Proportionalized Audit Report

Prepared by: InAllHonesty

Table of Contents

Protocol Summary

Proportionalized is a simple staking protocol that awards reward tokens to users that stake specific tokens.

Disclaimer

InAllHonesty made all the efforts to find as many vulnerabilities in the code in the given time period of 48 hours, but holds no responsibilities for the findings provided in this document. A security audit is not an endorsement of the underlying business or product. The audit was time-boxed and the review of the code was solely on the security aspects of the Solidity implementation of the contracts.

Risk Classification

	Impact: High	**Impact: Medium**	**Impact: Low**
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

Impact refers to the potential harm or consequence to the users or the protocol due to the vulnerability.

High Impact:

- Funds are directly or nearly directly at risk.
- There's a severe disruption of protocol functionality or availability.

Medium Impact:

- Funds are indirectly at risk.
- There's some level of disruption to the protocol's functionality or availability.

Low Impact:

- Funds are not at risk.
- However, a function might be incorrect, state might not be handled appropriately, etc.

How to evaluate the Likelihood of exploitation

Likelihood represents the probability of the impact occurring due to the vulnerability.

High Likelihood:

Highly probable to happen. For instance, a hacker can call a function directly and extract money.

Medium Likelihood:

It might occur under specific conditions. For example, a peculiar ERC20 token is used on the platform.

Low Likelihood:

Unlikely to occur. An example might be if a hard-to-change variable is set to a unique value on a very specific block.

Audit Details

The findings described below correspond to the following commit hash:

```
00f1b227f61692fcecbda1d03d48f4e017516075
```

Scope

The following contracts:

- Rewards.sol
- Staking.sol
- Token.sol

Roles

The protocol has only two roles:

- Owner: The user who is allowed to call setters and emergency functions.
- Regular User: The user who stakes and earns rewards.

Executive Summary

The audit was carried over 48 hours. The protocol was checked for regular staking projects vulnerabilities using manual review as well as automated tools.

Issues found

Severity	Number of issues found
High	1
Medium	5
Low	3
Informational	4
Gas	7

Findings

High

[H-01] The way `minimumStake` is configured makes it impossible to work with tokens that do not have 18 decimals

Description: The protocol will not work with USDT/USDC or any other different decimal != 18 tokens.

`minimumStake` is first declared here

```
    uint256 public minimumStake = 10_000 ether;
```

additionally, there's a setter function here

```
function setMinimumStake(uint256 _minimumStake) external onlyOwner {
    require(_minimumStake > 1_000 ether, "Minimum amount too low");
    minimumStake = _minimumStake;
}
```

In both instances, the `minimumStake` is configured to be higher than `1_000 ether`.

`ether` is a Solidity suffix that has the value of `1e18`. Thus the `minimumStake` is at least `1000 * 1e18`.

If this limit is applied to a **6 decimal token like USDT/USDC** then for a user's transaction to go through they would need to send at least `1000 * 1e12` USDT/USDC, which is a ridiculous amount of money.

Recommendation:

Perform the following changes:

```
[...]
   IERC20 public immutable stakingToken;
   Rewards public rewardsContract;
   address public feeCollector;
   uint256 public totalStaked;
--  uint256 public minimumStake = 10_000 ether;
++  uint256 public minimumStake
   uint256 public roundDuration = 7 days;
   uint256 public roundFinishAt;
   uint256 public roundUpdatedAt;
   uint256 public roundRewardRate;
[...]

--  constructor(address _owner, address _stakingToken, address _feeCollector)
Ownable(_owner) {
++  constructor(address _owner, address _stakingToken, address _feeCollector,
uint256 _minimumStake) Ownable(_owner) {
       require(_feeCollector != address(0), "Invalid fee collector address");
       stakingToken = IERC20(_stakingToken);
       feeCollector = _feeCollector;
++     minimumStake = _minimumStake;
   }

[...]

   function setMinimumStake(uint256 _minimumStake) external onlyOwner {
--     require(_minimumStake > 1_000 ether, "Minimum amount too low");
       minimumStake = _minimumStake;
   }
```

We need to be more flexible in setting the `minimumStake`, or we will not be able to accommodate different decimal tokens.

Medium

[M-01] Centralization Risk

Description: The `owner` role is a single point of failure and `onlyOwner` can use a few critical functions. Whether it is done due to ill intent or due to being hacked/losing access to the private keys, the owner has the ability to `emergencyTransferStakedTokens` and `recoverTokens` in `Staking.sol` and `emergencyTransferRewardTokens` and `recoverTokens` in `Rewards.sol`, functions that a hacker could use to completely drain the contracts.

