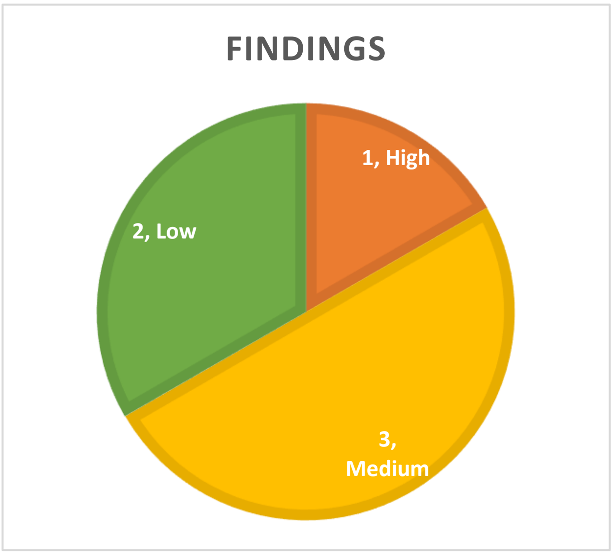
Trust Security

Smart Contract Audit

LinkPool September Upgrade

27.09.2023

Executive summary

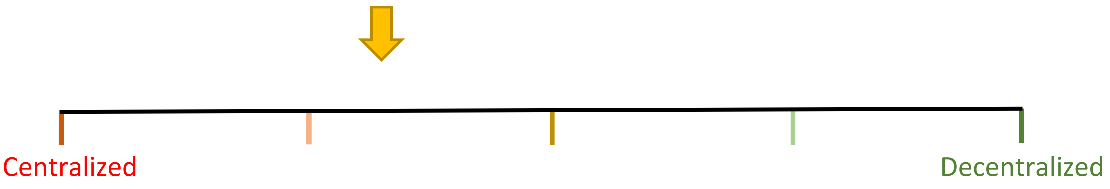


Category	Liquid Staking
Audited file count	9
Lines of Code	1594
Auditor	Lambda
Time period	18.09.2023 – 27.09.2023

Findings

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

Centralization score



Signature

Digitally signed by Trust
Date: 2023-10-01 11:18:37

Document properties

Versioning

Version	Date	Description
0.1	27.09.2023	Client report
0.2	01.10.2023	Mitigation review

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

- PriorityPool.sol
- SlashingKeeper.sol
- StakingPool.sol
- CommunityVCS.sol
- CommunityVault.sol
- OperatorVCS.sol
- OperatorVault.sol
- Vault.sol
- VaultControllerStrategy.sol

Repository details

- **Repository URL:** https://github.com/stakedotlink/contracts/
- **Commit hash:** 38ee0792f6b0590dc5bc88eac352f7f7ec2f6dbe
- **Mitigation commit hash:** f8ee1b5aa1d41e61aea44eb5e29075b50a2c1998

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

Lambda is a Security Researcher and Developer with multiple years of experience in IT security and traditional finance. This experience combined with his academic background in Data Science, Mathematical Finance, and High-Performance Computing enables him to thoroughly examine even the most complicated code bases, resulting in several top placements in various audit contests.

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	**Moderate**	There is documentation for the end user, but the code could be documented in more detail.
Best practices	**Excellent**	Project consistently adheres to industry standards and uses well tested libraries.
Centralization risks	**Moderate**	Some computations are performed off-chain and the owner has extended privileges.

Findings

High severity findings

TRST-H-1 Wrong accounting for first priority pool deposit may lead to locked funds

- **Category:** Logical flaws
- **Source:** PriorityPool.sol:L562
- **Status:** Fixed

Description

The priority pool calls the function *stakingPool.getSharesByStake()* in *_depositQueuedTokens()* to update the variable **sharesSinceLastUpdate** before making a deposit. However, *getSharesByStake()* returns 0 when no deposits were performed yet. But this value does not correspond to the mint logic in the staking pool contract, as the *_mint()* function assumes a 1:1 exchange rate for the first deposit:

```
        uint256 sharesToMint = getSharesByStake(_amount);
        if (sharesToMint == 0) {
            sharesToMint = _amount;
        }
```

This means that after the first deposit (resulting in x shares) to the staking pool, **sharesSinceLastUpdate** will have the value 0, although it should have the value x. This accounting error can lead to locked funds, as the function *updateDistribution()* decreases **sharesSinceLastUpdate** by the distributed amount. However, since the variable will be too low, the shares from the first deposit cannot be distributed.

Recommended mitigation

Either change the *getSharesByStake()* logic to assume a 1:1 exchange rate for the first deposit or handle this case explicitly in *_depositQueuedTokens()* (like in *mint()*).

