# Roots Security Review

## Pashov Audit Group

Conducted by: 0x37, 0xbepresent, btk, Shaka

February 9th 2025 - February 21st 2025

# Contents

# 1. About Pashov Audit Group

Pashov Audit Group consists of multiple teams of some of the best smart contract security researchers in the space. Having a combined reported security vulnerabilities count of over 1000, the group strives to create the absolute very best audit journey possible - although 100% security can never be guaranteed, we do guarantee the best efforts of our experienced researchers for your blockchain protocol. Check our previous work here or reach out on Twitter @pashovkrum.

# 2. Disclaimer

A smart contract security review can never verify the complete absence of vulnerabilities. This is a time, resource and expertise bound effort where we try to find as many vulnerabilities as possible. We can not guarantee 100% security after the review or even if the review will find any problems with your smart contracts. Subsequent security reviews, bug bounty programs and on-chain monitoring are strongly recommended.

# 3. Introduction

A time-boxed security review of the **roots-fi/roots-core** repository was done by **Pashov Audit Group**, with a focus on the security aspects of the application's smart contracts implementation.

# 4. About Roots

Roots is a DeFi protocol that allows users to borrow the $MEAD stablecoin by using Berachain-native assets as collateral, requiring a minimum collateral ratio of 120% to prevent liquidation. Users can redeem their collateral by repaying their $MEAD debt and earn rewards through staking in the Stability Pool or participating in liquidation events.

# 5. Risk Classification

| Severity | Impact: High | Impact: Medium | Impact: Low |
|---|---|---|---|
| Likelihood: High | Critical | High | Medium |
| Likelihood: Medium | High | Medium | Low |
| Likelihood: Low | Medium | Low | Low |

## 5.1. Impact

- High - leads to a significant material loss of assets in the protocol or significantly harms a group of users.
- Medium - only a small amount of funds can be lost (such as leakage of value) or a core functionality of the protocol is affected.
- Low - can lead to any kind of unexpected behavior with some of the protocol's functionalities that's not so critical.

## 5.2. Likelihood

- High - attack path is possible with reasonable assumptions that mimic on-chain conditions, and the cost of the attack is relatively low compared to the amount of funds that can be stolen or lost.
- Medium - only a conditionally incentivized attack vector, but still relatively likely.
- Low - has too many or too unlikely assumptions or requires a significant stake by the attacker with little or no incentive.

## 5.3. Action required for severity levels

- Critical - Must fix as soon as possible (if already deployed)
- High - Must fix (before deployment if not already deployed)
- Medium - Should fix
- Low - Could fix

# 6. Security Assessment Summary

*review commit hash* - c5dfa1eb74bf368e231203d33d7099c682fbe712

*fixes review commit hash* - ce3f41d9d96211fdcd6fb4a68510fa8195c90637

## Scope

The following smart contracts were in scope of the audit:

- `Factory`
- `TroveManager`
- `StabilityPool`
- `BorrowerOperations`
- `Staker`

# 7. Executive Summary

Over the course of the security review, 0x37, 0xbepresent, btk, Shaka engaged with Roots to review Roots. In this period of time a total of **27** issues were uncovered.

## Protocol Summary

| **Protocol Name** | Roots |
|---|---|
| **Repository** | https://github.com/roots-fi/roots-core |
| **Date** | February 9th 2025 - February 21st 2025 |
| **Protocol Type** | CDP Stablecoin |

## Findings Count

| Severity | Amount |
|---|---|
| Critical | 1 |
| High | 6 |
| Medium | 7 |
| Low | 13 |
| **Total Findings** | **27** |

# Summary of Findings

| ID | Title | Severity | Status |
|---|---|---|---|
| [C-01] | Incorrect interface used for gauge contract in Staker | Critical | Resolved |
| [H-01] | Overwriting collateral breaks collateral gains logic | High | Resolved |
| [H-02] | Stale totalActiveDebt used in openTrove causing incorrect debt update | High | Resolved |
| [H-03] | Incorrect boost management leads to staking reward loss | High | Resolved |
| [H-04] | Borrowers fail to claim their surplus collateral | High | Resolved |
| [H-05] | Incorrect activateBoost interface is used | High | Resolved |
| [H-06] | BGT stake rewards are locked | High | Resolved |
| [M-01] | DoS on last user claiming collateral gains in StabilityPool | Medium | Resolved |
| [M-02] | Base rate may decay at a slower rate than expected | Medium | Resolved |
| [M-03] | Users can prevent absorbing bad debt by sandwiching liquidations | Medium | Acknowledged |
| [M-04] | SetRewardsFeeBps fails to update reward calculations | Medium | Resolved |
| [M-05] | Validator transition delay can lead to boosted token mismanagement | Medium | Resolved |
| [M-06] | Troves can be sorted incorrectly due to interest accrual | Medium | Acknowledged |

| | | | |
|---|---|---|---|
| [M-07] | TroveManager does not work with non-18 decimal tokens | Medium | Acknowledged |
| [L-01] | Staker.setRewardsFeeBps() does not check if the new fee is less than 100% | Low | Resolved |
| [L-02] | Inefficient activation of boosts if BGT's activateBoostDelay is changed | Low | Resolved |
| [L-03] | Users might receive less collateral due to rounding down | Low | Resolved |
| [L-04] | Amount of collaterals enabled in StabilityPool is not limited | Low | Acknowledged |
| [L-05] | Loss evasion by stability depositors | Low | Acknowledged |
| [L-06] | Collateral redemptions might be prevented unnecessarily | Low | Acknowledged |
| [L-07] | StabilityPool and BorrowerOperations can be reinitialized | Low | Resolved |
| [L-08] | The Staker and TroveManager cannot operate with non-conforming ERC20 tokens | Low | Acknowledged |
| [L-09] | Fee on transfer tokens are not supported | Low | Acknowledged |
| [L-10] | Missing events in Staker functions | Low | Acknowledged |
| [L-11] | boostThreshold check is missing in setValidator | Low | Resolved |
| [L-12] | Open trove or add collateral may be blocked | Low | Resolved |
| [L-13] | Boost activation delay vulnerability in Staker due to setValidator abuse | Low | Resolved |

# 8. Findings

## 8.1. Critical Findings

### [C-01] Incorrect interface used for gauge contract in `Staker`

#### Severity

**Impact:** High

**Likelihood:** High

#### Description

The `Staker` contract is meant to interact with the `RewardVault` contract. However, the `IGauge` interface does not match the interface of the `RewardVault` contact for the `getReward` function.

```
File: IRewardVault.sol
function getReward(address account, address recipient) external returns
  (uint256);
```

```
File: IGauge.sol
function getReward(address) external returns (uint256);
```

This will provoke the `_updateRewardIntegral()` function to revert when `_fetchRewards()` is executed.

```
File: Staker.sol
    function _fetchRewards() internal {
        gauge.getReward(address(this));
```

The `_updateRewardIntegral()` function is executed on the most important operations for the protocol, such as opening, closing and adjusting troves, collateral redemption or liquidation, breaking the protocol's functionality and leading to funds being locked.