Moreover, the owner has the power to stop (`setStopped`) the contract and complete control over the unstaking fees which can be set as high as 100%.

```
    function setRegularUnstakeFeePercentage(uint256 _feePercentage) external
onlyOwner {
        require(_feePercentage <= 10000);
        regularUnstakeFeePercentage = _feePercentage;
    }

    function setFastUnstakeFeePercentage(uint256 _feePercentage) external
onlyOwner {
        require(_feePercentage <= 10000);
        fastUnstakeFeePercentage = _feePercentage;
    }
```

The unstake cooldown for both the regular and fast unstakes can be set as high as `90 days`.

```
    function setRegularUnstakeCooldown(uint256 _cooldown) external onlyOwner {
        require(_cooldown <= 90 days);
        regularUnstakeCooldown = _cooldown;
    }

    function setFastUnstakeCooldown(uint256 _cooldown) external onlyOwner {
        require(_cooldown <= 90 days);
        fastUnstakeCooldown = _cooldown;
    }
```

The reward allocation can be changed to values as low as `1e6` inside the duration of the `roundDuration`.

Recommendation:
Make the owner a multi-sig and/or introduce a timelock on sensitive functions.

[M-02] The way the `Staking::setRewardsAllocation` is set up will force the protocol to always use the same token both as `stakingToken` and `rewardToken`, which represents an unnecessary limitation

Description: The `Staking::setRewardsAllocation` function performs a check regarding the Rewards token balance, to ensure that the stakers will be able to receive their funds, and prevent any `ERC20InsufficientBalance` errors.

```
    function setRewardsAllocation(uint256 _amount) external onlyOwner
updateReward(address(0)) {
        require(address(rewardsContract) != address(0), "Rewards contract address
is not set");

        if (block.timestamp >= roundFinishAt) {
            roundRewardRate = _amount / roundDuration;
        } else {
            uint256 remainingRewards = (roundFinishAt - block.timestamp) *
roundRewardRate;
            roundRewardRate = (_amount + remainingRewards) / roundDuration;
        }
        require(roundRewardRate > 0, "Reward rate = 0");
@>      require(roundRewardRate * roundDuration <=
stakingToken.balanceOf(address(rewardsContract)), "Reward amount > balance");
        roundFinishAt = block.timestamp + roundDuration;
        roundUpdatedAt = block.timestamp;
    }
```

The problem is `require(roundRewardRate * roundDuration <= stakingToken.balanceOf(address(rewardsContract)), "Reward amount > balance");` forces the `Rewards` contract to use only the `stakingToken`.

Recommendation:

Perform the following changes:

```
    function setRewardsAllocation(uint256 _amount) external onlyOwner
updateReward(address(0)) {
       require(address(rewardsContract) != address(0), "Rewards contract address
is not set");

       if (block.timestamp >= roundFinishAt) {
           roundRewardRate = _amount / roundDuration;
       } else {
           uint256 remainingRewards = (roundFinishAt - block.timestamp) *
roundRewardRate;
           roundRewardRate = (_amount + remainingRewards) / roundDuration;
       }
       require(roundRewardRate > 0, "Reward rate = 0");
--      require(roundRewardRate * roundDuration <=
stakingToken.balanceOf(address(rewardsContract)), "Reward amount > balance");
++      require(roundRewardRate * roundDuration <=
IERC20(rewardsContract.rewardToken()).balanceOf(address(rewardsContract)), "Reward
amount > balance");
       roundFinishAt = block.timestamp + roundDuration;
```

```
        roundUpdatedAt = block.timestamp;
    }
```

In this way, we check the `rewardToken` configured in the `Rewards` contract. Thus if we want to pay `token1` as rewards we could do it, if after some time a better opportunity presents itself for the protocol, i.e. `token2`, we just need to redeploy `Rewards` using `token2` as `rewardToken` and the existing functionality of `Staking` contract would allow us to call `setRewardsContract` + `setRewardsAllocation` and switch to new rewards. Or, even better, to avoid the necessity to redeploy `Rewards` whenever we want to change the reward token, we could drop the `immutable` characteristic from `IERC20 public immutable rewardToken;` in `Rewards.sol` and configure a setter that changes the reward token. In that way, the owner has full flexibility with minimum costs.

