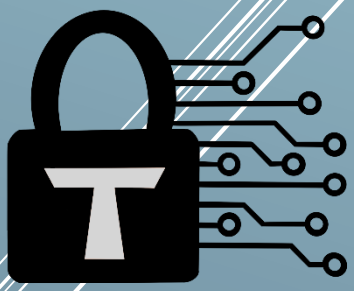
Trust Security

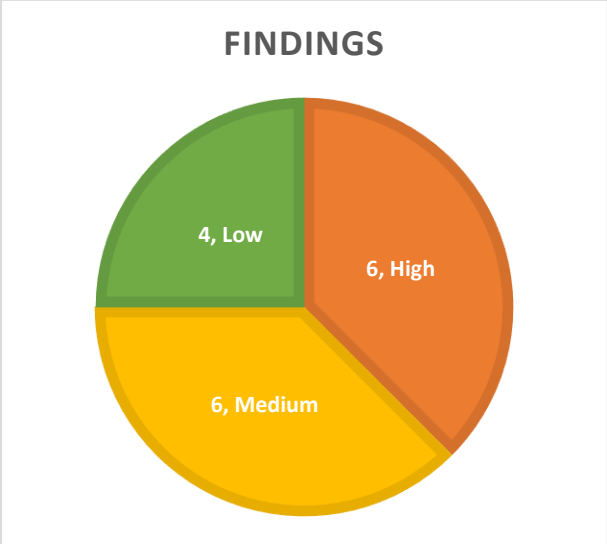


Smart Contract Audit

BugHole LSD

10/07/24

Executive summary

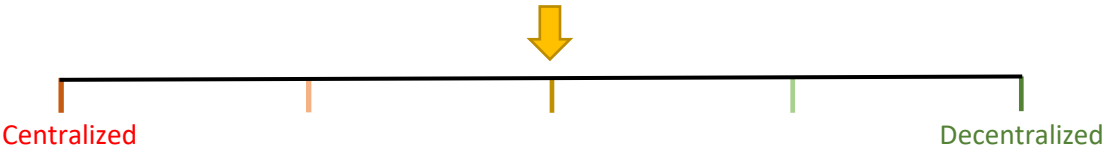


Category	Liquid Staking
Audited file count	15
Lines of Code	931
Auditor	cccz ether_sky
Time period	03/07/2024- 10/07/2024

Findings

Severity	Total	Fixed	Open	Acknowledged
High	6	-	-	-
Medium	6	-	-	-
Low	4	-	-	-

Centralization score



Signature

Digitally signed by Trust
Date: 2024.07.10 16:29:18 +03'00'

Document properties

Versioning

Version	Date	Description
0.1	10/07/2024	Client report

Contact

Trust

trust@trust-security.xyz

Introduction

Trust Security serves as a long-term security partner of the Reserve Protocol. It has conducted the audit at the customer's request. The audit is focused on uncovering security issues and additional bugs contained in the code defined in scope. Additional recommendations have been given when appropriate.

Scope

- contracts/bughole/NodeController/NodeController.sol
- contracts/bughole/StakingToken/StakingToken.sol
- contracts/bughole/NodeService/NodeService.sol
- contracts/bughole/interface/kaia/ICnStakingV3.sol
- contracts/bughole/interface/kaia/IPublicDelegation.sol
- contracts/bughole/NodeController/INodeController.sol
- contracts/bughole/library/Validator.sol
- contracts/bughole/NodeService/INodeService.sol
- contracts/bughole/enums/State.sol
- contracts/bughole/StakingToken/IStakingToken.sol
- contracts/bughole/structs/Unstake.sol
- contracts/bughole/Treasury/Treasury.sol
- contracts/bughole/structs/Claim.sol
- contracts/bughole/structs/Node.sol
- contracts/bughole/interface/kaia/IKIP163.sol

Repository details

- **Repository URL:** https://github.com/bug4city/bughole-lsd
- **Commit hash:** 57259a82a6945f400ce158149ac5712e4300496b

About Trust Security

Trust Security has been established by top-end blockchain security researcher Trust, in order to provide high quality auditing services. Trust is the leading auditor at competitive auditing service Code4rena, reported several critical issues to Immunefi bug bounty platform and is currently a Code4rena judge.

About the Auditors

A top competitor in audit contests, cccz has achieved superstar status in the security space. He is a Black Hat / DEFCON speaker with rich experience in both traditional and blockchain security.

ether_sky is a security researcher with a focus on blockchain security. He specializes in algorithms and data structures and has a solid background in IT development. He has placed at the top of audit contests in Code4rena and Sherlock recently.

Disclaimer

Smart contracts are an experimental technology with many known and unknown risks. Trust Security assumes no responsibility for any misbehavior, bugs or exploits affecting the audited code or any part of the deployment phase.

Furthermore, it is known to all parties that changes to the audited code, including fixes of issues highlighted in this report, may introduce new issues and require further auditing.

Methodology

In general, the primary methodology used is manual auditing. The entire in-scope code has been deeply looked at and considered from different adversarial perspectives. Any additional dependencies on external code have also been reviewed.