# Recommendations

```
function _fetchRewards() internal {
-       gauge.getReward(address(this));
+       gauge.getReward(address(this), address(this));
```

# 8.2. High Findings

## [H-01] Overwriting collateral breaks collateral gains logic

### Severity

**Impact:** High

**Likelihood:** Medium

### Description

In the `StabilityPool` contract, when 180 days have passed since the sunset of a collateral, enabling a new collateral overwrites the previous one.

In this process, the `epochToScaleToSums` mapping is cleared, but not the `collateralGainsByDepositor` and `depositSums` mappings. This means that the pending gains for the old collateral will be counted as gains for the new collateral.

As a result, the collateral gains logic is broken, leading to different situations, such as:

- DoS on functions that execute `_accrueDepositorCollateralGain` due to underflow on `uint256 firstPortion = sums[i] - depSums[i];`.
- Users being unable to claim their collateral gains because there is not enough collateral in the pool.
- Users claiming more gains than they should, taking them from other users (see proof of concept).

Additionally, the current overwriting system requires that a new collateral is enabled in order to disable the sunset collateral, which is not ideal.

### Proof of concept

Add the following code to the file `test/unit/Roots.t.sol` and run `forge test --mt test_sunsetCollateral`.

```
function _openTroveAndProvideToSP(
    TroveManager troveManager,
    address user,
    uint256 collAmount,
    uint256 debtAmount
) private {
    Collateral collateral = Collateral(address(troveManager.collateralToken()));
    vm.startPrank(user);
    collateral.mint(collAmount);
    collateral.approve(address(system.borrowerOperations), collAmount);
    system.borrowerOperations.openTrove(
      troveManager,
      user,
      1e16,
      collAmount,
      debtAmount,
      address
    ), address(0
    system.debtToken.approve(address(system.stabilityPool), debtAmount);
    system.stabilityPool.provideToSP(debtAmount);
    vm.stopPrank();
}

function test_sunsetCollateral() external {
    uint256 collateralToProvide = 4e18;
    uint256 amountToBorrow = 1e18;

    // Alice opens trove with coll A and provides MEAR to stability pool
    _openTroveAndProvideToSP
      (troveManagerA, ALICE, collateralToProvide * 100, amountToBorrow);

    // Bob opens trove with coll A and provides MEAR to stability pool
    _openTroveAndProvideToSP
      (troveManagerA, BOB, collateralToProvide, amountToBorrow);

    // Bob becomes liquidable and is liquidated
    uint256 newCollateralPrice = (12e17 * amountToBorrow) / collateralToProvide;
    priceFeed.setPrice(address(collateralA), newCollateralPrice);
    vm.prank(LIQUIDATOR);
    system.liquidationManager.liquidate(troveManagerA, BOB);

    // Alice triggers accrual of collateral gains
    skip(1);
    vm.prank(ALICE);
    system.stabilityPool.withdrawFromSP(0);

    // Collateral A is sunset
    vm.startPrank(OWNER);
    troveManagerA.startSunset();
    system.stabilityPool.startCollateralSunset(collateralA);
    vm.stopPrank();

    // After 180 days, a new collateral is added
    skip(180 days + 1);
    Collateral collateralC = new Collateral("Collateral C", "CC");
    priceFeed.setPrice(address(collateralC), 2e18);

    vm.prank(OWNER);
    system.factory.deployNewInstance(
        address(collateralC),
        address(priceFeed),
        address(0),
        address(0),
        address(0),
        abi.encodeWithSelector(
            TroveManager.setParameters.selector,
            minuteDecayFactor,
```

```
                redemptionFeeFloor,
                maxRedemptionFee,
                borrowingFeeFloor,
                maxBorrowingFee,
                interestRateInBps,
                maxDebt,
                MCR
            ),
            bytes("")
        );
        TroveManager troveManagerC = TroveManager(system.factory.troveManagers(2));

        // Alice opens trove with coll C and provides MEAR to stability pool
        _openTroveAndProvideToSP
          (troveManagerC, ALICE, collateralToProvide * 100, amountToBorrow);

        // Bob opens trove with coll C and provides MEAR to stability pool
        _openTroveAndProvideToSP
          (troveManagerC, BOB, collateralToProvide, amountToBorrow);

        // Bob becomes liquidable and is liquidated
        priceFeed.setPrice(address(collateralC), newCollateralPrice);
        vm.prank(LIQUIDATOR);
        system.liquidationManager.liquidate(troveManagerC, BOB);

        // Alice claims gains for coll C, receiving also the amount corresponding to
        // coll A
        uint256[] memory collateralIndexes = new uint256[](1);
        collateralIndexes[0] = 0;
        vm.prank(ALICE);
        system.stabilityPool.claimCollateralGains(ALICE, collateralIndexes);

        // Bob tries to claim gains for coll C, but his gains have been sent to
        // Alice
        vm.prank(BOB);
        vm.expectRevert("ERC20: transfer amount exceeds balance");
        system.stabilityPool.claimCollateralGains(BOB, collateralIndexes);
    }
```

## Recommendations

Avoid overwriting sunset collaterals and create a new function to disable them after the `expiry` time has been reached.

# [H-02] Stale `totalActiveDebt` used in `openTrove` causing incorrect debt update

## Severity

**Impact:** Medium

**Likelihood:** High

## Description

In the `TroveManager::openTrove` function, the `totalActiveDebt` is cached into a local variable `supply` before the `_accrueActiveInterests` function is called. The `_accrueActiveInterests` function itself updates the `totalActiveDebt` to reflect accrued interest. However, the `openTrove` function subsequently updates `totalActiveDebt` using the cached `supply` value instead of the potentially updated `totalActiveDebt`.

