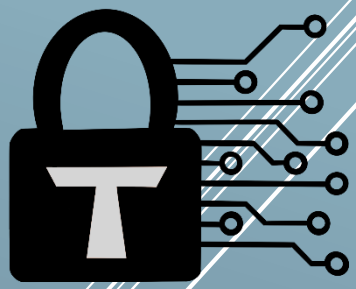
Trust Security

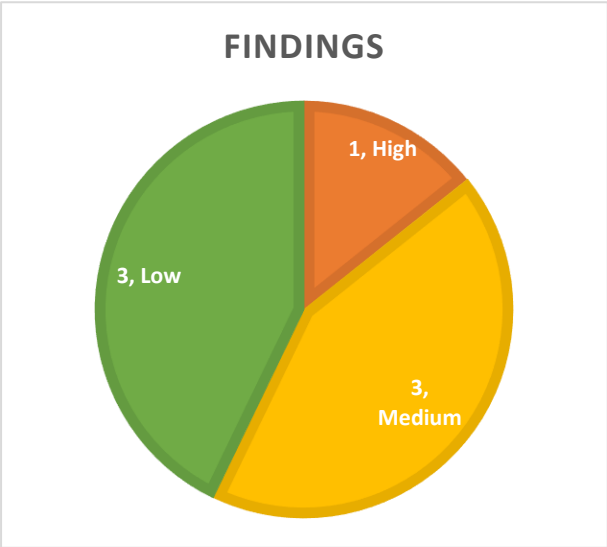


Smart Contract Audit

LinkPool Native Withdrawals

04/02/25

Executive summary

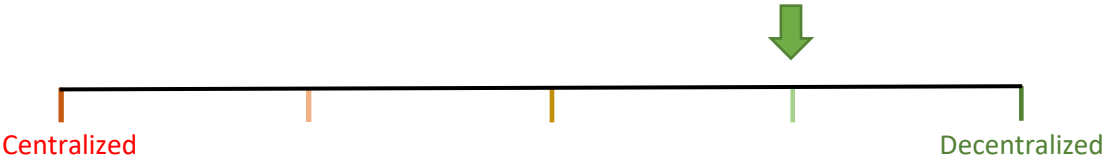


Category	Liquid Staking
Audited file count	11
Lines of Code	2623
Auditor	cccz, SpicyMeatball
Time period	13/01/2025-27/01/2025

Findings

Severity	Total	Fixed	Acknowledged
High	1	1	-
Medium	3	1	2
Low	3	2	1

Centralization score



Signature

Digitally signed by Trust
Date: 2025.02.04 19:11:47 Z00'00'

Document properties

Versioning

Version	Date	Description
0.1	27/01/2025	Client report
0.2	01/02/2025	Mitigation review
0.3	04/02/2025	Mitigation review #2
0.4	04/02/2025	Update TRST-M-3 response

Contact

Trust

trust@trust-security.xyz

Introduction

Trust Security has conducted an audit at the customer's request. The audit is focused on uncovering security issues and additional bugs contained in the code defined in scope. Some additional recommendations have also been given when appropriate.

Scope

The following files are in scope of the audit:

- ./contracts/core/priorityPool/PriorityPool.sol
- ./contracts/core/priorityPool/WithdrawalPool.sol
- ./contracts/linkStaking/base/VaultControllerStrategy.sol
- ./contracts/linkStaking/FundFlowController.sol
- ./contracts/core/StakingPool.sol
- ./contracts/core/RebaseController.sol
- ./contracts/linkStaking/OperatorVCS.sol
- ./contracts/linkStaking/CommunityVCS.sol
- ./contracts/linkStaking/base/Vault.sol
- ./contracts/linkStaking/OperatorVault.sol
- ./contracts/linkStaking/CommunityVault.sol

The state of the project at the base commit is assumed safe.

Repository details

- **Repository URL:** https://github.com/stakedotlink/contracts
- **Base commit hash:** 309bae1a21287183688c363d9e33eceb9a3f3a1b
- **Commit hash:** 5f3d2829f86bc74d6b9e805d7e61d9392d6b21b1
- **Mitigation review hash:** 6f83320fc4ebc2ca55624b46e2626c4412180926
- **Mitigation review hash #2:** dab656fe32e08ad1899b0a6c115bb0c0967de177

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.

SpicyMeatball is a member of the Code4rena Zenith and has reported over 100 bugs in various DeFi protocols.