[M-03] `minimumStake` invariant is broken by unstaking

Description: Whenever a user stakes the protocol enforces a `minimumStake`, a parameter that can be modified via `Staking::setMinimumStake`. The problem is whenever a user calls `Staking::initiateUnstake` they can initiate the unstake of any amount, regardless of what's left in `userStake.amount`, under the condition that `_amount > 0 && _amount <= userStake.amount`. This means that `userStake.amount` can fall under `minimumStake` as long as the user waits for the `regularUnstakeCooldown`/`fastUnstakeCooldown` and pays the associated fee, rendering the whole concept of a `minimumStake` useless.

POC:

```
    function testStakeUnstakeBreakMin() public {
        deal(address(token), user1, 10_000 * 1e18);
        //Perform stake and initiateUnstake immediately
        vm.startPrank(user1);
        token.approve(address(staking), 10_000 * 1e18);
        staking.stake(10_000 * 1e18);
        staking.initiateUnstake(9_500 * 1e18, true);
        vm.stopPrank();

        assertEq(token.balanceOf(address(staking)), 10_000 * 1e18);
        assertEq(staking.totalStaked(), 10_000 * 1e18);

        //Time passes 1 day
        vm.warp(timeNow + 1 days);

        //Call unstake
        vm.startPrank(user1);
        staking.unstake();
        vm.stopPrank();

        (uint256 amount,,,,,) = staking.stakes(user1);
        //Check the remaining amount is < than minimumStake
        assertLt(amount, staking.minimumStake());
    }
```

Recommendation:

Enforce the `minimumStake` in the unstaking process the same way it's enforced in the staking process. The user should have two options:

1. Unstake the full amount;
2. Unstake an arbitrary amount while leaving at least the `minimumStake` in the protocol;

```
    function initiateUnstake(uint256 _amount, bool _fastUnstake) external
nonReentrant stoppedEmergency {
        Stake storage userStake = stakes[msg.sender];
        require(_amount > 0 && _amount <= userStake.amount, "Invalid unstake
amount");
        require(!userStake.initiatedUnstake, "Unstake already initiated");
++      require(userStake.amount - _amount >= minimumStake || userStake.amount -
_amount == 0, "Invalid unstake amount");
        userStake.initiatedUnstake = true;
        userStake.initiatedUnstakeAmount = _amount;
        userStake.initiatedUnstakeFast = _fastUnstake;
        userStake.initiatedUnstakeTime = block.timestamp;

        emit InitiatedUnstake(msg.sender, _amount, _fastUnstake);
    }
```

[M-04] The `regularUnstakeCooldown`/`fastUnstakeCooldown` and/or `regularUnstakeFeePercentage`/`fastUnstakeFeePercentage` setters affect the ongoing cooldown and fee of a user that called `initiateUnstake` but did not call `unstake`

Description: When a user calls `initiateUnstake` they do it knowing a set of parameters under which the unstaking takes place, i.e. a cooldown and a fee. Those vary depending on whether the user chooses a regular unstaking or a fast unstaking. The problem arises when within the cooldown period the owner calls one of the 4 setters that concern unstaking parameters: `setRegularUnstakeCooldown`, `setFastUnstakeCooldown`, `setRegularUnstakeFeePercentage` and `setFastUnstakeFeePercentage`.

POC:

```
    function testStakeInitUnstakeChangeCdandFee() public {
        // We are testing the scenario when user 1 stakes and after 3 days owner
calls setRewardsAllocation to a different amount
        deal(address(token), user1, 100_000 * 1e18);
        uint256 userBalanceBefore = token.balanceOf(user1);

        // Perform first stake
        vm.startPrank(user1);
        token.approve(address(staking), 100_000 * 1e18);
        staking.stake(100_000 * 1e18);
        vm.stopPrank();

        assertEq(token.balanceOf(address(staking)), 100_000 * 1e18);
```

```
        assertEq(staking.totalStaked(), 100_000 * 1e18);

        // Warp 3 days into the future
        vm.warp(timeNow + 3 days);

        // User initiates unstake (cd 7 days)
        vm.startPrank(user1);
        staking.initiateUnstake(100_000 * 1e18, false);
        vm.stopPrank();

        // Owner changes the normal unstaking period
        vm.startPrank(owner);
        staking.setRegularUnstakeCooldown(30 days); // from 7 days to a month
        staking.setRegularUnstakeFeePercentage(9000); // from 1% to 90%
        vm.stopPrank();

        // Warp 7 days into the future
        vm.warp(timeNow + 10 days);

        // Try unstaking after the initial normal unstaking period passes, it
reverts
        vm.startPrank(user1);
        vm.expectRevert();
        staking.unstake();
        vm.stopPrank();

        // Warp 30 days into the future
        vm.warp(timeNow + 33 days);

        // Try unstaking after the new normal unstaking period passes
        vm.startPrank(user1);
        staking.unstake();
        vm.stopPrank();

        // Unstaking the full amount in 7 days with 1% fee as intended became
available in 30 days with 90% fee
        assertEq(token.balanceOf(user1), 100_000 * 1e18 * 1_000 / 10_000);
    }
```