Specifically, the code caches `totalActiveDebt` in line 754:

```
File: TroveManager.sol
754:     uint256 supply = totalActiveDebt;
```

Then, interest is accrued, potentially updating `totalActiveDebt` in line 761 by calling `_accrueActiveInterests()`:

```
File: TroveManager.sol
761:     uint256 currentInterestIndex = _accrueActiveInterests();
```

Inside `_accrueActiveInterests`, `totalActiveDebt` is updated:

```
File: TroveManager.sol
1183:            totalActiveDebt = currentDebt + activeInterests;
```

Finally, in line 776, `totalActiveDebt` is updated using the *cached* `supply` value:

```
File: TroveManager.sol
774:@>        uint256 _newTotalDebt = supply + _compositeDebt;
775:          require(
   _newTotalDebt+defaultedDebt<=maxSystemDebt,
   "Collateraldebtlimitreached"
);
776:@>        totalActiveDebt = _newTotalDebt;
```

Because `supply` is the value of `totalActiveDebt` before interest accrual in `_accrueActiveInterests`, the update in line 776 does not incorporate the interest accrued within the same `openTrove` transaction. This means the `totalActiveDebt` may be incorrectly updated, potentially underestimating the actual debt in the system.

Consider the next scenario:

1. Initial state:

- `totalActiveDebt = 1000`
- Interest rate is such that `_accrueActiveInterests()` will increase `totalActiveDebt` by 100.

2. `openTrove` call:

- User calls `openTrove` with `_compositeDebt = 500`.
- Line 754: `supply = totalActiveDebt;` => `supply = 1000` (cached value).
- Line 761: `_accrueActiveInterests()` is called. Inside `_accrueActiveInterests()` (line 1183): `totalActiveDebt` is updated to `1000 + 100 = 1100`.
- Line 776: `totalActiveDebt = _newTotalDebt;` where `_newTotalDebt = supply + _compositeDebt = 1000 + 500 = 1500`.
- `totalActiveDebt` is set to `1500`.

3. Expected vs. actual `totalActiveDebt`:

- Expected `totalActiveDebt`: Initial debt (1000) + accrued interest (100) + new debt (500) = `1600`.
- Actual `totalActiveDebt` after `openTrove`: `1500`.

The `totalActiveDebt` is underestimated by `100` (the accrued interest within the same transaction).

## Recommendations

To correct this issue, ensure that the `totalActiveDebt` is updated with the most recent value *after* the interest accrual. Instead of caching `totalActiveDebt` at the beginning of the function.

# [H-03] Incorrect boost management leads to staking reward loss

## Severity

**Impact:** Medium

**Likelihood:** High

# Description

The `rewardCache::dropBoost` function (BGT 0x656b95E550C07a9ffe548bd4085c72418Ceb1dba) is designed to drop boosted rewards for a given validator. However, it checks the `dropBoostQueue` storage to confirm whether sufficient time has passed since the last boost. The problem arises because `rewardCache::queueDropBoost` is never called prior to `dropBoost` in functions like `Staker::_redeemRewards` and `Staker::setValidator`. Consequently, the `dropBoostQueue` remains empty, and `dropBoost` always returns `false`, preventing reward adjustments.

1. The `queueDropBoost` function is used to queue a future drop of boosted tokens:

```
function queueDropBoost(bytes calldata pubkey, uint128 amount) external {
@>      QueuedDropBoost storage qdb = dropBoostQueue[msg.sender][pubkey];
        uint128 dropBalance = qdb.balance + amount;
        // check if the user has enough boosted balance to drop
        if
          (boosted[msg.sender][pubkey] < dropBalance) NotEnoughBoostedBalance.selector
@>      (qdb.balance, qdb.blockNumberLast) = (dropBalance, uint32
  (block.number));
        emit QueueDropBoost(msg.sender, pubkey, amount);
    }
```

2. The `dropBoost` function attempts to drop boosts but requires a valid `dropBoostQueue` entry:

```
function dropBoost(address user, bytes calldata pubkey) external returns
      (bool) {
@>      QueuedDropBoost storage qdb = dropBoostQueue[user][pubkey];
@>      (uint32 blockNumberLast, uint128 amount) =
  (qdb.blockNumberLast, qdb.balance);
        // `amount` must be greater than zero to avoid reverting as
        // `withdraw` will fail with zero amount.
@>      if (amount == 0 || !_checkEnoughTimePassed
  (blockNumberLast, dropBoostDelay)) return false;
..
```

3. `_redeemRewards` directly calls `dropBoost` without `queueDropBoost`:

```
File: Staker.sol
305:            if (toFulfill > 0) {
306:                rewardCache.dropBoost(validator, uint128(toFulfill));
307:            }
```

4. Similarly, `setValidator` calls `dropBoost` without using `queueDropBoost`:

```
File: Staker.sol
80:         uint256 boosted = rewardCache.boosts(address(this));
81:         if (boosted > 0) {
82:             rewardCache.dropBoost(oldValidator, uint128(boosted));
83:         }
```

## Recommendations

Update the logic in `Staker::_redeemRewards` and `Staker::setValidator` to ensure that `rewardCache::queueDropBoost` is properly called before invoking `dropBoost`.

# [H-04] Borrowers fail to claim their surplus collateral

## Severity

**Impact:** High

**Likelihood:** Medium

## Description

In roots, if the TCR is less than CCR, the system will enter the recovery mode. In the recovery mode, some borrow positions will be liquidated even if their borrow positions' ICR is larger than MCR.

In this kind of liquidation, we will not seize all collateral token from the borrower. We will calculate the collateral token according to the MCR. It means that there are some left collateral token for the borrowers, named `collSurplus`. We will record the liquidated borrower's `surplusBalances` via function `addCollateralSurplus`.

Liquidated borrowers can claim the left collateral via function `claimCollateral`. The problem is that roots protocol is a little bit different with Prisma. All of our collateral tokens will be staked in the Staker contract to earn some rewards. But in function `claimCollateral`, we try to transfer collateral token from the TroveManager directly. This transaction will be reverted. If we want to withdraw some collateral from the Trove Manager, we should use `staker.onWithdrawal` to withdraw these collateral from the Staker contract.

```
function _tryLiquidateWithCap(
        ITroveManager troveManager,
        address _borrower,
        uint256 _debtInStabPool,
        uint256 _MCR,
        uint256 _price
    ) internal returns (LiquidationValues memory singleLiquidation) {
        uint256 collToOffset = (entireTroveDebt * _MCR) / _price;

        singleLiquidation.collGasCompensation = _getCollGasCompensation
          (collToOffset);
        singleLiquidation.debtGasCompensation = DEBT_GAS_COMPENSATION;

        singleLiquidation.debtToOffset = entireTroveDebt;

                 singleLiquidation.collToSendToSP = collToOffset - singleLiquidation.c

        troveManager.closeTroveByLiquidation(_borrower);

        uint256 collSurplus = entireTroveColl - collToOffset;
        if (collSurplus > 0) {
            singleLiquidation.collSurplus = collSurplus;
            troveManager.addCollateralSurplus(_borrower, collSurplus);
        }
```

```
function addCollateralSurplus
      (address borrower, uint256 collSurplus) external {
        _requireCallerIsLM();
        surplusBalances[borrower] += collSurplus;
    }
```

```
function claimCollateral(address _receiver) external {
        uint256 claimableColl = surplusBalances[msg.sender];
        require(claimableColl > 0, "No collateral available to claim");

        surplusBalances[msg.sender] = 0;

        collateralToken.safeTransfer(_receiver, claimableColl);
    }
```

# Recommendations

Withdraw the surplus collateral from the Staker contract when one borrow position is liquidated.