Qualitative analysis

Metric	Rating	Comments
Code complexity	**Good**	Project kept code as simple as possible, reducing attack risks
Documentation	**Good**	Project is mostly very well documented.
Best practices	**Good**	Project consistently adheres to industry standards.
Centralization risks	**Moderate**	Project relies on admin to set correct parameters. A compromised admin account could risk the safety of funds.

Findings

High severity findings

TRST-H-1 Underflow issue in _distributeReward() will prevent users from unstaking assets

- **Category:** Underflow issues
- **Source:** NodeController.sol
- **Status:** Open

Description

NodeController._distributeReward() distributes the rewards generated by *PublicDelegation* to *StakingToken*, it takes the **current rewards** by subtracting the **total staked assets** and subtracting the **previously accumulated rewards** from the **total claimable assets** of all the *NodeServices*. Then it updates the **accumulated rewards**, at this point **_totalKaiaAmount + _totalUnstakingAmount = _totalStakingAmount + _lastNativeTokenRewardAmount**.

```
    function _distributeReward() private whenNotPaused {
        uint256 _totalKaiaAmount = getTotalClaimable();
        uint256 _totalStakingAmount = totalStakingAmount;
        uint256 _totalUnstakingAmount = totalUnstakingAmount;
        uint256 _lastNativeTokenRewardAmount = lastNativeTokenRewardAmount;

        //Total - (staked + sum of previous interest - unstaked) => current interest
        uint256 currentReward = _totalKaiaAmount + _totalUnstakingAmount -
_totalStakingAmount - _lastNativeTokenRewardAmount;

        lastNativeTokenRewardAmount = _totalKaiaAmount + _totalUnstakingAmount -
_totalStakingAmount;

        if (currentReward > 0) {
            totalReward = totalReward.add(currentReward);
        }

        IStakingToken(stakingTokenAddress).changeRatio(lastNativeTokenRewardAmount,
currentReward);

        distributeRewardCount = distributeRewardCount.add(1);
        emit Reward(distributeRewardCount, _totalKaiaAmount, _totalStakingAmount,
_totalUnstakingAmount, _lastNativeTokenRewardAmount, currentReward);
    }
```

The problem is that the *_staking()*/*_unstaking()* calls that follow *_distributeReward()* will mint or burn shares via *PublicDelegation*, and because of the rounding, it will cause:

_totalKaiaAmount + _totalUnstakingAmount < _totalStakingAmount + _lastNativeTokenRewardAmount

That leads to *_distributeReward()* reverting due to underflow, and will revert the subsequent stake/unstake, until *PublicDelegation* accumulates more rewards. A malicious user can call *stake(1 wei)* after *PublicDelegation* generates rewards, thus preventing other users from unstaking their funds, thus freezing the user's assets.

```
Consider in PD A, totalSupply = 1000, totalAsset = 1000,
```

```
NodeService(NS A) has 900 shares, and other users have 100 shares.

In PD B, totalSupply = 1000, totalAsset = 1000,
NodeService(NS B) has 900 shares, and other users have 100 shares.

Afterwards, rewards are generated
PD A: totalSupply = 1000, totalAsset = 1100.
PD B: totalSupply = 1000, totalAsset = 1300.

ratio = 1.2

1.  unstake example:
NS user unstake(100 shares, PD A), assets = 120.

first call _distributeReward(), _totalKaiaAmount = 990 + 1170 = 2160.
_totalKaiaAmount + _totalUnstakingAmount == _totalStakingAmount +
lastNativeTokenRewardAmount

In PD A.withdraw(), previewWithdraw(120 assets) = 110 shares (rounding up)
burn 110 shares, PD A: totalSupply = 890, totalAsset = 980, NS has 790 shares

Then other operations call _distributeReward(), _totalKaiaAmount = 1170 + 790*980/890
= 1170 + 869 = 2039, _totalKaiaAmount decreases by 121 , but _totalUnstakingAmount
only increases by 120, _totalKaiaAmount + _totalUnstakingAmount < _totalStakingAmount
+ lastNativeTokenRewardAmount, it will underflow.

2.  stake example:
NS user stake(120 assets, PD A), shares = 100.

first call _distributeReward, _totalKaiaAmount = 990 + 1170 = 2160.
_totalKaiaAmount + _totalUnstakingAmount == _totalStakingAmount +
lastNativeTokenRewardAmount

In PD A.stake(), _convertToShares(120 assets) = 109 shares (rounding down)
mint 109 shares, PD A: totalSupply = 1109, totalAsset = 1220, NS has 1009 shares

Then other operations call _distributeReward(), _totalKaiaAmount = 1170 +
1009*1220/1109 = 1170 + 1109 = 2279, _totalKaiaAmount increases by 119 , but
_totalStakingAmount increases by 120, _totalKaiaAmount + _totalUnstakingAmount <
_totalStakingAmount + lastNativeTokenRewardAmount, it will underflow
```