Team response

Fixed

Mitigation review

The functions *getSharesByStake()* and *getStakeByShares()* now assume a 1:1 exchange rate instead of returning 0 when the total amount of shares is 0. Because of this, the special logic inside *_mint()* to handle this case has been removed and the amount that is returned by *getSharesByStake()* now always corresponds to the amount that will be minted.

Medium severity findings

TRST-M-1 Different treatment of a zero Merkle root can lead to too many tokens being removed from the queue

- **Category:** Consistency issues
- **Source:** PriorityPool.sol:L285
- **Status:** Acknowledged

Description

The functions *withdraw()* and *unqueueTokens()* both allow to remove unused tokens from the queue (by providing the corresponding merkle proof). However, they have a different logic to do so and treat the case when **merkleRoot** is zero differently. In *withdraw()*, the logic is as follows:

```
            if (_merkleProof.length != 0) {
                bytes32 node =
keccak256(bytes.concat(keccak256(abi.encode(account, _amount,
_sharesAmount))));
                if (!MerkleProofUpgradeable.verify(_merkleProof, merkleRoot,
node)) revert InvalidProof();
            } else if (accountIndexes[account] < merkleTreeSize) {
                revert InvalidProof();
            }
```

Because of this logic, the user must provide a valid Merkle proof when their address (variable **account**) has performed deposits in the past. On the other hand, the logic in *unqueueTokens()* looks like this:

```
        if (merkleRoot != bytes32(0) && accountIndexes[account] <
merkleTreeSize) {
            bytes32 node =
keccak256(bytes.concat(keccak256(abi.encode(account, _amount,
_sharesAmount))));
            if (!MerkleProofUpgradeable.verify(_merkleProof, merkleRoot,
node)) revert InvalidProof();
        }
```

When the variable **merkleRoot** is equal to zero, the check is therefore completely skipped in the function and the user can pass an arbitrary value for **_amount**, potentially allowing them to withdraw tokens that are no longer queued (because they were already distributed earlier).

Recommended mitigation

Use the same logic in both functions and handle the case when **merkleRoot** is equal to bytes32(0). Either disallow this value completely or only allow it in the beginning, but not when a Merkle root was previously set, as this allows the user to withdraw tokens that are no longer queued.

Team response

The Merkle root will never be 0 after it has been updated once but the redundant check of whether Merkle root is 0 has been removed.

Mitigation review

If a value of 0 will never occur, the current logic is fine, and it makes sense to remove the zero check.

TRST-M-2 Queue deposit limits of the priority pool can be circumvented

- **Category:** Validation flaws
- **Source:** PriorityPool.sol:L285
- **Status:** Acknowledged

Description

The priority pool contract contains two variables **queueDepositMin** and **queueDepositMax** that are the "min amount of tokens required for deposit" and "max amount of tokens that can be deposited at once" according to the inline comments. While these limits are enforced when the deposit is triggered by a Chainlink keeper (via the function *performUpkeep()*), a user can circumvent them and perform arbitrary small or large deposits. This is possible because the function *depositQueuedTokens()* does not pass these limits to the internal function *_depositQueuedTokens()*. It has the two arguments **_queueDepositMin** and **_queueDepositMax** instead, which are passed to the internal function and used as limits there.

Recommended mitigation

Remove the two arguments from *depositQueuedTokens()* and pass the configured limits to the internal function.

Team response

This is by design, the limits are only there to limit the gas used per transaction

Mitigation review

If the only purpose of the limits is to limit the gas usage, it is OK that they can be circumvented with this function. A comment has been added to clarify that this is intended behavior.

TRST-M-3 Staking pool may not deposit all tokens into the strategies

- **Category:** Capital efficiency issues
- **Source:** StakingPool.sol:L414
- **Status:** Acknowledged

Description

The *depositLiquidity()* function queries *strategy.canDeposit()* to get the amount of tokens that a strategy currently accepts. It then deposits this amount (or just the remaining balance if it is smaller than the limit) and decreases the remaining amount to deposit by the value of the variable. This implicitly assumes that the strategy accepted the whole amount. However, it is possible that a strategy takes less tokens than the value that is returned by *canDeposit()*. For instance, VaultControllerStrategy will not deposit into a vault if the deposits of the vault will be smaller than **_minDeposits** after the deposit. The remaining tokens are sent back to the staking pool instead. The *canDeposit()* function does not take this edge case into account, leading to this discrepancy between the amount that is actually deposited and the amount that the query returns. These tokens remain in the contract until the next deposit and do not earn any yield.

Recommended mitigation

There are two ways to resolve the issue:

1. Ensure that there are never be any discrepancies between *canDeposit()* and the amount that *deposit()* accepts.
2. Query the amount that was transferred and subtract this amount in *depositLiquidity()*.