# [H-05] Incorrect `activateBoost` interface is used

# Severity

**Impact:** Medium

**Likelihood:** High

## Description

In Staker contract, we will activate our queued boost BGT. The interface we use is as below:

```
function activateBoost(address validator) external;
```

```
function _tryToBoost() internal {
        if (queued > 0 && blockDelta > 8191) {
            rewardCache.activateBoost(validator);
        }
    }
```

When we check the BGT's implementation, we will find out that the correct interface should be like as below:

```
function activateBoost
      (address user, bytes calldata pubkey) external returns (bool) {
        ...
    }
```

We use the incorrect interface, one `address user` parameter is needed. And this will cause that we cannot boost as expected. And this `_tryToBoost()` will be triggered by `_updateRewardIntegral`. And most functions in TroveManger will be impacted.

## Recommendations

Follow the BGT's implementation and trigger the correct `activateBoost()` interface.

# [H-06] BGT stake rewards are locked

## Severity

**Impact:** Medium

**Likelihood:** High

## Description

In Staker contract, we will stake our collateral token to get some BGT reward tokens. BGT reward tokens can be boosted. When we boost our BGT token in the staker contract, BGT contract will help us stake our BGT token into BGTStaker contract(https://berascan.com/address/0x44F07Ce5AfeCbCC406e6beFD40cc2998eEb8c7C6).

```
function _tryToBoost() internal {
            if (queued > 0 && blockDelta > 8191) {
            rewardCache.activateBoost(validator);
        }
```

```
function activateBoost
      (address user, bytes calldata pubkey) external returns (bool) {
        IBGTStaker(staker).stake(user, amount);
        return true;
    }
```

When we check BGTStaker's implementation, we stake BGT token into BGT Staker, we can get some HONEY rewards. We can get these rewards via interface `getReward`. But we miss one interface in Staker contract to get this part of rewards.

```
function getReward() external returns (uint256) {
        return _getReward(msg.sender, msg.sender);
    }
```

# Recommendations

Add one interface in Staker contract to claim this boost rewards.

# 8.3. Medium Findings

## [M-01] DoS on last user claiming collateral gains in `StabilityPool`

### Severity

**Impact:** High

**Likelihood:** Low

### Description

When a sunset collateral is overwritten in the `StabilityPool` contract, the value of `lastCollateralError_Offset` is not reset, which means that the new collateral will inherit the value of the previous one. This can provoke the value of `collateralGainPerUnitStaked` to be higher than it should be:

```
function _computeRewardsPerUnitStaked(
(...)
        uint256 collateralNumerator =
          (_collToAdd * DECIMAL_PRECISION) + lastCollateralError_Offset[idx];
(...)
                collateralGainPerUnitStaked = collateralNumerator / _totalDebtTokenDe
```

This value is stored in the `epochToScaleToSums` mapping and used later to calculate the collateral gains for depositors.

As a result, the last user to claim their collateral gains might not be able to do so, as there might not be enough collateral in the pool to cover the overvalued gains.

### Recommendations

Reset the `lastCollateralError_Offset` value for the index reassigned to the new collateral in the `_overwriteCollateral` function.

```
+       lastCollateralError_Offset[idx] = 0;
        indexByCollateral[_newCollateral] = idx + 1;
        emit CollateralOverwritten(collateralTokens[idx], _newCollateral);
        collateralTokens[idx] = _newCollateral;
```

# [M-02] Base rate may decay at a slower rate than expected

## Severity

**Impact:** Medium

**Likelihood:** Medium

## Description

The base rate is used to calculate fees for redemptions and new issuances of debt, and its value decays over time and increases based on redemption volume.

On redemption, the decay on the base rate is calculated based on the number of minutes passed since the last fee operation. What is important to note is that the number of minutes passed is rounded down to the nearest minute.

```
File: TroveManager.sol
    function _calcDecayedBaseRate() internal view returns (uint256) {
@>      uint256 minutesPassed =
  (block.timestamp - lastFeeOperationTime) / SECONDS_IN_ONE_MINUTE;
```

However, when `lastFeeOperationTime` is updated at the end of the operation, the value stored is the current block timestamp.

```
File: TroveManager.sol
    function _updateLastFeeOpTime() internal {
        uint256 timePassed = block.timestamp - lastFeeOperationTime;

        if (timePassed >= SECONDS_IN_ONE_MINUTE) {
@>          lastFeeOperationTime = block.timestamp;
```

This inconsistency can lead to the base rate decaying slower than expected.

For example, if 119 seconds have passed since the last fee operation, the number of minutes passed rounds down to 1. The base rate will decay as if only 60 seconds had passed, however, `lastFeeOperationTime` will be updated taking into account the full 119 seconds. This means that in the worst-case scenario the base rate will only decay for the amount corresponding to 60 seconds every 119 seconds.

## Recommendations

```
function _updateLastFeeOpTime() internal {
-        uint256 timePassed = block.timestamp - lastFeeOperationTime;
+        uint256 minutesPassed =
+ (block.timestamp - lastFeeOperationTime) / SECONDS_IN_ONE_MINUTE;

-        if (timePassed >= SECONDS_IN_ONE_MINUTE) {
+        if (minutesPassed > 0) {
-            lastFeeOperationTime = block.timestamp;
+            lastFeeOperationTime += minutesPassed * SECONDS_IN_ONE_MINUTE;
             emit LastFeeOpTimeUpdated(block.timestamp);
         }
    }
```

# [M-03] Users can prevent absorbing bad debt by sandwiching liquidations

## Severity

**Impact:** High

**Likelihood:** Low

## Description

When a liquidation occurs and the stability pool does not have enough funds to absorb the bad debt, this debt is distributed among all active troves.