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	**Good**	Project does not introduce significant unnecessary centralization risks.

.

Findings

High severity findings

TRST-H-1 The rewarded LINK tokens may be lost

- **Category:** Logical issues
- **Source:** VaultControllerStrategy.sol, CommunityVCS.sol, OperatorVCS.sol
- **Status:** Fixed

Description

Anyone can call *CommunityVCS.claimRewards()* to claim the reward tokens to *CommunityVCS*, and these can be restaked to *CommunityPool* when depositing.

```
    function claimRewards(
        uint256[] calldata _vaults,
        uint256 _minRewards
    ) external returns (uint256) {
        address receiver = address(this);
        uint256 balanceBefore = token.balanceOf(address(this));
        for (uint256 i = 0; i < _vaults.length; ++i) {
            ICommunityVault(address(vaults[_vaults[i]])).claimRewards(_minRewards,
receiver);
        }
        uint256 balanceAfter = token.balanceOf(address(this));
        return balanceAfter - balanceBefore;
    }
```

CommunityVCS uses **totalDeposits** to track all LINK deposited to *CommunityPool* and uses the staked and rewarded tokens in *CommunityPool* minus **totalDeposits** as rewards.

The problem here is that once the rewards are claimed in *CommunityVCS* and restake to *CommunityPool*, **totalDeposits** will track the deposited rewards, which will make the staked and rewarded tokens in *CommunityPool* minus **totalDeposits** to be 0, resulting in lost rewards.

```
    function deposit(uint256 _amount, bytes calldata _data) external {
        token.safeTransferFrom(msg.sender, address(this), _amount);

        (uint256 minDeposits, uint256 maxDeposits) = getVaultDepositLimits();

        uint256 toDeposit = token.balanceOf(address(this));
        uint64[] memory vaultIds = abi.decode(_data, (uint64[]));

        uint256 deposited = _depositToVaults(toDeposit, minDeposits, maxDeposits,
vaultIds);

        totalDeposits += deposited;
        totalPrincipalDeposits += deposited;

        if (deposited < toDeposit) {
            token.safeTransfer(address(stakingPool), toDeposit - deposited);
        }
```

For example, the protocol stakes 1000 LINK to the *CommunityPool* and generates 200 rewards.

In *_updateStrategyRewards()*, *CommunityVCS.getDepositChange()* returns *getTotalDeposits()* - **totalDeposits** == 1200 - 1000 == 200, **depositChange** is 200, that is, each share will appreciate to 1.2 LINK tokens.

```
    function getDepositChange() public view virtual returns (int) {
        uint256 totalBalance = token.balanceOf(address(this));
        for (uint256 i = 0; i < vaults.length; ++i) {
            totalBalance += vaults[i].getTotalDeposits();
        }
        return int(totalBalance) - int(totalDeposits);
    }
```

But if the attacker call *CommunityVCS.claimRewards()* before *_updateStrategyRewards()*, the 200 LINK reward will be claimed into *CommunityVCS*, then the attacker call *CommunityVCS.deposit()*, the 200 reward tokens will be restaked into the *CommunityPool*, and **totalDeposits** will increase by 200.

In *CommunityVCS.getDepositChange()*, *getTotalDeposits()* will increase by 200 (restaked into the *CommunityPool*), and since **totalDeposits** also increases by 200, this will cause *getDepositChange()* to return 0, which leads to the loss of rewards.

And for *OperatorVCS*, *raiseAlert()* sends alert rewards directly to *OperatorVCS*, which may also be counted in **totalDeposits** *in deposit()*, resulting in lost rewards.

In addition, idle LINK in *CommunityVCS* and *OperatorVCS* can cause inaccurate results of *canDeposit()*.

Recommended mitigation

It is recommended that once rewards are claimed into *CommunityVCS* and *OperatorVCS*, *_updateStrategyRewards()* is called immediately to synchronize the rewards into **totalStaked** to avoid lost rewards.

Team response

Fixed.

Mitigation review

The fix makes the code no longer restake rewarded LINK, so **totalDeposits** no longer tracks rewarded LINK, resolving the issue.