Recommended mitigation

It is recommended to check for underflow before calculating **currentReward**.

```
    function _distributeReward() private whenNotPaused {
        uint256 _totalKaiaAmount = getTotalClaimable();
        uint256 _totalStakingAmount = totalStakingAmount;
        uint256 _totalUnstakingAmount = totalUnstakingAmount;
        uint256 _lastNativeTokenRewardAmount = lastNativeTokenRewardAmount;
+       if( _totalKaiaAmount + _totalUnstakingAmount > _totalStakingAmount +
_lastNativeTokenRewardAmount) {
            //Total - (staked + sum of previous interest - unstaked) => current
interest
            uint256 currentReward = _totalKaiaAmount + _totalUnstakingAmount -
_totalStakingAmount - _lastNativeTokenRewardAmount;

            lastNativeTokenRewardAmount = _totalKaiaAmount + _totalUnstakingAmount -
_totalStakingAmount;
        }
+       else { uint256 currentReward = 0 }
```

TRST-H-2 The first staker can steal the subsequent user's assets through the inflation attack

- **Category:** Inflation attacks
- **Source:** StakingToken.sol
- **Status:** Open

Description

Neither *StakingToken* nor *PublicDelegation* has defenses against inflation attack, which makes the first staker can steal the subsequent user's assets through the inflation attack.

```
Consider that StakingToken has just been deployed, and it has an also newly deployed
PublicDelegation as node.

Alice, as the first staker, stakes 1 wei native token to StakingToken, with the ratio
of 1e18, and mints 1 wei share to Alice.

Alice sends 1e18 wei native tokens to PublicDelegation and calls
NodeController.distributeReward(), at this time _totalKaiaAmount is 1e18 + 1, and 1e18
wei native tokens will be distributed as rewards and ratio will be updated as (1e18)
+1) * 1e18 / 1 = 1e36 + 1e18.

When Bob stakes 2e18 wei native tokens, Bob receives 2e18*1e18/(1e36 + 1e18) = 1 wei
share (rounding down).

At this point, the total assets are 2e18+1e18+1 native tokens, and Bob's 1 share is
worth only 1.5e18 native tokens, resulting in Alice stealing 0.5e18 native tokens.
```

Recommended mitigation

It is recommended to use decimals offset to defend against the inflation attack.

TRST-H-3 Pools will suffer reduced reward rates due to staking and unstaking mismatch

- **Category:** Logical issues
- **Source:** StakingToken.sol
- **Status:** Open

Description

In *StakingToken*, users can choose any node to stake or unstake.

```
    function unstake(address stakeNode, uint256 stakingTokenAmount) public
whenNotPaused validAddress(stakeNode) validMoreThanZero(stakingTokenAmount)
nonReentrant {
        _unstake(stakeNode, _msgSender(), stakingTokenAmount);
    }
```

The problem here is that the node that the user unstakes can be different from the node that they stake. Malicious users can select nodes with lower reward rates to stake, and then select nodes with higher reward rates to unstake, thus lowering the reward rate of the protocol.

Recommended mitigation

It is recommended to keep track of the shares users get when they stake to any node, and only allow users to unstake from the nodes they staked, for the amount of shares staked.

It is worth noting that users are only allowed to unstake from other nodes if the *totalSupply()* of staked nodes is 0.

TRST-H-4 Using the same index for claim in NodeService and NodeController will make users unable to claim their assets

- **Category:** Logical issues
- **Source:** NodeService.sol, NodeController.sol
- **Status:** Open

Description

Suppose there are two *NodeServices*. One user made 3 unstake requests in the first *NodeService* and 2 requests in the second. The length of **userUnstakeInfo** for that user in the first *NodeService* is 3, and in the second *NodeService*, it is 2.

```
function _unstake(address account, uint256 amount) private returns (uint256) {
    uint256 count = node.getUserRequestCount(address(this));
    uint256 requestId = node.userRequestIds(address(this), count - 1);
    uint256 userRequestCount = userUnstakeInfo[account].length;
    userUnstakeInfo[account].push(Unstake.UnstakeInfo(userRequestCount, requestId,
address(this), amount, block.timestamp, 0, State.ChangeState.Unstaked)); // 3 in the
first
    return requestId;
}
```

However, there is only one *NodeController*, and the length of **unstakeInfos** for that user in the *NodeController* is 5.

```
function _unstake(address payable stakeNode, address account, uint256 amount) private
whenNotPaused validAddress(stakeNode) validAddress(account) validMoreThanZero(amount)
returns (uint256) {
    uint256 requestId = INodeService(nodeInfo.node).unstake(account, amount);
    unstakeInfos[account].push(Unstake.UnstakeInfo(unstakeInfos[account].length,
requestId, stakeNode, amount, block.timestamp, 0, State.ChangeState.Unstaked)); // 5
    return requestId;
}
```

At this point, it becomes impossible to claim the last two requests.

For example, if the last request is for the first *NodeService*, the *_claim()* function of the first *NodeService* should be called and the **index** parameter is 4. However, this **index** exceeds the length of **userUnstakeInfo**, and the user won't be able to claim.