Before a big liquidation that will cause a redistribution of bad debt, trove owners can withdraw their collateral just before the liquidation happens and open the trove again just after the liquidation, thus avoiding the redistribution of bad debt and increasing the bad debt of other troves.

## Recommendations

Implement a two-step mechanism for closing troves such that the request and the execution of the closing are separated by a time delay.

# [M-04] `SetRewardsFeeBps` fails to update reward calculations

## Severity

**Impact:** Medium

**Likelihood:** Medium

## Description

The `Staker::setRewardsFeeBps` function updates the fee percentage applied to rewards earned by the `Staker` contract. However, it does not call `_updateRewardIntegral` before modifying the fee. `_updateRewardIntegral` is responsible for calculating and integrating the rewards and fees into the system. Without invoking `_updateRewardIntegral`, all pending rewards are calculated using the new fee rate, potentially leading to inaccurate rewards distribution.

Code analysis:

1. The `setRewardsFeeBps` function is defined as follows:

```
File: Staker.sol
111:     function setRewardsFeeBps(uint16 _rewardsFeeBps) external onlyOwner {
112:         rewardsFeeBps = _rewardsFeeBps;
113:     }
```

2. The reward calculation occurs in the following section:

```
File: Staker.sol
185:                 uint256 earned = gauge.earned(address(this));
186:@>               uint256 fees = earned.mulDiv
  (rewardsFeeBps, 10000, Math.Rounding.Down);
187:                 integral += (earned - fees).mulDiv
  (1e18, supply, Math.Rounding.Down);
188:                 rewardIntegral = integral;
189:                 earnedFees += fees;
190:                 _fetchRewards();
```

The reward fees are applied using the `rewardsFeeBps` at the time of calculation. If `rewardsFeeBps` is updated without recalculating rewards, the pending rewards will be processed with the new fee rate, leading to inaccurate results. Consider the next scenario:

1. The protocol is configured with a `rewardsFeeBps` of 5%.
2. `Staker` have earned rewards, but the rewards have not yet been processed.
3. The owner calls `setRewardsFeeBps(10)` to increase the fee to 10%.
4. When `_updateRewardIntegral` is eventually called, the pending rewards are calculated using the new 10% fee instead of the original 5%, leading to incorrect reward and fee calculations.

## Recommendations

To ensure accurate reward distribution, the `setRewardsFeeBps` function should invoke `_updateRewardIntegral` before updating the `rewardsFeeBps`. This ensures that any pending rewards are processed using the old fee rate.

# [M-05] Validator transition delay can lead to boosted token mismanagement

## Severity

**Impact:** Medium

**Likelihood:** Medium

## Description

The `setValidator` function in `Staker.sol` contains a vulnerability related to the management of boosted tokens through the `rewardCache.dropBoost` function call. The issue arises from the potential failure of `dropBoost` to drop boosts associated with the `oldValidator` due to inadequate time passing since the last boost action.

```
File: Staker.sol
    function setValidator(address _newValidator) external onlyOwner {
...
        uint256 boosted = rewardCache.boosts(address(this));
        if (boosted > 0) {
@>          rewardCache.dropBoost(oldValidator, uint128(boosted));
        }
...
@>      validator = _newValidator;
    }
```

In the `dropBoost` function of `rewardCache` (BGT 0x656b95E550C07a9ffe548bd4085c72418Ceb1dba), there exists a condition `_checkEnoughTimePassed` which determines whether sufficient time has passed since the last dropBoost action to allow for dropping boosts. If this condition fails (returns `false`), `dropBoost` will not drop the boosts associated with `oldValidator`, leaving `userBoosts[oldValidator].boost` unchanged. Since the function continues its course, the new validator is assigned (code line `Staker#L90`).

```
function dropBoost(address user, bytes calldata pubkey) external returns
     (bool) {
        // ...
        if (amount == 0 || !_checkEnoughTimePassed
          (blockNumberLast, dropBoostDelay)) return false;
        // ...
    }
```

If boosts for `oldValidator` are not successfully dropped, boosts intended for `_newValidator` will be blocked, potentially causing a leakage of value as boosted tokens are not correctly allocated. In the end, only a few tokens will move to the new validator.

## Recommendations

Add a check to ensure the returned value from dropBoost is true. If it's not, you can revert the transaction:

```
function setValidator(address _newValidator) external onlyOwner {
        ...
        uint256 boosted = rewardCache.boosts(address(this));
        if (boosted > 0) {
-           rewardCache.dropBoost(oldValidator, uint128(boosted));
+           bool successDropBoost = rewardCache.dropBoost(oldValidator, uint128
+ (boosted));
+           require(successDropBoost, "dropBoost function failed");
        }
        ...
```

# [M-06] Troves can be sorted incorrectly due to interest accrual

## Severity

**Impact:** High

**Likelihood:** Low

# Description

The `SortedTroves` contract is expected to keep troves sorted by descending nominal ICR. This is a crucial invariant for the system, as it determines the order in which troves are liquidated and redeemed. However, the interest accrual mechanism can cause troves to be sorted incorrectly.

The NICR is calculated as `collateral / debt` for each trove, and the `collateral` and `debt` values are calculated as follows:

```
function getEntireDebtAndColl(address _borrower)
        public
        view
        returns (
          uint256debt,
          uint256coll,
          uint256pendingDebtReward,
          uint256pendingCollateralReward
        )
    {
        Trove storage t = Troves[_borrower];
        debt = t.debt;
        coll = t.coll;

        (
          pendingCollateralReward,
          pendingDebtReward
        ) = getPendingCollAndDebtRewards(_borrower
        // Accrued trove interest for correct liquidation values. This assumes
        // the index to be updated.
        uint256 troveInterestIndex = t.activeInterestIndex;
        if (troveInterestIndex > 0) {
            (uint256 currentIndex,) = _calculateInterestIndex();
            debt = (debt * currentIndex) / troveInterestIndex;
        }

        debt = debt + pendingDebtReward;
        coll = coll + pendingCollateralReward;
    }
```

For the calculation of the debt, interests are applied over the "consolidated" debt and then the pending debt reward is added. This means that, depending on whether a trove has claimed its debt reward or not, its "consolidated" debt will be different, and thus, the interests accrued over time will differ. This makes it possible for two troves with the same original debt and collateral to diverge in their NICR over time.

The effect of this incorrect sorting of troves will be amplified by the time and when new troves are opened or existing troves are reinserted into the sorted list. As a result, the liquidations and redemptions will not be performed in the correct order, breaking two of the system's core functionalities.