Recommendation: The cooldown and the owed fee should be recorded in the Stake struct.

```
struct Stake {
    uint256 amount;
    uint256 rewards;
    uint256 rewardsRate;
    bool initiatedUnstake;
    bool initiatedUnstakeFast;
    uint256 initiatedUnstakeAmount;
    uint256 initiatedUnstakeTime;
++  uint256 cooldown;
```

```
++  uint256 feePercentage;
}
```

These two new `userStake` parameters will be updated inside the `initiateUnstake`. When the user calls `unstake` the cooldown check and the fee calculation will pick up the appropriate parameters in the `userStake`.

[M-05] In cases where the `roundDuration` is very long and `regularUnstakeCooldown` is small the user can still earn while having a lesser risk compared to the other users of the protocol

Description: The root of the problem is the user's ability to earn after the `unstake` cooldown expires. The current settings of the project don't show this risk, because, with a `roundDuration` of 7 days and a `regularUnstakeCooldown` of 7 days, this situation doesn't exist, but let's analyze the following situation:

The owner decides to provide rewards for 1 year, leaving the regular `unstake` cooldown at 7 days.
A user can come on day 1, deposit an amount, and immediately call `initiateUnstake` with the full amount. After 7 days they can call `unstake` anytime they want, while still being able to earn the rewards associated with the stake like all the other users for the rest of the year. Is there a better opportunity available within this project or somewhere else? No problem, call `unstake` instantly get access to your funds and move on. Did something happen with the project and the owner is forced to call `setStopped(true)`? No problem, simply front-run the `setStopped(true)` call by calling `unstake` with a higher gas fee and get access to your funds before they get locked.

There is an obvious risk discrepancy that will be corrected by the market, i.e. everyone will stake and initiate unstake to benefit from the flexibility while losing nothing.

POC:

```
    function testStakeInitUnstakeKeepsEarningRewards() public {
        deal(address(token), user1, 100_000 * 1e18);

        // Set a small unstake CD
        vm.startPrank(owner);
        staking.setRegularUnstakeCooldown(1 days);
        vm.stopPrank();

        // Perform first stake and immediately call initunstake
        vm.startPrank(user1);
        token.approve(address(staking), 100_000 * 1e18);
        staking.stake(100_000 * 1e18);
        staking.initiateUnstake(100_000 * 1e18, false);
        vm.stopPrank();

        assertEq(token.balanceOf(address(staking)), 100_000 * 1e18);
        assertEq(staking.totalStaked(), 100_000 * 1e18);

        // 1 day passed - user1 is now eligible to unstake
        // vm.warp(timeNow + 1 days);
        // vm.startPrank(user1);
```

```
        // staking.unstake();
        // vm.stopPrank();
        // Testing if the user receives back his initial amount - 1% fee - it
passes, we comment out to continue demonstration
        // assertEq(token.balanceOf(user1), (100_000 - 1_000) * 1e18);

        // 7 days passed - the whole roundDuration
        vm.warp(timeNow + 7 days);
        vm.startPrank(user1);
        staking.claimRewards();
        vm.stopPrank();

        console.log("User1 claimed balance:", token.balanceOf(user1));
        assertApproxEqAbs(token.balanceOf(user1), initialRewardAllocation, 1e6);

        // User claimed the rewards for a full 7 days period
    }
```

Recommendation: Change the way rewards are computed for the users with `userStake.initiatedUnstake = true;` by making sure they are eligible to earn rewards up to the point their cooldown expired, thus eliminating the incentive to try to game the protocol.

Low

[L-01] Setter functions should emit events

The protocol should emit events for critical parameter changes.
These events should include all the relevant information like the new and old parameter values.