Medium severity findings

TRST-M-1 Vault removal may result in freeze of funds

- **Category:** Logical issues
- **Source:** OperatorVCS.sol
- **Status:** Fixed

Description

After removing the Vault, *OperatorVCS.removeVault()* will reorder the vaults behind the removed vault by moving them forward one by one.

```
        for (uint256 i = 0; i < numVaults; ++i) {
            if (address(vaults[i]) == vault) {
                index = i;
                break;
            }
        }
        for (uint256 i = index; i < numVaults - 1; ++i) {
            vaults[i] = vaults[i + 1];
        }
        vaults.pop();
```

For example, for vaults [a,b,c,d,e,f], after removing c, it will be reordered as [a,b,d,e,f].

Since the protocol uses vault groups to manage vaults, e.g., for 2 vault groups, Group0 is [a,c,e] and Group1 is [b,d,f], after removing c, Group0 will be [a,d,f], and Group1 will be [b,e], which changes the groups for all vaults after c.

In *FundFlowController*, vaults are unbonded in group order to ensure that withdrawals are always available. When removing a vault, the group ID of following vaults is decremented by 1. This causes the next call to *updateVaultGroups()* to call *unbond()* of the Vault that is in the **claimPeriod**, which results in a revert, and actually it should be called 7 days later. This will disrupt the funds control and result in a 7-day freeze of withdrawals.

The POC is as follows:

```
  it('removeVault mess up updateVaultGroups', async () => {
    const {
      accounts,
      strategy,
      stakingPool,
      rewardsController,
      vaults,
      stakingController,
      fundFlowController,
      adrs,
      token,
    } = await loadFixture(deployFixture)

    await stakingPool.deposit(accounts[0], toEther(1000), [encodeVaults([])])
    await rewardsController.setReward(vaults[5], toEther(40))
    await rewardsController.setReward(vaults[6], toEther(100))
    await stakingPool.updateStrategyRewards([0], encode(0))
    await rewardsController.setReward(vaults[5], toEther(50))

    await fundFlowController.updateVaultGroups()
    await time.increase(claimPeriod)
```

```
    await fundFlowController.updateVaultGroups()
    await time.increase(claimPeriod)
    await fundFlowController.updateVaultGroups()
    await time.increase(claimPeriod)
    await fundFlowController.updateVaultGroups()
    await time.increase(claimPeriod)
    await fundFlowController.updateVaultGroups()

    await stakingPool.withdraw(accounts[0], accounts[0], toEther(130),
[encodeVaults([0, 5])])
    await time.increase(claimPeriod)
    await fundFlowController.updateVaultGroups()

    await stakingController.removeOperator(vaults[5])
    await strategy.queueVaultRemoval(5)
    await strategy.removeVault(0)

    await time.increase(claimPeriod)
    await fundFlowController.updateVaultGroups() // revert
  })
```

Recommended mitigation

One option is to leave the removed entry empty in **vaults** in *removeVault()* and prioritize populating them when deploying vaults.

Team response

Fixed.

Mitigation review

The fix implements the recommendations but introduces new issue.

The issue is that vaults that have been removed can be removed again by calling *queueVaultRemoval()*/*removeVault()*, with the impact that user can bypass the reward updater to call *_updateStrategyRewards()* in *removeVault()*.

Team response #2

Fixed.

Mitigation review #2

This fix disabled removing the removed vaults.

TRST-M-2 The *removeStrategy()* function does not work

- **Category:** Logical issues
- **Source:** StakingPool.sol
- **Status:** Acknowledged

Description

The *stakingPool.removeStrategy()* function withdraws **totalDeposits** amount of LINK from the strategy. For *CommunityVCS* and *OperatorVCS*, **totalDeposits** is the total amount of LINK for their five vault groups.

```
    function removeStrategy(
```

```
        uint256 _index,
        bytes memory _strategyUpdateData,
        bytes calldata _strategyWithdrawalData
    ) external onlyOwner {
        require(_index < strategies.length, "Strategy does not exist");

        uint256[] memory idxs = new uint256[](1);
        idxs[0] = _index;
        _updateStrategyRewards(idxs, _strategyUpdateData);

        IStrategy strategy = IStrategy(strategies[_index]);
        uint256 totalStrategyDeposits = strategy.getTotalDeposits();
        if (totalStrategyDeposits > 0) {
            strategy.withdraw(totalStrategyDeposits, _strategyWithdrawalData);
        }
```

However, *VaultDepositController.withdraw()* allows up to **totalUnbonded** (LINK amount in one of the 5 vault groups) to be withdrawn, which makes *stakingPool.removeStrategy()* always throw **InsufficientTokensUnbonded** error.

```
    function withdraw(uint256 _amount, bytes calldata _data) external {
        if (!fundFlowController.claimPeriodActive() || _amount > totalUnbonded)
            revert InsufficientTokensUnbonded();
```

Recommended mitigation

It is recommended to design the *removeStrategy()* function specifically for LINK Staking.