# Proof of concept

Add the following code to the file `test/unit/Roots.t.sol` and run `forge test --mt test_trovesNotSorted`.

```
import {ISortedTroves} from "../../src/interfaces/ISortedTroves.sol";

(...)

    function _openTrove(
        TroveManager troveManager,
        address user,
        uint256 collAmount,
        uint256 debtAmount
    ) private {
        Collateral collateral = Collateral(address(troveManager.collateralToken
          ()));
        vm.startPrank(user);
        collateral.mint(collAmount);
        collateral.approve(address(system.borrowerOperations), collAmount);
        system.borrowerOperations.openTrove(
          troveManager,
          user,
          1e16,
          collAmount,
          debtAmount,
          address
        ), address(0
        vm.stopPrank();
    }

    // Checks if first and second troves are sorted by descending NICR
    function _trovesAreSortedByDescNICR() private view returns (bool) {
        ISortedTroves sortedTrovesA = troveManagerA.sortedTroves();
        address trove0 = sortedTrovesA.getFirst();
        address trove1 = sortedTrovesA.getNext(trove0);
        uint256 nicr0 = troveManagerA.getNominalICR(trove0);
        uint256 nicr1 = troveManagerA.getNominalICR(trove1);
        return nicr0 >= nicr1;
    }

    function test_trovesNotSorted() external {
        address CHARLIE = makeAddr("CHARLIE");

        // Alice, Bob and Charlie open troves
        _openTrove(troveManagerA, ALICE, 4e18, 1e18);
        _openTrove(troveManagerA, BOB, 4e18, 1e18 + 100);
        _openTrove(troveManagerA, CHARLIE, 4e18, 3e18);

        // Troves are sorted correctly
        assert(_trovesAreSortedByDescNICR());

        // Charlie is liquidated and a redistribution of debt and coll occurs
        priceFeed.setPrice(address(collateralA), 0.7e18);
        vm.prank(LIQUIDATOR);
        system.liquidationManager.liquidate(troveManagerA, CHARLIE);

        // Alice claims reward and the pending debt is added to her trove
        vm.prank(ALICE);
        troveManagerA.claimReward(ALICE);

        // Troves are still sorted correctly
        assert(_trovesAreSortedByDescNICR());

        // Interest accrues over 1 month
        skip(30 days);

        // For Alice, interests apply to original debt + redistribution debt
        // As Bob has not claimed his reward, interests apply only to original
        // debt
        // As a result, Alice's debt is now higher than Bob's
        // Troves are not sorted correctly anymore
```

```
        assert(!_trovesAreSortedByDescNICR());

        // Charlie opens a trove with lower NICR than Bob's, but occupies the
        // position
        // of the trove with the highest NICR
        _openTrove(troveManagerA, CHARLIE, 4e18, 1.6705e18);
        ISortedTroves sortedTrovesA = troveManagerA.sortedTroves();
        address trove0 = sortedTrovesA.getFirst();
        address trove1 = sortedTrovesA.getNext(trove0);
        address trove2 = sortedTrovesA.getNext(trove1);
        uint256 nicr0 = troveManagerA.getNominalICR(trove0);
        uint256 nicr2 = troveManagerA.getNominalICR(trove2);
        assert(trove0 == CHARLIE);
        assert(trove2 == BOB);
        assert(nicr0 < nicr2);
    }
```

## Recommendations

Given the complexity of the interest accrual mechanism, there seems to be no straightforward solution to this issue. The team should consider redesigning the interest accrual mechanism to ensure that troves are always sorted correctly or decide to disable this functionality by imposing a zero interest rate.

# [M-07] TroveManager does not work with non-18 decimal tokens

## Severity

**Impact:** High

**Likelihood:** Low

## Description

The `_redeemCollateralFromTrove()` function, used internally by `redeemCollateral()`, assumes all collateral tokens have 18 decimals, matching the debt token. However, when handling tokens with different precisions, incorrect calculations occur due to this formula:

```
singleRedemption.collateralLot =
          (singleRedemption.debtLot * DECIMAL_PRECISION) / _price;
```

Since `singleRedemption.debtLot`, `_price`, and `DECIMAL_PRECISION` are all 18-decimal values, this results in an invalid collateralLot for tokens with different decimals. This leads to:

- Underflows due to precision mismatches.
- Incorrect collateral balances, causing miscalculations in accounting.

Example issue in `newColl` calculation:

```
uint256 newColl = (t.coll) - singleRedemption.collateralLot;
```

Additionally, `_updateBaseRateFromRedemption()` is affected, as `_CollateralDrawn` should match the collateral's precision, while _price and `_totalDebtSupply` remain in 1e18 precision.

# Recommendations

Ensure all calculations are normalized to a consistent precision(i.e. 18 decimals).

## 8.4. Low Findings

### [L-01] `Staker.setRewardsFeeBps()` does not check if the new fee is less than 100%

In order to prevent the rewards fee from being set to a value greater than 100%, it is recommended to check if the new fee is less than or equal to 10,000 basis points.

```
function setRewardsFeeBps(uint16 _rewardsFeeBps) external onlyOwner {
+       require(_rewardsFeeBps <= 10000);
        rewardsFeeBps = _rewardsFeeBps;
    }
```

### [L-02] Inefficient activation of boosts if `BGT`'s `activateBoostDelay` is changed

The `Staker._tryToBoost()` function calls `BGT.activateBoost()` and `BGT.queueBoost()` only when more than 8191 blocks have passed since the last boost.

```
@>  if (queued > 0 && blockDelta > 8191) {
        rewardCache.activateBoost(validator);
    }

    uint256 notInQueue = rewardCache.unboostedBalanceOf(address(this));
@>  if (notInQueue >= boostThreshold && blockDelta > 8191) {
        reward.queueBoost(validator, uint128(notInQueue));
    }
```

While the `activateBoostDelay` parameter in `BGT` is currently set to 8191, this parameter is configurable, so it can be changed in the future. As a result, the activation of boosts will happen less often than they could, resulting in fewer rewards for users.

Instead of using the hardcoded value of 8191 consider checking the current `activateBoostDelay` value in the `BGT` contract. Another approach could be using a storage value in the `Staker` contract, that can be updated by the owner when the `BGT` contract changes the `activateBoostDelay` value.

# [L-03] Users might receive less collateral due to rounding down

In `TroveManager.redeemCollateral()` the troves are looped through until the total debt amount is redeemed or the remaining debt in the trove is less than the minimum debt amount.