Recommended mitigation

Modify the *_unstake()*, *_claim()* and *getUnstakeRequestAmount()* functions in the *NodeController* as follows:

```
function _unstake(address payable stakeNode, address account, uint256 amount) private
whenNotPaused validAddress(stakeNode) validAddress(account) validMoreThanZero(amount)
returns (uint256) {
uint256 requestId = INodeService(nodeInfo.node).unstake(account, amount);
```

```
+    uint256 length =
INodeService(nodeInfo.node).getUnstakeRequestInfoLength(account);
-    unstakeInfos[account].push(Unstake.UnstakeInfo(unstakeInfos[account].length,
requestId, stakeNode, amount, block.timestamp, 0, State.ChangeState.Unstaked));
+    unstakeInfos[account].push(Unstake.UnstakeInfo(length - 1, requestId, stakeNode,
amount, block.timestamp, 0, State.ChangeState.Unstaked));
    return requestId;
}
…
function _claim(address account, uint256 index) private whenNotPaused
validAddress(account) validMoreThanEqualZero(index) returns (State.ChangeState) {
    require(unstakeInfos[account].length > index, "NodeController:: unstake request
not exists");
    Unstake.UnstakeInfo memory unstakeRequest = unstakeInfos[account][index];
    address stakeNode = unstakeRequest.unstakeNode;
    require(stakeNode != address(0), "NodeController:: unstake request not exists");
    emit Claimed(unstakeRequest.unstakeNode, account, index);
-    State.ChangeState changeState = INodeService(payable(stakeNode)).claim(account,
index);
+    State.ChangeState changeState = INodeService(payable(stakeNode)).claim(account,
unstakeRequest.index);
    return changeState;
}
…
function getUnstakeRequestAmount(address account, uint256 index) public view
validAddress(account) validMoreThanEqualZero(index) returns (uint256) {
    Unstake.UnstakeInfo memory unstakeRequest = unstakeInfos[account][index];
-    return
INodeService(payable(unstakeRequest.unstakeNode)).getUnstakeRequestAmount(account,
index);
+    return
INodeService(payable(unstakeRequest.unstakeNode)).getUnstakeRequestAmount(account,
unstakeRequest.index);
}
```

TRST-H-5 Incorrect calculation of ratio in StakingToken leads to user loss of stake

- **Category:** Accounting errors
- **Source:** StakingToken.sol
- **Status:** Open

Description

Whenever there is a reward in the *PublicDelegation* and these rewards are distributed in the *NodeController*, the *changeRatio()* function of the *StakingToken* is called.

```
function _distributeReward() private whenNotPaused {
    uint256 currentReward = _totalKaiaAmount + _totalUnstakingAmount -
_totalStakingAmount - _lastNativeTokenRewardAmount;

    lastNativeTokenRewardAmount = _totalKaiaAmount + _totalUnstakingAmount -
_totalStakingAmount;
    if (currentReward > 0) {
        totalReward = totalReward.add(currentReward);
    }
    IStakingToken(stakingTokenAddress).changeRatio(lastNativeTokenRewardAmount,
currentReward); // here
}
```

In this function, the new **ratio** is calculated, and new stakers will mint shares according to this new ratio.

However, the problem is that the rewards are being added twice, which incorrectly increases the ratio.

For example, if users stake 1000 tokens in the *StakingToken*, the total supply is also 1000. When there is a reward of 200 tokens, the ratio is calculated as 1400/1000 because the 200 rewards are added twice.

```
function changeRatio(uint256 totalRemainReward, uint256 currentReward) public
onlyNodeController { // totalRemainReward = 200, currentReward = 200
    uint256 fee = 0;
    uint256 finalReward = 0;
    if (currentReward > 0) {
        fee = currentReward.mul(feeRatio).div(100);
        finalReward = currentReward.sub(fee);
        uint256 currentRewardStakingToken = getRatioStakingTokenByNativeToken(fee);
        _mint(treasuryAddress, currentRewardStakingToken);
        totalReward = totalReward.add(finalReward);
        totalStaking = totalStaking.add(currentReward); // 1000 + 200 = 1200
    }
    ratio = (totalStaking.add(totalRemainReward)).mul(1 * 10 **
18).div(totalSupply()); // (1200 + 200) / 1000
}
```

The correct ratio is clearly 1200/1000. As a result, new stakers experience a loss due to this miscalculated ratio.

Recommended mitigation

Add the reward to the **totalStaking** only once.

```
function changeRatio(uint256 totalRemainReward, uint256 currentReward) public
onlyNodeController {
    uint256 fee = 0;
    uint256 finalReward = 0;
    if (currentReward > 0) {
        fee = currentReward.mul(feeRatio).div(100);
        finalReward = currentReward.sub(fee);
        uint256 currentRewardStakingToken = getRatioStakingTokenByNativeToken(fee);
        _mint(treasuryAddress, currentRewardStakingToken);
        totalReward = totalReward.add(finalReward);
        totalStaking = totalStaking.add(currentReward);
    }
-    ratio = (totalStaking.add(totalRemainReward)).mul(1 * 10 **
18).div(totalSupply());
+    ratio = totalStaking.mul(1 * 10 ** 18).div(totalSupply());
}
```

TRST-H-6 Miscalculation in NodeController._claim() causes rewards to be overstated

