations contract require that the nest of interest is active or inactive. This is handled with a call to the following internal functions:

```
// BorrowerOperations.sol
function _requireNestIsActive(INestManager _nestManager, address _borrower) internal
view {
    uint256 status = _nestManager.getNestStatus(_borrower);
    // CP: 1 corresponds to Status enum field "active"
    require(status == 1, "BorrowerOps: Nest does not exist or is closed");
}

function _requireNestisNotActive(INestManager _nestManager, address _borrower)
internal view {
    uint256 status = _nestManager.getNestStatus(_borrower);
    require(status != 1, "BorrowerOps: Nest is active");
}
```

In these functions an error-prone handling of the enum "Status" takes place, in order to confirm that the nest's status is active or not:

```
// NestManagerStorage.sol
enum Status {
    nonExistent,
    active,
    closedByOwner,
    closedByLiquidation,
    closedByRedemption
}
```

At the same time, contract NestManagerBase already provides such a functionality wrapped in a function:

```
// NestManagerBase.sol
function _requireNestIsActive(address _borrower) internal view {
     require(nests[_borrower].status  ==  Status.active,  "NestManager:  Nest  does  not
exist or is closed");
}
```

Similarly in StabilityPool.sol:

```
// StabilityPool.sol
function _requireUserHasNest(address _depositor) internal view {
    // CP: use nestManager._requireNestIsActive(_depositor) instead
   require(nestManager.getNestStatus(_depositor) == 1, "StabilityPool: Caller must
have an active nest to withdraw CollateralTokenGain to");
```

B. There is a significant amount of code duplication in NestManagerLiquidations.sol. More specifically, functions liquidateNests and batchLiquidateNests differ only in a single function call:

```
function liquidateNests(uint _n, bool isWNatTransfer) external override {
    LiquidationTotals memory totals;
    // ...
    if (vars.recoveryModeAtStart) {
    totals = _getTotalsFromLiquidateNestsSequence_RecoveryMode(contractsCache,
vars.price, vars.YUSDInStabPool, _n);
    } else {
 totals = _getTotalsFromLiquidateNestsSequence_NormalMode(
contractsCache.activePool, contractsCache.defaultPool, vars.price,
vars.YUSDInStabPool, _n);
    // ...
}

function batchLiquidateNests(address[] memory _nestArray, bool isWNatTransfer)
public override {
    LiquidationTotals memory totals;
    // ...
    if (vars.recoveryModeAtStart) {
 totals = _getTotalFromBatchLiquidate_RecoveryMode(activePoolCached,
defaultPoolCached, vars.price, vars.YUSDInStabPool, _nestArray);
    } else {
 totals = _getTotalsFromBatchLiquidate_NormalMode(activePoolCached,
defaultPoolCached, vars.price, vars.YUSDInStabPool, _nestArray);
    // ...
}
```

Recommendation

We suggest using NestManager::_requireNestIsActive so as to avoid the hardcoding of an enum's field but also avoid code duplication. To avoid code duplication and decrease NestManagerLiquidations.sol size, consider having only one function accepting both parameters `address[] nestArray, uint _n` and adopt a convention to tell one case from another e.g. based on the length of the `nestArray`.

INFO-12	Internal function could be private
Contract(s)	`NestManagerBase`
Status	**Resolved**

Description

Internal function NestManagerBase::_removeNestOwner removes a borrower from the nestOwners array and is called only by NestManagerBase::_closeNest function which makes sure that the borrower is also deleted from the sortedNests data structure.

```
function _removeNestOwner(address _borrower, uint NestOwnersArrayLength) internal {}
```

The internal scope of this function makes it accessible to derived contracts although it should never be called by any other function than NestManagerBase::_closeNest.

Recommendation

We suggest altering the scope of NestManagerBase::_removeNestOwner to private for clarity due to its sensitive functionality.

INFO-13	Inconsistent function naming
Contract(s)	`NestManagerLiquidations`
Status	**Resolved**

Description

There is a typo in one of the four functions of NestManagerLiquidations construct that calculate the liquidation-related values of the struct `LiquidationsTotals`:

```
function _getTotalsFromLiquidateNestsSequence_RecoveryMode ()
function _getTotalsFromLiquidateNestsSequence_NormalMode ()
function _getTotalsFromBatchLiquidate_NormalMode ()
function _getTotalFromBatchLiquidate_RecoveryMode () // CP: getTotal… -> getTotals…
```

Recommendation

We suggest renaming the mistyped function.

INFO-14	Redundant inheritance
Contract(s)	`NestManagerMain`
Status	**Resolved**

Description

Contract NestManagerMain declares inheritance from both NestManagerBase and NestManagerStorage but the latter is already inherited because of the first:

```
contract NestManagerMain is NestManagerStorage, NestManagerBase, INestManagerMain {}
abstract contract NestManagerBase is NestManagerStorage, INestManagerBase {}
```

Recommendation

We suggest removing redundant inheritance declaration of NestManagerStorage.