In the last iteration, it is possible that `remainingDebt` is so low that the collateral received in exchange rounds down to zero.

```
// Get the CollateralLot of equivalent value in USD
        singleRedemption.collateralLot =
          (singleRedemption.debtLot * DECIMAL_PRECISION) / _price;
```

While the user will not received collateral for the amount of `remainingDebt` debt tokens, these tokens will be burned from his balance and used to reduce the total debt of the trove.

Consider adding a check to prevent burning the user's debt tokens (and reducing the trove's debt) if no collateral is received in exchange.

```
if (
        icrError > 5e14
+            || singleRedemption.collateralLot == 0
             || _getNetDebt(newDebt) < IBorrowerOperations
               (borrowerOperationsAddress).minNetDebt()
    ) {
        singleRedemption.cancelledPartial = true;
        return singleRedemption;
    }
```

# [L-04] Amount of collaterals enabled in `StabilityPool` is not limited

The arrays in `depositSums`, `collateralGainsByDepositor`, `epochToScaleToSums`, and `lastCollateralError_Offset` have a size of 256. However, in the `enableCollateral` function there is no check for the total number of collaterals already enabled in the contract.

If more than 256 collaterals are enabled, the operations that used the arrays mentioned above will revert due to an out-of-bounds access.

Consider adding a check for the total number of collaterals enabled in the `enableCollateral` function.

```
+           require(length < 256);
            collateralTokens.push(_collateral);
            indexByCollateral[_collateral] = collateralTokens.length;
```

# [L-05] Loss evasion by stability depositors

In extreme scenarios, a CDP's collateral ratio may drop below 100%. If this occurs during normal mode, depositors of the stability pool might be motivated to withdraw their deposits to prevent losses before the liquidation process begins.

Consider adding a check in `StabilityPool.withdrawFromSP()` to prevent withdrawals when there are troves that are liquidatable.

See more details in this GH issue from Liquity V1.

# [L-06] Collateral redemptions might be prevented unnecessarily

`TroveManager` does not allow redemption of collateral when the TCR is lower than the MCR.

As we can read in the Liquity V1 documentation, the purpose of this check is to prevent unpegging events when the TCR is close to under-collateralization, that could be triggered by an attacker with a lot of debt tokens.

However, in Root's implementation, the global TCR is checked against the MCR of the specific trove.

```
require(IBorrowerOperations(borrowerOperationsAddress).getTCR
        () >= _MCR, "Cannot redeem when TCR < MCR");
```

Imagine the following scenario:

1. We have a trove for a volatile collateral token with an MCR of 145%.
2. The TCR of the trove is 160%, so it is healthy.
3. The global TCR is 144%, so the system is far from an unpegging event.

While there is no reason to prevent the redemption of collateral, the current implementation would not allow it, as the global TCR is lower than the MCR of the trove.

Consider using a global MCR for the check instead of the trove's MCR and/or comparing the TCR of the trove against the MCR of the trove.

## [L-07] `StabilityPool` and `BorrowerOperations` can be reinitialized

The `initialize()` functions in the `StabilityPool` and `BorrowerOperations` contracts can be called multiple times, changing the value of crucial state variables, that are expected not to change after the initialization.

While these functions can only be called by the owner, which is considered a trusted entity, the capabilities of the owner are limited in other functions to reduce its power to the minimum required. See for example:

```
File: Staker.sol
    function setGauge(
      address_newGauge,
      address_validator,
      uint256_boostThreshold
    ) external onlyOwner {
@>      require(address(gauge) == address(0));
```

It is recommended to add a check in the `initialize()` functions to prevent reinitialization.

```
File: StabilityPool.sol
    function initialize(
      IDebtToken_debtTokenAddress,
      address_factory,
      address_liquidationManager
    )
        external
        onlyOwner
    {
+       require(debtToken == IDebtToken
+ (0), "StabilityPool already initialized");
        debtToken = _debtTokenAddress;
```

```
File: BorrowerOperations.sol
    function initialize
      (address _debtTokenAddress, address _factory) external onlyOwner {
+        require(debtToken == IDebtToken
+ (0), "BorrowerOperations already initialized");
        debtToken = IDebtToken(_debtTokenAddress);
```

## [L-08] The `Staker` and `TroveManager` cannot operate with non-conforming ERC20 tokens

The `Staker` and `TroveManager` contracts call `transfer()`, `transferFrom()`, and `approve()` directly without safety checks. Many ERC-20 tokens deviate from the standard by not returning a value, causing these calls to revert.

In addition, `Staker.onWithdraw()` function is called by the `TroveMnager` to send the user collateral back:

```
function onWithdrawal(address _to, uint256 _amount) external {
        require(msg.sender == address(troveManager), "Only trove manager");

        if (address(gauge) != address(0)) {
            gauge.withdraw(_amount);
        }

        asset.transfer(_to, _amount);
    }
```

- Some ERC20 Tokens do not revert on failure of the transfer function, but return a bool value instead.
- Some do not return any value. Therefore it is required to check if a value was returned, and if true, which value it is.
- If the `onWithdrawal()` transfer silently fails, the funds will remain inside the Staker contract and the borrower has no chance to recover them.

Recommendation: Use OpenZeppelin's SafeERC20 to ensure compatibility and prevent unexpected failures.

## [L-09] Fee on transfer tokens are not supported

The BorrowerOperations contract assume that the amount sent is the same as the amount received:

```
// Move the collateral to the Trove Manager
        collateralToken.safeTransferFrom(msg.sender, address
          (troveManager), _collateralAmount);

        // Create the trove
        (vars.stake, vars.arrayIndex) =
            troveManager.openTrove(
              account,
              _collateralAmount,
              vars.compositeDebt,
              vars.NICR,
              _upperHint,
              _lowerHint
            );
```

However, this assumption does not hold for certain types of tokens:

- Fee-on-transfer tokens: Some tokens, like PAXG, deduct a fee during transfers. Even tokens like USDT have the capability to charge fees, though they currently do not.
- stETH corner case: Tokens like stETH have a 1-2 wei discrepancy, where the received amount is slightly less than expected.
- Rebasing tokens: Some tokens rebase dynamically, meaning balances can increase or decrease over time. stETH also falls into this category.

As a result, BorrowerOperations internal balance may end up being lower than anticipated, leading to failed operations. To ensure accuracy, calculate the actual amount received by measuring the contract's balance before and after the transfer.

# [L-10] Missing events in `Staker` functions

Several important functions within the `Staker` contract, including `setValidator`, `_updateRewardIntegral`, and `claimReward`, lack event emissions.