- **Category:** Logical issues
- **Source:** NodeController.sol
- **Status:** Open

Description

Consider the following example:

Users stake 1000 tokens to the *PublicDelegation*, and a staker made a request to unstake 200 tokens.
The following specific values are used in the *_distributeReward()* function.

```
_totalKaiaAmount = 800;
_totalStakingAmount = 1000;
_totalUnstakingAmount = 200;
lastNativeTokenRewardAmount = 0;
```

There were 200 rewards in the *PublicDelegation*.

```
_totalKaiaAmount = 1000;
_totalStakingAmount = 1000;
_totalUnstakingAmount = 200;
lastNativeTokenRewardAmount = 200;
```

That staker finally claims their 200 tokens.
If the unstake request was expired and canceled, those 200 tokens are staked again in the *_claim()* function of the *PublicDelegation*.

```
function _claim(uint256 _id) private {
    require(requestIdToOwner[_id] == _msgSender(), "Not the owner of the request.");
    baseCnStakingV3.withdrawApprovedStaking(_id);
    (, uint256 _asset, , ICnStakingV3.WithdrawalStakingState _state) =
_withdrawalRequest(_id);
    if (_state == ICnStakingV3.WithdrawalStakingState.Canceled) {
        uint256 _shares = _convertToShares(_asset, totalSupply(), _totalAssets() -
_asset);
        _mint(_msgSender(), _shares);  // here
        return;
    }
}
```

As a result, **_totalKaiaAmount** increases to 1200, but it calls the *_distributeReward()* function before decreasing the **totalUnstakingAmount** to 0.

```
function _claim(address account, uint256 index) private whenNotPaused
validAddress(account) validMoreThanEqualZero(index) returns (State.ChangeState) {
    State.ChangeState changeState = INodeService(payable(stakeNode)).claim(account,
index);
    if (changeState == State.ChangeState.Canceled) {
        _distributeReward();  // here
        totalUnstakingAmount = totalUnstakingAmount.sub(unstakeRequest.amount);
        unstakeInfos[account][index].state = State.ChangeState.Canceled;
    } else if (changeState == State.ChangeState.Claimed) {
    }
    return changeState;
}
```

The values are now as follows:

```
_totalKaiaAmount = 1200;
_totalStakingAmount = 1000;
_totalUnstakingAmount = 200;
lastNativeTokenRewardAmount = 400;
```

The first impact is that the rewards(**lastNativeTokenRewardAmount**) are wrongly calculated as 400, causing future transactions to be reverted until an additional 200 rewards are generated. The second impact is that the ratio is incorrectly calculated in the *StakingToken* because the 200 rewards are accidently added.

Recommended mitigation

Update **totalUnstakingAmount** before distributing rewards.

```
function _claim(address account, uint256 index) private whenNotPaused
validAddress(account) validMoreThanEqualZero(index) returns (State.ChangeState) {
    State.ChangeState changeState = INodeService(payable(stakeNode)).claim(account,
index);
    if (changeState == State.ChangeState.Canceled) {
-         _distributeReward();
        totalUnstakingAmount = totalUnstakingAmount.sub(unstakeRequest.amount);
+         _distributeReward();
        unstakeInfos[account][index].state = State.ChangeState.Canceled;
    } else if (changeState == State.ChangeState.Claimed) {
    }
    return changeState;
}
```

Medium severity findings

TRST-M-1 setActiveNode() does not set nodeInfos.isActive, making the isActive setting not work

- **Category:** Logical issues
- **Source:** NodeController.sol
- **Status:** Open

Description

The *setActiveNode()* function only sets **nodes.isActive** but does not set **nodeInfos.isActive**.

```
    function setActiveNode(address payable nodeAddress, bool isActive) public
onlyRole(NODE_MANAGER) validAddress(nodeAddress) {
        require(nodeInfos[nodeAddress].node != address(0), "NodeController:: node not
exists");

        nodes[nodeInfos[nodeAddress].index].isActive = isActive;

        emit NodeChange(nodeInfos[nodeAddress].index, nodeAddress, isActive);
    }
```

This makes it impossible for managers to enable or disable stake.

```
    function _stake(address payable stakeNode, address account, uint256 amount)
private whenNotPaused validAddress(stakeNode) validAddress(account)
validMoreThanZero(amount) {
        Node.NodeInfo memory nodeInfo = nodeInfos[stakeNode];

        require(nodeInfo.isActive, "NodeController:: node is not active");

        totalStakingAmount = totalStakingAmount.add(msg.value);
        INodeService(stakeNode).stake{value: amount}(account);

        emit Staked(stakeNode, account, amount);
    }
```

Recommended mitigation

It is recommended to change as follows.

```
    function setActiveNode(address payable nodeAddress, bool isActive) public
onlyRole(NODE_MANAGER) validAddress(nodeAddress) {
        require(nodeInfos[nodeAddress].node != address(0), "NodeController:: node not
exists");

+       nodeInfos[nodeAddress].isActive = isActive;
        nodes[nodeInfos[nodeAddress].index].isActive = isActive;

        emit NodeChange(nodeInfos[nodeAddress].index, nodeAddress, isActive);
    }
```

TRST-M-2 The fee in StakingToken will be higher than expected

- **Category:** Accounting errors
- **Source:** StakingToken.sol

- **Status:** Open

Description

Since fees are charged before updating the ratio in *StakingToken*, the fees charged will be higher than expected.

```
Consider totalSupply = 1000, totalStaking = 1000, ratio = 1e18.
call changeRatio(0,100)
fee = 100 * 10 / 100 = 10.
getRatioStakingTokenByNativeToken(10) = 10
ratio = 1100 / 1010 = 1.089.
10 share worth 10.89 assets, instead of 10.
```

Recommended mitigation

It is recommended to first update the ratio in *changeRatio()* and then convert the fee to share.

