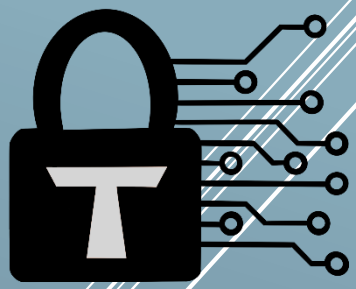
Trust Security



Smart Contract Audit

LinkPool Metis Staking

30/04/24

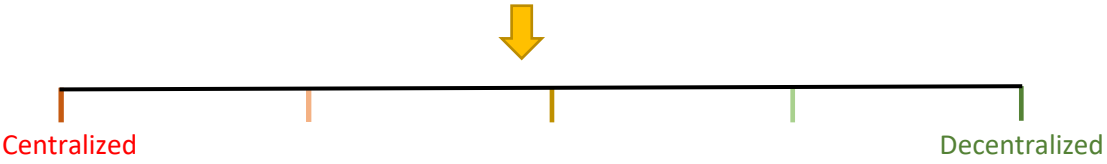
Executive summary

Category	Liquid Staking
Audited file count	4
Lines of Code	545
Auditor	cccz, 0xladboy
Time period	01/04/24-07/04/24

Findings

Severity	Total	Fixed	Acknowledged
High	-	-	-
Medium	4	3	1
Low	4	1	3

Centralization score



Signature

Digitally signed by Trust
Date: 2024.04.30 11:38:32 +03'00'

Document properties

Versioning

Version	Date	Description
0.1	07/04/24	Client report
0.2	30/04/24	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

The changes applied in the PR for the files below are considered in scope:

- metisStaking/SequencerVault.sol
- metisStaking/SequencerVCS.sol
- metisStaking/ccip/SequencerRewardsCCIPReceiver.sol
- metisStaking/ccip/SequencerRewardsCCIPSender.sol

Repository details

- **Repository URL:** https://github.com/stakedotlink/contracts
- **PR URL:** https://github.com/stakedotlink/contracts/pull/97
- **Commit hash:** 4a6789ae4f71cbeb48921b266e0a3fba35413e15
- **Mitigation review hash:** 5188534c238c3eec11442056bab4f0f52bd06811

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

As 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.

0xladboy is a researcher specializing in blockchain security, known for consistently providing top-tier audits for clients. he achieved multiple top finishes in Sherlock and C4 contests in 2023-2024.

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**	The project relies on the owner and bots to trigger crucial functions

.

Findings

Medium severity findings

TRST-M-1 Token approval is not granted to new router in setRouter()

- **Category:** Logical flaws
- **Source:** SequencerRewardsCCIPSender.sol
- **Status:** Fixed

Description

When the *SequencerRewardsCCIPSender.sol* contract is deployed, it grants approval to the *CCIP router* contract to approve and transfer **LINK** and **METIS** tokens.

```
        linkToken.approve(_router, type(uint256).max);
        metisToken.approve(_router, type(uint256).max);
```

This setup ensures that when a cross-chain message is dispatched to the *CCIP router* within the Metis L2 network, the **LINK** tokens are available to cover transaction fees, and **METIS** tokens can be transferred into the designated pool prior to the message being bridged.

However, when the *router* contract is updated: the contract does not grant approval to the new **router**.

```
    function setRouter(address _router) external onlyOwner {
        if (_router == address(0)) revert ZeroAddress();
        router = IRouterClient(_router);
    }
```

As a result, this newly updated **router** lacks the necessary approvals to transfer the **fee** tokens and **METIS** tokens, thus blocking any outgoing messages via the *CCIP router*.

Recommended mitigation

Granting **LINK** token approval and **METIS** Token approval to new **router** and reset old **router**'s approval to 0.

```
    function setRouter(address _router) external onlyOwner {
        if (_router == address(0)) revert ZeroAddress();
+       linkToken.approve(address(router), 0);
+       metisToken.approve(address(router), 0);
+       linkToken.approve(_router, type(uint256).max);
+       metisToken.approve(_router, type(uint256).max);
        router = IRouterClient(_router);
    }
```

Team response

Fixed.

Mitigation Review

The fix implements the recommendation.

TRST-M-2 SequencerVault.updateDeposits should be payable to call LockingPool.withdrawalRewards()

- **Category:** Integration issues
- **Source:** SequencerVault.sol
- **Status:** Fixed

Description

When **METIS** tokens accrue rewards, the vault controller can execute the *updateDeposits()* function within each vault to withdraw rewards from the *LockingPool*.

```
        uint256 claimedRewards;
        if (_minRewards != 0 && rewards >= _minRewards) {
            lockingPool.withdrawRewards(seqId, _l2Gas);
            trackedTotalDeposits -= SafeCastUpgradeable.toUint128(rewards);
            totalDeposits -= rewards;
            claimedRewards = rewards;
        }
```

Managed by the Metis team, the *LockingPool* orchestrates the withdrawal of rewards by bridging L1 **METIS** tokens to the METIS network.