Functions that change critical parameters and lack events:
In `Rewards.sol`: `setStakingContract`;
In `Staking.sol`: `setRewardsAllocation`, `setRegularUnstakeCooldown`, `setFastUnstakeCooldown`, `setRegularUnstakeFeePercentage`, `setFastUnstakeFeePercentage`, `setRewardsContract`, `setRewardsDuration`, `setFeeCollector`, `setMinimumStake` and `setStopped`.

[L-02] Return not checked for EnumerableSet functions

The contract uses `add` and `remove` from OZ's EnumerableSet. The problem is these two functions don't revert on failure, they simply return `false`:

```
[...]
    function _add(Set storage set, bytes32 value) private returns (bool) {
        if (!_contains(set, value)) {
            set._values.push(value);
            // The value is stored at length-1, but we add 1 to all indexes
            // and use 0 as a sentinel value
            set._positions[value] = set._values.length;
            return true;
        } else {
            return false;
        }
    }
[...]
    function _remove(Set storage set, bytes32 value) private returns (bool) {
        // We cache the value's position to prevent multiple reads from the same
storage slot
        uint256 position = set._positions[value];

        if (position != 0) {
            // Equivalent to contains(set, value)
            // To delete an element from the _values array in O(1), we swap the
element to delete with the last one in
            // the array, and then remove the last element (sometimes called as
'swap and pop').
            // This modifies the order of the array, as noted in {at}.

            uint256 valueIndex = position - 1;
            uint256 lastIndex = set._values.length - 1;

            if (valueIndex != lastIndex) {
                bytes32 lastValue = set._values[lastIndex];

                // Move the lastValue to the index where the value to delete is
```

```
                set._values[valueIndex] = lastValue;
                // Update the tracked position of the lastValue (that was just
moved)
                set._positions[lastValue] = position;
            }

            // Delete the slot where the moved value was stored
            set._values.pop();

            // Delete the tracked position for the deleted slot
            delete set._positions[value];

            return true;
        } else {
            return false;
        }
    }
[...]

    function add(Bytes32Set storage set, bytes32 value) internal returns (bool) {
        return _add(set._inner, value);
    }

    function remove(Bytes32Set storage set, bytes32 value) internal returns (bool)
{
        return _remove(set._inner, value);
    }
```

Moreover, in `Staking::stake`, `Staking::compoundStake` and `Staking::migrateStakers`, the following check is performed before using `add`:

```
        if (!stakers.contains(msg.sender)) {
            stakers.add(msg.sender);
        }
```

But as we can see in the `_add` function above, this check is already performed inside the `add` function.

Recommendation

```
    function stake(uint256 _amount) external nonReentrant stoppedEmergency
updateReward(msg.sender) {
        require(_amount > 0, "Amount is 0");

        Stake storage userStake = stakes[msg.sender];

        require(userStake.amount + _amount >= minimumStake, "Stake amount too
low");

        stakingToken.safeTransferFrom(msg.sender, address(this), _amount);
```

```
        userStake.amount += _amount;
        totalStaked += _amount;

--        if (!stakers.contains(msg.sender)) {
--            stakers.add(msg.sender);
--        }
++      bool success = stakers.add(msg.sender);
++      require(success, "EnumerableSet add failed");
        emit Staked(msg.sender, _amount, totalStaked);
    }
```

The same modification should be applied in the case of `Staking::compoundStake`, `Staking::migrateStakers` and in the `Staking::_removeStaker` (with `remove` instead of `add`).

[L-03] Users who unstake their whole staked balance should be forced to claim their rewards as well

Description: Not forcing a user who decided to unstake all the staked amount to claim his rewards as well facilitates a weird situation in which the user has nothing staked, but still has rewards to be claimed that do not earn anything. This is extremely capital inefficient for the user, and the only way this happens is by user mistake.

The protocol was forced to take this into account when designing `Staking::compoundStake`, thus the `coumpundStake` function checks if the user has unclaimed rewards via `require(userStake.rewards > 0, "No rewards to compound");` and then checks again if the user is in the Enumerable set, and if not it adds him. This makes the `coumpundStake` function inefficient from a gas perspective because it needs to account for a very niche case while making the whole function more gas-expensive for every other user.