Team response

Acknowledged.

TRST-M-3 Users can time their deposit/withdrawal to get rewards or evade losses

- **Category:** Front-running issues
- **Source:** PriorityPool.sol
- **Status:** Fixed + Acknowledged

Description

There are no constraints on when users can deposit or withdraw LINK from the *PriorityPool*. This leads to several issues:

- A user can frontrun a losing reward update by initiating withdrawal transaction, potentially evading token losses;
- A user can take a LINK flash loan, deposit it into the pool, manually trigger reward update in the *RebaseController*:

```
    function updateRewards(bytes calldata _data) external {
        if (priorityPool.poolStatus() == IPriorityPool.PoolStatus.CLOSED) revert
PoolClosed();
        _updateRewards(_data);
    }
```

Then, they can queue and finalize a withdrawal - all within a single block:

```
    function performUpkeep(bytes calldata _performData) external {
        ---SNIP---


        priorityPool.executeQueuedWithdrawals(toWithdraw, data);
        _finalizeWithdrawals(toWithdraw);
    }
```

Recommended mitigation

Introducing delayed withdrawals on top of the existing queued withdrawal algorithm should address the issue. Specifically, when a user calls *withdraw()* in the *PriorityPool*, their withdrawal request should be saved and could only be queued into the *WithdrawalPool* after a specified delay.

Team response

Fixed.

Mitigation review

The fix added access control to *updateRewards()*, preventing users from diluting rewards by calling *updateRewards()* immediately after depositing. Furthermore, front running reward loss can be mitigated by Automation calling *performUpkeep()* to pause pool.

Still, note that an account holding a substantial amount of LINK could still sandwich the *updateRewards()* call to dilute rewards.

Team response

Acknowledged.

Low severity findings

TRST-L-1 Rounding issues in *StakingPool* lead to burning less shares for users

- **Category:** Rounding issues
- **Source:** StakingPool.sol
- **Status:** Fixed

Description

The *getSharesByStake()* and *getStakeByShares()* functions of stakingPool are used for conversion between shares and assets when transferring assets, and they are always rounded down.

```
    function getSharesByStake(uint256 _amount) public view returns (uint256) {
        uint256 totalStaked = _totalStaked();
        if (totalStaked == 0) {
            return _amount;
        } else {
            return (_amount * totalShares) / totalStaked;
        }
    }
...
    function getStakeByShares(uint256 _amount) public view returns (uint256) {
        if (totalShares == 0) {
            return _amount;
        } else {
            return (_amount * _totalStaked()) / totalShares;
        }
    }
```

The rounding down behavior leads to the following impacts:

1. When **totalStake** < **totalShares**: Unlikely, but in this case, *getStakeByShares(1)* is rounded down to 0, which results in anyone being able to spend 1 wei of another user's shares without approval in *transferSharesFrom()*, and this allows the user to transfer dead shares out of address 0, leading to a potential inflation attack.

```
    function transferSharesFrom(
        address _sender,
        address _recipient,
        uint256 _sharesAmount
    ) external returns (bool) {
        uint256 tokensAmount = getStakeByShares(_sharesAmount);
        _spendAllowance(_sender, msg.sender, tokensAmount);
        _transferShares(_sender, _recipient, _sharesAmount);
        return true;
    }
```

2. When **totalStake** > **totalShares**: This is the normal case, in this case, *getSharesByStake(1)* is rounded down to 0, which results in anyone can request a withdrawal of 1 wei LINK in *PriorityPool* and the protocol burns the user's 0 shares.

```
    function withdraw(
        address _account,
        address _receiver,
        uint256 _amount,
        bytes[] calldata _data
```

```
    ) external onlyPriorityPool {
        uint256 toWithdraw = _amount;
        if (_amount == type(uint256).max) {
            toWithdraw = balanceOf(_account);
        }

        uint256 balance = token.balanceOf(address(this));
        if (toWithdraw > balance) {
            _withdrawLiquidity(toWithdraw - balance, _data);
        }
        require(
            token.balanceOf(address(this)) >= toWithdraw,
            "Not enough liquidity available to withdraw"
        );

        _burn(_account, toWithdraw);
        totalStaked -= toWithdraw;
        token.safeTransfer(_receiver, toWithdraw);
    }
```

Recommended mitigation

Consider that changing *StakingPool.withdraw()* may break integration, it is recommended to require **_amount > 0** in *PriorityPool._withdraw()*.