It is recommended to emit events in the mentioned functions to improve the contract's observability and facilitate off-chain monitoring.

# [L-11] `boostThreshold` check is missing in `setValidator`

In Staker, there is one `boostThreshold` variable. When there are more unboosted BGT in our contract than `boostThreshold`, we will queue and boost. The problem is that in `setValidator` function, we don't check the `boostThreshold` limitation. Then even if we have quite little unboosted BGT(e.g. a few weis), we will queue this unboosted BGT.

```
function setValidator(address _newValidator) external onlyOwner {
        if (balance > 0) {
            rewardCache.queueBoost(_newValidator, uint128(balance));
        }
    }
```

Considering that there is one boost delay, we have to wait several hours to queue more BGTs and we may lost a little bit rewards than expected.

Suggestion: Add the `boostThreshold` check in the `setValidator` function.

# [L-12] Open trove or add collateral may be blocked

When we open one trove or add some extra collateral tokens into one trove, we will stake these collateral tokens into the gauge.

```
function onDeposit(uint256 _amount) external {
        require(msg.sender == address(troveManager), "Only trove manager");
        if (address(gauge) != address(0)) {
            gauge.stake(_amount);
        }
    }
```

When we check the RewardVault(gauge)'s implementation, the `stake` function has one `whenNotPaused` modifier. If this contract is paused for some reason, it will cause that our open trove/ add collateral will be blocked.

```
function stake(uint256 amount) external nonReentrant whenNotPaused {
        _stake(msg.sender, amount);
    }
```

The stake behavior aims to get some more rewards. When the gauge is paused, it should not block our normal open trove or add collateral. Especially, when users want to add some collateral to reduce their borrow position's risk, users' behavior should not be blocked.

# [L-13] Boost activation delay vulnerability in `Staker` due to `setValidator` abuse

The `Staker` contract's `setValidator` function, intended to update the validator address, has an unintended side effect that can be abused by a malicious owner to delay or prevent boost activation. The vulnerability lies in the fact that `setValidator` unconditionally calls `rewardCache.queueBoost(_newValidator, ...)`.

The `queueBoost` function in the `RewardCache` contract updates the `blockNumberLast` timestamp every time it's called, regardless of whether a new boost is actually queued. The `activateBoost` function relies on a time delay (`blockBufferDelay` of `8191` blocks) since `blockNumberLast` to activate a queued boost.

A malicious owner can repeatedly call `setValidator` with the same validator address. Each call will trigger a new `queueBoost` call, resetting the `blockNumberLast` to the current block number. This constant resetting of `blockNumberLast` prevents the `blockDelta` (difference between current block and `blockNumberLast`) from ever exceeding the required `blockBufferDelay` of `8191` blocks. As a result, the `activateBoost` function is not called, effectively stalling the boost activation mechanism.

This vulnerability can negatively impact stakers by delaying or preventing the activation of their boosts.

Consider the next scenario:

1. Initial state:

- `validator` in `Staker` is set to `validatorAddressA`.
- `boostedQueue[stakerContract][validatorAddressA].blockNumberLast` is at some block number (let's say, block `100`).
- `boostedQueue[stakerContract][validatorAddressA].balance` has some queued boost amount.

2. Malicious owner action:

- The owner of the `Staker` contract calls `setValidator(validatorAddressA)`. Note that `validatorAddressA` is the *same* as the current `validator`.
- Inside `setValidator`:
  - `rewardCache.queueBoost(validatorAddressA, ...)` is called.
  - In `queueBoost`, `boostedQueue[stakerContract][validatorAddressA].blockNumberLast` is updated to the *current* `block.number` (let's say, block `200`).

```
File: Staker.sol
85:         uint256 balance = rewardCache.unboostedBalanceOf(address(this));
86:         if (balance > 0) {
87:@>          rewardCache.queueBoost(_newValidator, uint128(balance));
88:         }
```

```
File: BGT.sol (Inferred RewardCache Contract)
217:      function queueBoost(
  bytescalldatapubkey,
  uint128amount
) external checkUnboostedBalance(msg.sender, amount
218:          userBoosts[msg.sender].queuedBoost += amount;
219:          unchecked {
220:              QueuedBoost storage qb = boostedQueue[msg.sender][pubkey];
221:              // `userBoosts[msg.sender].queuedBoost` >= `qb.balance`
222:              // if the former doesn't overflow, the latter won't
223:              uint128 balance = qb.balance + amount;
224:@>            (qb.balance, qb.blockNumberLast) = (balance, uint32
  (block.number));
225:          }
226:          emit QueueBoost(msg.sender, pubkey, amount);
227:      }
```

3. Repeated abuse:

- The malicious owner repeatedly calls `setValidator(validatorAddressA)` every few blocks.
- Each call resets `boostedQueue[stakerContract][validatorAddressA].blockNumberLast` to the latest block number.

4. Boost Activation Prevention:

- The `_tryToBoost` function in `Staker` checks `blockDelta = block.number - lastBoost`.
- Because `lastBoost` (which is `boostedQueue[stakerContract][validatorAddressA].blockNumberLast`) is constantly being updated to recent block numbers by the malicious owner, `blockDelta` will almost always be less than `8191`.
- The condition `blockDelta > 8191` in `_tryToBoost` will rarely be met.
- Therefore, `rewardCache.activateBoost(validator)` in `_tryToBoost` is not called, and the queued boost remains inactive.

```
File: Staker.sol
    function _tryToBoost() internal {
        IBGT rewardCache = reward;

        (uint32 lastBoost, uint128 queued) = rewardCache.boostedQueue(address(this), validator);

        uint256 blockDelta = block.number - lastBoost;

        if (queued > 0 && blockDelta > 8191) {
            rewardCache.activateBoost(validator);
        }

        uint256 notInQueue = rewardCache.unboostedBalanceOf(address(this));
        if (notInQueue >= boostThreshold && blockDelta > 8191) {
            reward.queueBoost(validator, uint128(notInQueue));
        }
    }
```

Recommendations:

To prevent unnecessary queuing, the condition `blockDelta > 8191` should be validated in `setValidator`. This ensures that validator migration only becomes feasible after a queuing process from a `_tryBoost` execution has concluded and the required delay period has elapsed. Also, it seems to me that it is necessary to start a "validator migration process" so that `_tryBoost` is not executed again before owner can execute `setValidator`.

Additionally, modify the `setValidator` function in the `Staker` contract to prevent calling `rewardCache.queueBoost` when the `_newValidator` address is the same as the current `validator`.