Recommendation: To eliminate both the capital inneficiency caused by the user mistake and the need to check if the user is in `stakers` inside the `coumpundStake` function perform the following modifications:

```
[...]
    function compoundStake() external nonReentrant stoppedEmergency
updateReward(msg.sender) {
        Stake storage userStake = stakes[msg.sender];
        require(userStake.rewards > 0, "No rewards to compound");

        uint256 rewardsEarned = earned(msg.sender);
        userStake.rewards = 0;
        userStake.amount += rewardsEarned;
        totalStaked += rewardsEarned;

        rewardsContract.transferRewards(address(this), rewardsEarned);

--      if (!stakers.contains(msg.sender)) {
--          stakers.add(msg.sender);
--      }

        emit RewardsCompounded(msg.sender, rewardsEarned);
```

```
    }
[...]
    function unstake() external nonReentrant stoppedEmergency
updateReward(msg.sender) {
        Stake storage userStake = stakes[msg.sender];
        require(userStake.initiatedUnstake, "Unstake not initiated");
        uint256 cooldown = userStake.initiatedUnstakeFast ? fastUnstakeCooldown :
regularUnstakeCooldown;
        require(block.timestamp >= userStake.initiatedUnstakeTime + cooldown,
"Cooldown not met");

        uint256 feePercentage = userStake.initiatedUnstakeFast ?
fastUnstakeFeePercentage : regularUnstakeFeePercentage;
        uint256 fee = (userStake.initiatedUnstakeAmount * feePercentage) / 10000;
        uint256 amountAfterFee = userStake.initiatedUnstakeAmount - fee;

        userStake.initiatedUnstake = false;
        userStake.amount -= userStake.initiatedUnstakeAmount;
        totalStaked -= userStake.initiatedUnstakeAmount;

        stakingToken.safeTransfer(msg.sender, amountAfterFee);
        stakingToken.safeTransfer(feeCollector, fee);
++      if (userStake.rewards > 0) {
++          claimRewards();
++      }
        if (userStake.amount == 0) {
            _removeStaker(msg.sender);
        }

        emit Unstaked(msg.sender, userStake.initiatedUnstakeAmount, totalStaked,
userStake.initiatedUnstakeFast);
    }

--    function claimRewards() external nonReentrant stoppedEmergency
updateReward(msg.sender) {
++    function claimRewards() public nonReentrant stoppedEmergency
updateReward(msg.sender) {
```

We added a `userStake.rewards > 0` to ensure that `claimRewards()` does not revert the `unstake` function.

Informational

[I-01] Solidity pragma should be specific, not wide

The current pragma Solidity directive is `^0.8.24`. It is recommended to specify a specific compiler version to ensure that the byte code produced does not vary between builds. Contracts should be deployed using the same compiler version/flags with which they have been tested.

[I-02] Empty `require()` statements

Readability could be improved by adding a string message.

The following were found in `Staking.sol`

- Line: 250

```
        require(_cooldown <= 90 days);
```

- Line: 255

```
        require(_cooldown <= 90 days);
```

- Line: 260

```
        require(_feePercentage <= 10000);
```

- Line: 265

```
        require(_feePercentage <= 10000);
```

[I-03] Dust left in the `Rewards` contract after a full `roundDuration` period

Due to the way rewards are computed, first in the `rewardPerToken` function which performs a division, which truncates the final result, then this returned value is used in `_earned` which is again divided at some point by `1e18`, leads to a small precision loss. This leaves dust amounts in the `Rewards` contract, which the owner should correct from time to time by calling the `emergencyTransferRewardTokens` function in the `Rewards` contract.

[I-04] Specifically state the visibility of variables to avoid confusion and increase the readability of the code

`bool isStopped = true;` is treated as internal by default, whether this is intended or not its visibility should be specified.

```
--    bool isStopped = true;
++    bool internal isStopped = true;
```

[I-05] Protocol lacks NatSpec comments

It is used to document the behavior and purpose of functions and contracts directly within the source code, making it easier for developers and auditors to understand the contract's functionality.

NatSpec comments should be included in all contracts in scope.

Gas

[G-01] If + Custom errors are more gas-efficient than require + "error message"

Custom errors decrease both deploy and runtime gas costs and should be used instead of the `require + "error message"` statements.