INFO-15	Misleading comments and function name
Contract(s)	`NestManagerMain, CollSurplusPool`
Status	**Resolved**

Description

Function NestManagerMain::_redeemCloseNest is called after a nest has been close()d, for accounting purposes. However, the function's name together with the comments that describe it imply that the nest closes while in the function or after its execution.

```
function _redeemCollateralFromNest(...) internal returns (...)
{
  // ...
  _closeNest(_borrower, Status.closedByRedemption);
  _redeemCloseNest(_contractsCache,_borrower,
loansStableSettings.YUSD_GAS_COMPENSATION(),newColl);
  // ...
}

/*
* Called when a full redemption occurs, and closes the nest.
* The redeemer swaps (debt - liquidation reserve) YUSD for (debt - liquidation
reserve) worth of collateral token, so the YUSD liquidation reserve left corresponds
to the remaining debt.
* In order to close the nest, the YUSD liquidation reserve is burned, and the
corresponding debt is removed from the active pool.
* The debt recorded on the nest's struct is zero'd elswhere, in _closeNest.
* Any surplus collateral token left in the nest, is sent to the Coll surplus pool,
and can be later claimed by the borrower.
*/

function _redeemCloseNest(ContractsCache memory _contractsCache, address _borrower,
uint _YUSD, uint _collateral) internal {
    _contractsCache.yusdToken.burn(gasPoolAddress, _YUSD);
    // Update Active Pool YUSD, and send collateral token to account
    _contractsCache.activePool.decreaseYUSDDebt(_YUSD);
    // send collateral from Active Pool to CollSurplus Pool
    _contractsCache.collSurplusPool.accountSurplus(_borrower,_collateral);

_contractsCache.activePool.sendCollateralToken(address(_contractsCache.collSurplusPo
ol), _collateral, true);

}
```

This may lead to misconceptions and negatively affect the maintainability of the codebase while it harms readability.

Another case of misleading comment exists in CollSurplusPool.sol as follows:

```
/// CollSurplusPool.sol
// CP: actually returns the collateral balance of CollSurplusPool.
/* Returns the collateral token state variable at ActivePool address. */
function getCollateral() external view override returns (uint) {
    return collateralBalance;
}
```

Recommendation

We suggest updating the comments and giving NestManagerMain::_redeemCloseNest a more accurate name, e.g. _redeemClosedNest(). Also updating the comment for CollSurplusPool::getCollateral().

INFO-16	Decimal precision constant variable defined but not used
Contract(s)	`ActivePool`, `BorrowerOperations`, `StabilityPoolRewardTokenManager`, `StabilityPoolRewardTokenManagerStorage`, `StabilityPoolRewardTokenManagerUpgraded`, `FlareLoansStableBaseStorage`
Status	**Largely Resolved - Partially Dismissed**

Description

Contract StableMath defines the constant variable DECIMAL_PRECISION that is to be used throughout the codebase:

```
contract BaseMath {
    uint constant public DECIMAL_PRECISION = 1e18;
}
```

However most of the contracts do not use it. Instead decimal precision is hard coded any time needed, much like a magic constant without a name. There is also a contract that declares the constant variable DECIMAL_PRECISION again.

The contracts that hard code the decimal precision value are:

`ActivePool`
`BorrowerOperations`
`StabilityPoolRewardTokenManager`
`StabilityPoolRewardTokenManagerUpgraded`
`FlareLoansStableBaseStorage`

`StabilityPoolRewardTokenManagerStorage` redefines it (however contracts `StabilityPoolRewardTokenManager`, `StabilityPoolRewardTokenManagerUpgraded` still do not use the variable)

Recommendation

We suggest making each contract that performs calculations with decimal precision to inherit BaseMath and use the constant variable wherever needed.

INFO-17	Gas optimization
Contract(s)	`StabilityPool, NestManagerMain, StabilityPoolRewardTokenManager`
Status	**Resolved**

Description

A. In StabilityPool::withdrawCollateralTokenGain function getDepositorCollateralTokenGain, which gets to read from storage a number of times, is unnecessarily called twice:

```
function withdrawCollateralTokenGain(address _upperHint, address _lowerHint, bool
moveGainToNest, bool isWNatTransfer) external override {
    // ...
    // CP: getDepositorCollateralTokenGain is essentially called twice
    // CP: could be just require(depositorCollateralTokenGain > 0)
    _requireUserHasCollateralTokenGain(msg.sender);
    uint depositorCollateralTokenGain = getDepositorCollateralTokenGain(msg.sender)

function _requireUserHasCollateralTokenGain(address _depositor) internal view {
    uint collateralTokenGain = getDepositorCollateralTokenGain(_depositor);
    require(collateralTokenGain > 0, "StabilityPool: Caller must have non-zero
collateral token gain");

}
```

B. In modifier StabilityPoolRewardTokenManager::updateReward the reward per FLR token is calculated and stored in a local variable. It is assigned to the rewardPerFlrStored state variable only if it is a non-zero amount:

```
modifier updateReward(address account) virtual {
    rewardPerTokenStored = rewardPerToken();
    uint rewardPerFlr_ = rewardPerFlr();
    // CP: rewardPerFlr_ will be 0 only if rewardPerFlrStored is zero
    if (rewardPerFlr_ > 0) rewardPerFlrStored = rewardPerFlr_;
}
```

However, rewardPerFlr() could return zero only if rewardPerFlrStored equals 0. So this extra if statement is redundant and the result of rewardPerFlr() could be directly assigned to rewardPerFlrStored.

C. In NestManagerMain::redeemCollateral a while loop parses the sortedNests data structure until the first borrower with the lowest ICR but non liquidatable (i.e. borrower.ICR > MCR) is found:

```
while (currentBorrower != address(0) && getCurrentICR(currentBorrower, totals.price)
< loansStableSettings.MCR()) {
    currentBorrower = contractsCache.sortedNests.getPrev(currentBorrower);
}
```

Recommendation

A. We suggest first calculating depositorCollateralTokenGain and then check that it equals a non-zero amount for saving in gas costs.

B. Remove the redundant if statement and directly assign `rewardPerFlrStored = rewardPerFlr();`

C. We suggest declaring a local variable for the MCR value so as not to read it repeatedly from storage in the loop.

INFO-18	Typo in state variable’s name
Contract(s)	`StabilityPoolStorage, StabilityPool`
Status	**Resolved**

Description

The state variable that keeps track of the contract’s collateral balance is misspelled:

```
uint256 internal callateralBalance;
```

Recommendation

We suggest changing the variable’s name to `collateralBalance`.

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.