For dynamic strategies with involved business logic (like VaultControllerStrategy), ensuring that there are never any discrepancies can be quite hard, so the second approach may be preferable.

Team response

Modifying *depositLiquidity()* makes sense but for the time being, we want all deposits to be directed into the first strategy even if there's space in the second strategy to accomodate the amount that wasn't deposited.

Low severity findings

TRST-L-1 No gap variable in StakingRewardsPool

- **Category:** Upgradability issues
- **Source:** StakingRewardsPool.sol:L18
- **Status:** Acknowledged

Description

StakingPool is an upgradeable contract that inherits from StakingRewardsPool. However, StakingRewardsPool does not contain any gap variables, making the addition of new variables in future upgrades much harder.

Recommended mitigation

Consider introducing a gap variable.

Team response

Acknowledged.

TRST-L-2 Discrepancy between getStrategyRewards() and the actual fees in edge cases

- **Category:** Logical flaws
- **Source:** StakingPool.sol:L331
- **Status:** Fixed

Description

The function *getStrategyRewards()* returns “the amount of fees that will be paid on the rewards” according to the documentation. However, there is one edge case where the returned amount does not correspond to the amount that will be paid out by *updateStrategyRewards()*. This happens when **totalFeeAmounts** is greater than or equal to **totalStaked**. There is special logic in *updateStrategyRewards()* to handle this case and not pay any fees:

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

But because *getStrategyRewards()* does not contain this logic, it will return a wrong fee value in such a situation.

Recommended mitigation

Consider handling this edge case in *getStrategyRewards()* as well.

Team response

Fixed

Mitigation Review

This edge case is now handled in *getStrategyRewards()* as well and the function correctly returns 0 in such a situation.

Additional recommendations

TRST-R-1 Unimplemented withdrawal

Withdrawals from strategies were not implemented in the audited codebase. Because of that, removing strategies would currently fail (as it tries to withdraw all tokens) and it was not possible to assess if withdrawals would introduce any problems for the audited scope.

For instance, the current deposit logic in VaultControllerStrategy (which generally starts at the index of the last full vault plus 1, but also checks if the first vault is full) could potentially be problematic in combination with withdrawals. Depending on the exact withdrawal logic, it may lead to vaults in the middle that never get filled again.

TRST-R-2 Events for configuration updates

Most setters for configuration variables emit an event when a new value is set. However, *OperatorVault.setRewardsReceiver()* currently does not. Consider emitting an event when the rewards receiver is updated such that off-chain monitoring systems can easily catch such updates easily and react to them.

TRST-R-3 Optimization potential when removing strategies

StakingPool.removeStrategy() changes the position of all strategies that are after the removed one. This is expensive (linear in the number of strategies) and could lead in (very unlikely) edge cases with many strategies to a situation where they need to be reordered before the removal. The operation could be optimized by swapping the removed strategy only with the last one, which would result in a constant gas cost for the function.

Centralization risks

TRST-CR-1 Off-chain Merkle root generation

The Merkle root of the priority pool contract (which determines the number of shares a user receives) is generated off-chain and regularly updated by the distribution oracle address. It is therefore not enforced on-chain that the distribution will be performed according to the reSDL balance and queued tokens. A malicious owner could create a distribution according to different rules.

Note that the generation of the Merkle tree must ensure that the shares amount for a user is always cumulative, i.e., includes the already claimed shares. This is required even when the user exits the system completely for some time and then rejoins. If the already claimed tokens are not included, the variable **sharesAmountToClaim** within *PriorityPool.claimLSDTokens()* will underflow.

TRST-CR-2 Priority pool status controlled by the owner

The owner of the priority pool can set the status of the priority pool to closed, in which case no more withdrawals are possible. If this were abused, it could to a loss of funds for the users.

TRST-CR-3 Staking pool configuration options

The owner of the staking pool can add arbitrary new strategies or change the address of the priority pool (which is allowed to make deposits / withdrawals). While there are legitimate reasons for these configuration options, they would allow a malicious owner to drain the staking pool completely.

TRST-CR-4 Upgradeable contracts

The LinkPool contracts are upgradeable, and the owner of the system is therefore able to change the implementation at any time.

Systemic risks

Chainlink Dependency

LinkPool inherently depends on the Chainlink staking system and its correct functioning. Any bug or vulnerability in the staking contract may impact LinkPool significantly and lead to a temporary or permanent loss of funds.

While the staking contracts are not directly upgradeable, a migration from version 0.1 to version 0.2 is currently planned and there may be additional migrations in the future. These migrations may change the behavior of the staking contracts and therefore influence LinkPool. Because the LinkPool contracts are upgradeable, it is possible for the team to react to such changes. It is recommended to follow the development and future updates of the staking contracts closely such that these changes are detected and there is enough time for changing the LinkPool implementation if this is needed.