Team response

Fixed.

Mitigation review

The fix implements the recommendation.

TRST-L-2 More LINK may be withdrawn or less LINK may be deposited due to minDeposits limit

- **Category:** Logical issues
- **Source:** StakingPool.sol
- **Status:** Fixed

Description

When the strategy withdraws LINK from the vault, if the remaining tokens in the vault are less than **minDeposits**, all are taken out and sent to the *StakingPool*. Currently, **minDeposits** is 1 for Community pool and 1000 for Operator pool.

```
                } else if (deposits - toWithdraw > 0 && deposits - toWithdraw <
minDeposits) {
                    // cannot leave a vault with less than minimum deposits
                    vault.withdraw(deposits);
                    unbondedRemaining -= deposits;
                    break;
                } else {
                    vault.withdraw(toWithdraw);
                    unbondedRemaining -= toWithdraw;
                    break;
                }
            }
        }
```

```
        uint256 totalWithdrawn = totalUnbonded - unbondedRemaining;

        token.safeTransfer(msg.sender, totalWithdrawn);
```

The extra LINK will stays in *StakingPool*, and in *_withdrawLiquidity()*, **toWithdraw** should subtracts the LINK that was actually taken out instead of **strategyCanWithdrawdraw**.

```
    function _withdrawLiquidity(uint256 _amount, bytes[] calldata _data) private {
        uint256 toWithdraw = _amount;

        for (uint256 i = strategies.length; i > 0; i--) {
            IStrategy strategy = IStrategy(strategies[i - 1]);
            uint256 strategyCanWithdrawdraw = strategy.canWithdraw();

            if (strategyCanWithdrawdraw >= toWithdraw) {
                strategy.withdraw(toWithdraw, _data[i - 1]);
                break;
            } else if (strategyCanWithdrawdraw > 0) {
                strategy.withdraw(strategyCanWithdrawdraw, _data[i - 1]);
                toWithdraw -= strategyCanWithdrawdraw;
            }
        }
    }
```

In addition, when the **deposits** do not meet **minDeposits**, the amount of LINK deposited will be less than **strategyCanDeposit**, but **toDeposit** always subtracts **strategyCanDeposit**.

```
            if (canDeposit != 0 && vaultIndex != group.withdrawalIndex
&& !vault.isRemoved()) {
                if (deposits < _minDeposits && toDeposit < (_minDeposits - deposits))
{
                    break;
                }
...
                if (strategyCanDeposit >= toDeposit) {
                    strategy.deposit(toDeposit, _data[i]);
                    break;
                } else if (strategyCanDeposit > 0) {
                    strategy.deposit(strategyCanDeposit, _data[i]);
                    toDeposit -= strategyCanDeposit;
                }
```

Both of these problems cause more tokens to be left in *stakingPool* and not stake into Chainlink Pool. This will reduce the tokens going to the Chainlink Pool, thus slightly reducing the reward.

Recommended mitigation

It is recommended to have **toDeposit** and **toWithdraw** subtract the actual LINK deposited and withdrawn instead of subtracting **strategyCanWithdrawdraw** and **strategyCanDeposit** in *StakingPool*.

Team response

Fixed.

Mitigation review

The fix makes the code track LINK balance in the *StakingPool* to get the actual LINK deposited or withdrawn, which resolves the issue.

TRST-L-3 The *getMaxDeposits()* function overestimates the amounts that can be deposited into the strategy

- **Category:** Logical issues
- **Source:** VaultControllerStrategy.sol, OperatorVCS.sol
- **Status:** Acknowledged

Description

The *getMaxDeposits()* function assumes that all vaults are available for deposits if they have enough room and have not been removed:

```
    function getMaxDeposits() public view virtual override returns (uint256) {
        (, uint256 maxDeposits) = getVaultDepositLimits();
        return
            totalDeposits +
            (
                stakeController.isActive()
                    ? MathUpgradeable.min(
 >>                     vaults.length * maxDeposits - totalPrincipalDeposits,
                        ((stakeController.getMaxPoolSize() -
stakeController.getTotalPrincipal()) *
                            maxDepositSizeBP) / 10000
                    )
                    : 0
            );
    }
```

However, in *_depositToVaults()* it can be observed that the vaults with **group.withdrawalIndex** are skipped:

```
        if (canDeposit != 0 && vaultIndex != group.withdrawalIndex && !vault.isRemoved()
```

This will result in tokens being left unused in the *StakingPool*, and *PriorityPool* upkeeps may be reverted if unused limit is exceeded.