TRST-M-3 StakingToken mints or burns shares before updating the ratio, making the shares/assets conversion stale

- **Category:** Logical issues
- **Source:** StakingToken.sol
- **Status:** Open

Description

StakingToken should use the latest ratio to convert between assets and shares.

Consider that *NodeController.distributeReward()* is not called frequently to update the ratio. When the user calls *StakingToken.stake()*, it will first mint shares in *_plusStaking()* based on the stale ratio, and then call *NodeController.distributeReward()* in *NodeController.stake()* to update the ratio.

```
    function _stake(address stakeNode, address account, uint256 nativeTokenAmount)
private whenNotPaused validAddress(stakeNode) validAddress(account)
validMoreThanZero(nativeTokenAmount) {
        uint256 stakingToken = getRatioStakingTokenByNativeToken(nativeTokenAmount);

        _plusStaking(account, stakingToken, nativeTokenAmount);
        INodeController(nodeControllerAddress).stake{value:
nativeTokenAmount}(stakeNode, account);
    }
```

This is also true in *StakingToken.unstake()*, only in *StakingToken.claim()* will call *_plusStaking()* after *NodeController.claim()* to mint shares with the latest ratio.

This results in users getting more shares when staking and less assets when unstaking.

Recommended mitigation

It is recommended to call *NodeController.distributeReward()* at the beginning of *StakingToken.stake()/unstake()* to make the ratio fresh.

```
    function _stake(address stakeNode, address account, uint256 nativeTokenAmount)
private whenNotPaused validAddress(stakeNode) validAddress(account)
validMoreThanZero(nativeTokenAmount) {
+       INodeController(nodeControllerAddress).distributeReward();
        uint256 stakingToken = getRatioStakingTokenByNativeToken(nativeTokenAmount);

        _plusStaking(account, stakingToken, nativeTokenAmount);
        INodeController(nodeControllerAddress).stake{value:
nativeTokenAmount}(stakeNode, account);
    }
    ...
    function _unstake(address nodeAddress, address account, uint256
stakingTokenAmount) private whenNotPaused validAddress(nodeAddress)
validAddress(account) validMoreThanZero(stakingTokenAmount) {
+       INodeController(nodeControllerAddress).distributeReward();
        uint256 nativeTokenAmount =
getRatioNativeTokenByStakingToken(stakingTokenAmount);

        _minusStaking(account, stakingTokenAmount, nativeTokenAmount);
        INodeController(nodeControllerAddress).unstake(nodeAddress, account,
nativeTokenAmount);
    }
```

TRST-M-4 Incorrect Treasury implementation causing the treasury assets to be frozen

- **Category:** Logical issues
- **Source:** Treasury.sol
- **Status:** Open

Description

The *Treasury* below is currently in scope.

```
contract Treasury is Initializable, OwnableUpgradeable, ReentrancyGuardUpgradeable
{
    function initialize(address _owner) public initializer {
        __Ownable_init(_owner);
        __ReentrancyGuard_init();
    }
}
```

If this *Treasury* is used as the **treasury** in the *StakingToken*, fees are minted to this *Treasury*.

```
function changeRatio(uint256 totalRemainReward, uint256 currentReward) public
onlyNodeController {
    uint256 fee = 0;
    uint256 finalReward = 0;
    if (currentReward > 0) {
        fee = currentReward.mul(feeRatio).div(100);
        finalReward = currentReward.sub(fee);
        uint256 currentRewardStakingToken = getRatioStakingTokenByNativeToken(fee);
        _mint(treasuryAddress, currentRewardStakingToken);  // here
        totalReward = totalReward.add(finalReward);
        totalStaking = totalStaking.add(currentReward);
    }
}
```

However, there is currently no method to claim these fees from the *Treasury*.

Recommended mitigation

It is recommended to add functions to unstake and claim tokens from the *StakingToken*.