The *withdrawRewards()* function in the *LockingPool* contract is marked as payable and requires ETH payment, and the accompanying **ETH** is directed to an escrow contract to cover transaction fees.

Subsequently, the *_bridgeTo()* internal function is called, necessitating the transfer of **ETH** to the bridge contract to finalize the bridging process. Without this step, the reward withdrawal remains incomplete.

```
    function _bridgeTo(
        address _recipient,
        uint256 _amount,
        uint32 _l2gas
    ) internal {
        if (_amount == 0) {
            return;
        }

        IERC20(l1Token).safeIncreaseAllowance(bridge, _amount);
        IL1ERC20Bridge(bridge).depositERC20ToByChainId{value: msg.value}(
            l2ChainId,
            l1Token,
            l2Token,
            _recipient,
            _amount,
            _l2gas,
            ""
        );
    }
```

Because the *updateDeposits()* function is not marked as payable and does not send ETH when triggering *withdrawRewards(), withdrawRewards()* will always revert because the bridge requires ETH fee.

Recommended mitigation

To address the identified issue, the *updateDeposits()* function should be modified to be **payable**, enabling it to forward the received **ETH** to an external call, specifically *lockingPool.withdrawRewards()*.

However, this amendment may introduce a potential issue: in instances where *relockRewards()* is called by anyone before the rewards is zero, *withdrawRewards()* will not be called, and any ETH sent as a fee will become locked within the *SequencerVault* contract.

```
        if (_minRewards != 0 && rewards >= _minRewards) {
            lockingPool.withdrawRewards(seqId, _l2Gas);
            trackedTotalDeposits -= SafeCastUpgradeable.toUint128(rewards);
            totalDeposits -= rewards;
            claimedRewards = rewards;
        }
```

To prevent this undesirable outcome, it is crucial to withdraw unspent ETH at the end of *Vault.updateDeposits()* and in *VCS.updateDeposits()*.

Team response

Fixed.

Mitigation Review

The fix implements the recommendation.

TRST-M-3 getTotalDeposits() in SequencerVCS contract does not consider restaked rewards

- **Category:** Logical flaws
- **Source:** SequencerVCS.sol
- **Status:** Fixed

Description

In *stakingPool.depositLiquidity()*, *canDeposit()* is the maximum token amount that can be deposited into the strategy.

```
    function depositLiquidity() public {
        uint256 toDeposit = token.balanceOf(address(this));
        if (toDeposit > 0) {
            for (uint256 i = 0; i < strategies.length; i++) {
                IStrategy strategy = IStrategy(strategies[i]);
                uint256 strategyCanDeposit = strategy.canDeposit();
                if (strategyCanDeposit >= toDeposit) {
                    strategy.deposit(toDeposit);
                    break;
                } else if (strategyCanDeposit > 0) {
                    strategy.deposit(strategyCanDeposit);
                    toDeposit -= strategyCanDeposit;
                }
```

```
            }
        }
    }
```

canDeposit() is normally equal to *getMaxDeposits() - getTotalDeposits().*

```
    function canDeposit() public view virtual returns (uint256) {
        uint256 deposits = getTotalDeposits();
        if (deposits >= getMaxDeposits()) {
            return 0;
        } else {
            return getMaxDeposits() - deposits;
        }
    }
```

SequencerVCS.getMaxDeposits() is as follows and **maxLock** is currently 100,000.

```
    function getMaxDeposits() public view override returns (uint256) {
        return vaults.length * lockingInfo.maxLock();
    }
```

getTotalDeposits() is as follows, **totalDeposits** represents the amount deposited through *deposit()*, **l2Rewards** represents the amount of rewards that have been claimed but not yet credited.

```
    function getTotalDeposits() public view override returns (uint256) {
        return totalDeposits + l2Rewards;
    }
```

The problem here is that the re-staked reward tokens will not count in **totalDeposits** and **l2Rewards**.

Consider that there is only 1 Vault, **totalDeposits** is 50,000, 8,000 rewards are generated.

Alice calls *relockRewards()* to restake 8,000 rewards.

canDeposit() is supposed to be 100,000 - (50,000 + 8,000) = 42000.

However, **totalDeposits** = 5000, **l2Rewards** = 0, and *canDeposit()* is 50000.