Recommended mitigation

Subtract the deposit rooms of **group.withdrawalIndex** from the resulting available deposit amount.

Team response

Acknowledged.

Additional recommendations

TRST-R-1 Accumulated operatorRewards should be cleared when totalFeeAmounts is 0

In *StakingPool._updateStrategyRewards()*, if **totalFeeAmounts >= totalStaked**, it will set **totalFeeAmounts** to 0, i.e. no fees will be distributed.

```
        if (totalFeeAmounts >= totalStaked) {
            totalFeeAmounts = 0;
        }
```

The edge case is that in *OperatorVCS.updateDeposits()*, **operatorRewards** will be accumulated before the rewards are distributed and will not be cleared when **totalFeeAmounts** is 0. Later, when the operator claims the rewards, since these rewards do not exist, they will claim the rewards that other operators didn't claim.

```
        if (operatorRewards != 0) {
            receivers = new address[](1 + (depositChange > 0 ? fees.length : 0));
            amounts = new uint256[](receivers.length);
            receivers[0] = address(this);
            amounts[0] = operatorRewards;
            unclaimedOperatorRewards += operatorRewards;
        }
```

It is recommended to clear the accumulated **operatorRewards** when **totalFeeAmounts** is 0.

Centralization risks

TRST-CR-1 The *setWithdrawalPool()* function will freeze the funds in the old WithdrawalPool

When the owner calls *setWithdrawalPool()* to set a new **withdrawalPool**, unfilled withdrawals in the old **withdrawalPool** will be frozen and the user will not be able to withdraw their STLINK.

```
    function setWithdrawalPool(address _withdrawalPool) external onlyOwner {
        if (address(withdrawalPool) != address(0)) {

IERC20Upgradeable(address(stakingPool)).safeApprove(address(withdrawalPool), 0);
            token.safeApprove(address(withdrawalPool), 0);
        }

        IERC20Upgradeable(address(stakingPool)).safeApprove(_withdrawalPool,
type(uint256).max);
        token.safeApprove(_withdrawalPool, type(uint256).max);

        withdrawalPool = IWithdrawalPool(_withdrawalPool);
    }
```

Systemic risks

TRST-SR-1 Stale exchange rates may be used for deposits and withdrawals

StakingPool._updateStrategyRewards() traverses to get LINK in all Vaults to update the exchange rate of assets to shares. However, due to gas limitations, the protocol is unable to call *_updateStrategyRewards()* to bring the exchange rate up-to-date before depositing or withdrawing, which may result in the user making deposits and withdrawals with a stale exchange rate, resulting in slight profits or losses.

TRST-SR-2 RebaseController.updateRewards() may frontrun performUpkeep()

When the **depositChange** of any strategy is less than 0, the Automation will call *performUpkeep()* to close the priority pool. This would allow the owner to withdraw assets from the *SecurityPool* to repay the loss.

```
    function performUpkeep(bytes calldata _performData) external {
        if (priorityPool.poolStatus() == IPriorityPool.PoolStatus.CLOSED) revert
PoolClosed();

        uint256 strategyIdxWithLoss = abi.decode(_performData, (uint256));
        address[] memory strategies = stakingPool.getStrategies();

        if (IStrategy(strategies[strategyIdxWithLoss]).getDepositChange() >= 0)
            revert NoLossDetected();

        priorityPool.setPoolStatus(IPriorityPool.PoolStatus.CLOSED);
        if (address(securityPool) != address(0)) securityPool.initiateClaim();
    }
```

At the same time, if anyone calls *RebaseController.updateRewards()* first, it will repay the loss and make **depositChange** 0. This will use the assets in the *StakingPool* to repay the loss.

```
    function updateRewards(bytes calldata _data) external {
        if (priorityPool.poolStatus() == IPriorityPool.PoolStatus.CLOSED) revert
PoolClosed();
        _updateRewards(_data);
    }
```

So, when a loss occurs, the loss may be repaid by the *StakingPool* or by the *SecurityPool*.