TRST-M-5 The unstake request can be reverted in some cases

- **Category:** Logical issues
- **Source:** StakingToken.sol
- **Status:** Open

Description

When stakers attempt to unstake their shares from the *PublicDelegation*, the transaction can be reverted if there are insufficient tokens in that specific *PublicDelegation*.

```
function _unstake(address payable stakeNode, address account, uint256 amount) private
whenNotPaused validAddress(stakeNode) validAddress(account) validMoreThanZero(amount)
returns (uint256) {
    Node.NodeInfo memory nodeInfo = nodeInfos[stakeNode];
    require(amount <= INodeService(nodeInfo.node).getTotalKaiaAmount(),
"NodeController:: not enough staking amount");  // here
    totalUnstakingAmount = totalUnstakingAmount.add(amount);
    uint256 requestId = INodeService(nodeInfo.node).unstake(account, amount);
    unstakeInfos[account].push(Unstake.UnstakeInfo(unstakeInfos[account].length,
requestId, stakeNode, amount, block.timestamp, 0, State.ChangeState.Unstaked));
    emit Unstaked(stakeNode, account, amount);
    return requestId;
}
```

For example, consider two *PublicDelegations*:
User A stakes 100 tokens in the first *PublicDelegation* and user B stakes 100 tokens in the second *PublicDelegation*.
They will then get 100 shares respectively.
After some time, there are 30 rewards in the first *PublicDelegation* and 10 rewards in the second *PublicDelegation*, making the total staking 240 tokens.
If user B wants to unstake his 100 shares and selects the second *PublicDelegation* where they initially staked, but the second *PublicDelegation* only has 110 tokens available and the requested unstake amount is 120 tokens, the transaction will fail and be reverted.
In such case, user B should split their unstake request between both *PublicDelegations*.

Recommended mitigation

In the *StakingToken*, the user's unstake request can choose the minimum between the requested amount and the current available token amounts in the selected *PublicDelegation*. This approach allows user B to unstake 110 tokens from the second *PublicDelegation*, ensuring the shares equivalent to 10 tokens will remain in the first *PublicDelegation* in the above example.

TRST-M-6 StakingToken approval cannot be reset to 0

- **Category:** Compatibility issues
- **Source:** StakingToken.sol
- **Status:** Open

Description

The *approve()* function of the *StakingToken* uses the *validMoreThanZero()* modifier.

```
function approve(address spender, uint256 amount) public override(IERC20,
ERC20Upgradeable) whenNotPaused validAddress(spender) validMoreThanZero(amount)
returns (bool) {  // here
    return super.approve(spender, amount);
}
```

This means that attempting to approve a 0 amount will be reverted, this make the StakingToken behave differently from standard ERC20 tokens.

Recommended mitigation

It is recommended to use *validMoreThanEqualZero()* modifier.

Low severity findings

TRST-L-1 NodeService is not pausable despite the intention

- **Category:** Logical issues
- **Source:** NodeService.sol
- **Status:** Open

Description

The *NodeService* inherits from the *PausableUpgradeable* contract, but does not implement the *pause()* and *unpause()* functions, which makes the *whenNotPaused* modifier not work.

Recommended mitigation

It is recommended to implement functions to call the *_pause()* and *_unpause()* functions.

TRST-L-2 The fees can be changed retroactively by the staking manager

- **Category:** Logical issues
- **Source:** StakingToken.sol
- **Status:** Open

Description

When the **feeRatio** is changed, the current accumulated rewards are not distributed.

```
function setFeeRatio(uint256 _feeRatio) public onlyRole(STAKING_MANAGER) {
    feeRatio = _feeRatio;
}
```

Consequently, the next minting of fees will use the new **feeRatio**. If the **feeRatio** increases, then previous rewards will be distributed with the larger **feeRatio**, making the actual rewards less than expected.

Recommended mitigation

It is recommended to call *distributeReward()* function of the *NodeController* before changing the **feeRatio**.

TRST-L-3 The tracking of totalUnstakingAmount and userUnstakingAmount values in the NodeService is incorrect

- **Category:** Logical issues
- **Source:** NodeService.sol
- **Status:** Open

Description

In the *NodeService*, the **totalUnstakingAmount** and **userUnstakingAmount** should not only increase when users make an unstake request but also decrease when unstake requests are canceled, similar to how it is managed in the *NodeController*.

Recommended mitigation

It is recommended to decrease these values when the unstake requests are canceled in the *NodeService*.

TRST-L-4 NodeService.getUnstakeStatus() return incorrect data when the index exceeds the length of userUnstakeInfo

- **Category:** Logical issues
- **Source:** NodeService.sol
- **Status:** Open

Description

When the index exceeds the length of **userUnstakeInfo**, the *getUnstakeStatus()* function currently returns the unstake status of the request with ID 0 instead of reverting. This could introduce integration issues with other contracts.

```
function getUnstakeStatus(address account, uint256 index) public view returns
(IPublicDelegation.WithdrawalRequestState) {
    Unstake.UnstakeInfo memory unstakeInfo = getUnstakeRequestInfo(account, index);
    return
IPublicDelegation(publicDelegationAddress).getCurrentWithdrawalRequestState(unstakeInf
o.unstakeId);  // here
}
```

Recommended mitigation

It is recommended to change as follows.

```
function getUnstakeStatus(address account, uint256 index) public view returns
(IPublicDelegation.WithdrawalRequestState) {
    Unstake.UnstakeInfo memory unstakeInfo = getUnstakeRequestInfo(account, index);
+    if (unstakeInfo.unstakeTime == 0) revert();
    return
IPublicDelegation(publicDelegationAddress).getCurrentWithdrawalRequestState(unstakeInf
o.unstakeId);
}
```

Additional recommendations

TRST-R-1 StakingToken should override the _update() function

StakingToken inherits from *ERC20Upgradeable* and adds the *whenNotPaused* modifier to the *_mint(), _burn(), approve(), claim()* functions, but does not restrict token transfers.