Example modification for `Staking.sol`:

```
++  error Staking__ZeroAddress();

    constructor(address _owner, address _stakingToken, address _feeCollector)
Ownable(_owner) {
--      require(_feeCollector != address(0), "Invalid fee collector address");
++      if (_feeCollector == address(0)) revert Staking__ZeroAddress();
        stakingToken = IERC20(_stakingToken);
        feeCollector = _feeCollector;
    }
```

[G-02] `public` functions not used internally could be marked as `external`

Functions `recoverTokens`, `setStopped`, `stakeOf`, `getStakersNumber`, `getUnstakeCooldown` in `Staking.sol` and function `recoverTokens` in `Rewards.sol` visibility should be changed from public to external to improve gas efficiency and make the contract's intent clearer.

[G-03] Define and use `constant` variables instead of using literals

Throughout the contract, the number `10000` is used to represent the percentage precision and 100%.

It should be defined as a constant
`uint256 constant FEE_PRECISION = 10000;`

and perform the following changes:

```
[...]
    function unstake() external nonReentrant stoppedEmergency
updateReward(msg.sender) {
        Stake storage userStake = stakes[msg.sender];
        require(userStake.initiatedUnstake, "Unstake not initiated");
        uint256 cooldown = userStake.initiatedUnstakeFast ? fastUnstakeCooldown :
regularUnstakeCooldown;
        require(block.timestamp >= userStake.initiatedUnstakeTime + cooldown,
"Cooldown not met");

        uint256 feePercentage = userStake.initiatedUnstakeFast ?
fastUnstakeFeePercentage : regularUnstakeFeePercentage;
--      uint256 fee = (userStake.initiatedUnstakeAmount * feePercentage) / 10000;
++      uint256 fee = (userStake.initiatedUnstakeAmount * feePercentage) /
FEE_PRECISION;
```

```
        uint256 amountAfterFee = userStake.initiatedUnstakeAmount - fee;
[...]

    function setRegularUnstakeFeePercentage(uint256 _feePercentage) external
onlyOwner {
--      require(_feePercentage <= 10000);
++      require(_feePercentage <= FEE_PRECISION);
        regularUnstakeFeePercentage = _feePercentage;
    }

    function setFastUnstakeFeePercentage(uint256 _feePercentage) external
onlyOwner {
++     require(_feePercentage <= FEE_PRECISION);
        fastUnstakeFeePercentage = _feePercentage;
    }
[...]
```

The same changes can be performed for the `90 days` cooldown limit and for `1e18` throughout the contract. All these help save gas and improve code readability.

[G-04] Modifiers invoked only once should be moved inside the function

The `onlyStopped()` modifier is used only once in the `migrateStakers` function. The deployment cost will be decreased if we simply add a `require(isStopped, "not stopped");` at the beginning of the `migrateStakers` function.

[G-05] `nonReentrant` modifier is used everywhere

The protocol uses only standard ERC20 tokens and generally follows the Check-Effects-Interactions pattern. The `nonReentrant` modifier applied on every single function only consumes gas, both on function call and on deployment. The only external calls made are `safeTransferFrom` and `safeTransfer` from ERC20, and these cannot be reentered.

[G-06] Function `Staking::_lastTimeRewardApplicable` can be unpacked in the 2 places it's used in order to save on deployment costs

```
    modifier updateReward(address _account) {
        uint256 newRoundRewardRatePerToken = rewardPerToken();
        roundRewardRatePerToken = newRoundRewardRatePerToken;
--      roundUpdatedAt = _lastTimeRewardApplicable(roundFinishAt);
++      roundUpdatedAt = _min(roundFinishAt, block.timestamp);
        if (_account != address(0)) {
            Stake storage userStake = stakes[_account];
            userStake.rewards = _earned(userStake, newRoundRewardRatePerToken);
            userStake.rewardsRate = roundRewardRatePerToken;
        }
        _;
    }
```

```
    function rewardPerToken() public view returns (uint256) {
        uint256 total = totalStaked;
        if (total == 0) {
            return roundRewardRatePerToken;
        }
--       return roundRewardRatePerToken + (roundRewardRate *
(_lastTimeRewardApplicable(roundFinishAt) - roundUpdatedAt) * 1e18) / total;
++       return roundRewardRatePerToken + (roundRewardRate * (_min(roundFinishAt,
block.timestamp) - roundUpdatedAt) * 1e18) / total;
    }
```

[G-07] The visibility of some of the state variables should be changed to save gas

The `feeCollector`, `totalStaked`, `roundFinishAt`, `roundUpdatedAt`, `roundRewardRate` and `roundRewardRatePerToken` in `Staking.sol` should be made `internal`. The `stakingContract` variable in `Rewards.sol` should be made internal.