It is recommended to override *ERC20Upgradeable._update()* to add the *whenNotPaused* modifier.

TRST-R-2 addNode() does not need to set index again

addNode() does not need to set **index** again.

```
    function addNode(address payable nodeAddress, bool isActive) public
onlyRole(NODE_MANAGER) validAddress(nodeAddress) {
        require(nodeInfos[nodeAddress].node == address(0), "NodeController:: node
already exists");

        nodeInfos[nodeAddress] = Node.NodeInfo(nodes.length, nodeAddress, isActive);
-       nodeInfos[nodeAddress].index = nodes.length; //@audit: can remove it
        nodes.push(nodeInfos[nodeAddress]);

        emit NodeAdded(nodeInfos[nodeAddress].index, nodeAddress, isActive);
    }
```

Also *NodeService._claim()* does not need to set the state again

```
function _claim(address account, uint256 index) private returns (State.ChangeState) {
    ...
    if (state == IPublicDelegation.WithdrawalRequestState.Withdrawn) {
-       userUnstakeInfo[account][index].state = State.ChangeState.Claimed;  // @audit:
can remove this
        userClaimInfo[account].push(Claim.ClaimInfo(length, unstakeInfo.unstakeId,
block.timestamp, unstakeInfo.amount, State.ClaimState.Claimed));

        require(address(this).balance >= unstakeInfo.amount, "NodeController:: not
enough balance : ");
        (bool success,) = account.call{value: unstakeInfo.amount}("");
        require(success, "NodeController:: transfer failed : ");

        claimResult = State.ChangeState.Claimed;
        userUnstakeInfo[account][index].claimTime = block.timestamp;
        userUnstakeInfo[account][index].state = claimResult;
    } else if (state == IPublicDelegation.WithdrawalRequestState.Canceled) {
-       userUnstakeInfo[account][index].state = State.ChangeState.Canceled;  //
@audit: can remove this
        userClaimInfo[account].push(Claim.ClaimInfo(length, unstakeInfo.unstakeId,
block.timestamp, unstakeInfo.amount, State.ClaimState.Canceled));

        claimResult = State.ChangeState.Canceled;
        userUnstakeInfo[account][index].state = claimResult;
    }
    emit ChangedStake(claimResult, address(this), unstakeInfo.unstakeId, account,
unstakeInfo.amount);

    return claimResult;
```

```
}
```

TRST-R-3 NodeController and StakingToken do not implement the removeRole() function

NodeController and *StakingToken* call *_grantRole()* through the *addRole()* function, but do not implement a function to call *_revokeRole()*. It is recommended that *NodeController* and *StakingToken* implement a function to call *_revokeRole()*, or directly use the *grantRole()* and *revokeRole()* functions implemented by *AccessControlUpgradeable*.

TRST-R-4 There should be a single function to update bugholeNode in the NodeService and NodeController

Currently, both the *NodeController* and all *NodeServices* have their own **bugholeNode** variable and function to update it. It is recommended that *NodeServices* fetch **bugholeNode** from the *NodeController*. This would consolidate the update function for **bugholeNode** across the protocol.

TRST-R-5 Some variables in the NodeService that are currently unused

In the *NodeService*, variables such as **totalTreasuryAmount** and **userClaimCount** are not used. It is recommended to remove them.

```
uint256 private totalTreasuryAmount;
mapping(address => uint256) private userClaimCount;
```

TRST-R-6 FeeRatio should have an upper limit

In the *setFeeRatio()* function, any value can currently be set as **feeRatio**.

```
function setFeeRatio(uint256 _feeRatio) public onlyRole(STAKING_MANAGER) {
    feeRatio = _feeRatio;
}
```

However, it should be ensured that the **feeRatio** is less than 100, or more appropriately, a sensible upper limit to the fees the protocol would charge.

Centralization risks

CR-1 The admin has the ability to change the PublicDelegation at any time

In the *NodeService*, the admin can change the *PublicDelegation* without considering whether there are staked tokens or claimable tokens in the old *PublicDelegation*.

If there are tokens staked or claimable in the old *PublicDelegation* that is replaced with a new one, those tokens won't be claimable by stakers anymore.

CR-2 The admin has the ability to change the NodeController at any time

In the *StakingToken*, the admin can change the *NodeController* without considering whether the total supply is 0.

Non-zero total supply means that some tokens are staked to this *NodeController* and if it is replaced with a new one, old stakers won't be able to use their staked tokens anymore.

Systemic risks

TRST-SR-1 PublicDelegation is trusted

PublicDelegation owner can set **commissionTo** to any address. If the *PublicDelegation* owner sets **commissionTo** to an address that rejects receiving native tokens, then *_sweepAndStake()* will revert and *StakingToken.stake()/unstake()/claim()* will also not work.