Also, **l2Rewards** is not supposed to be considered in *getTotalDeposits()*, even though it represents un-credited rewards, it is already subtracted from sequencer and the rewards will be sent to the *stakingPool* via CCIP.

Recommendation Mitigation

It is recommended to implement *getTotalDeposits()* as follows

```
    function getTotalDeposits() public view override returns (uint256) {
-       return totalDeposits + l2Rewards;
+       uint256 totalDeposit = token.balanceOf(address(this));
+       for (uint256 i = 0; i < vaults.length; ++i) {
+           totalDeposit += vaults[i].getTotalDeposits();
+       }
+       return totalDeposit;
    }
```

Team response

Fixed.

Mitigation Review

The fix moves the relock logic into *updateDeposits()* so that re-staked rewards are immediately counted into **totalDeposits** when *updateDeposits()* is called.

TRST-M-4 Permissionlessly triggering relockRewards() in vaults leads to denial of service

- **Category:** Logical flaws
- **Source:** SequencerVault.sol
- **Status:** Acknowledged

Description

In the *LockingPool* contract, when the function *relock()* is called, it will clear the pending reward and add the pending reward as deposit balance for sequencer, but if there is no newly added locked amount, the transaction will revert with error "**No new locked added**".

```
    function increaseLocked(
        uint256 _seqId,
        uint256 _nonce,
        address _owner,
        uint256 _locked,
        uint256 _incoming,
        uint256 _fromReward
    ) external override OnlyManager {
        require(_locked <= maxLock, "locked>maxLock");

        // get increased number and transfer it into escrow
        uint256 increased = _incoming + _fromReward;
        require(increased > 0, "No new locked added");
        IERC20(l1Token).safeTransferFrom(_owner, address(this), _incoming);

        // get current total locked and emit event
        uint256 _totalLocked = totalLocked + increased;
        totalLocked = _totalLocked;

        emit Relocked(_seqId, increased, totalLocked);
        emit LockUpdate(_seqId, _nonce, _locked);
    }
```

The *relockRewards()* function in *SequencerVault.sol* doesn't have access control. It adds all pending rewards to the sequencer's lock amount. The *SequencerVCS* contract lets anyone run *relockRewards()* for all vaults.

```
    function relockRewards() external {
        if (seqId == 0 || getRewards() == 0) revert NoRewards();
        lockingPool.relock(seqId, 0, true);
    }
```

If user frontruns the *SequencerVCS.relockRewards()* by triggering *relockRewards()* for individual vaults, the *SequencerVCS.relockRewards()* will revert because *getRewards()* returns 0 and revert the transaction.

Recommended mitigation

Adding access control to function *relockRewards()* for both the Vault contract and VCS contract.

Team response

Acknowledged.

Low severity findings

TRST-L-1 CCIPSender payment token should be configurable

- **Category:** Integration issues
- **Source:** SequencerRewardsCCIPSender.sol
- **Status:** Acknowledged

Description

The CCIPSender payment token used as a fee is always **LINK** token. However, the CCIP admin can remove / unsupport a specific payment token as fee, in the case when the **LINK** token is disable as fee token, no cross-chain message can be sent via CCIP.

```
  function getFee(
    uint64 destChainSelector,
    Client.EVM2AnyMessage calldata message
  ) external view returns (uint256 feeTokenAmount) {
    if (destChainSelector != i_destChainSelector) revert
InvalidChainSelector(destChainSelector);

    uint256 gasLimit = _fromBytes(message.extraArgs).gasLimit;
    // Validate the message with various checks
    _validateMessage(message.data.length, gasLimit, message.tokenAmounts.length);

    FeeTokenConfig memory feeTokenConfig = s_feeTokenConfig[message.feeToken];
    if (!feeTokenConfig.enabled) revert NotAFeeToken(message.feeToken);
```

Recommended mitigation

Implement function to add/remove support for payment token in CCIP Sender contract.

Team response

Acknowledged.

TRST-L-2 Lack of function to unlock and claim the locked METIS

- **Category:** Integration issues
- **Source:** SequencerVCS.sol
- **Status:** Acknowledged

Description

It would require the vault contract to call *unlock()* and *unlockClaim()* via *LockingPool* to unstake the locked **METIS**, but in the current version of the vault contract, such feature is missing, results in protocol not able to claim the locked fund back.

Recommended mitigation

Add functions to call *unlock()* and *unlockClaim()*.

Team response

Acknowledged.

TRST-L-3 Funds can get stuck in SequencerVCS

- **Category:** Logical flaws
- **Source:** SequencerVCS.sol
- **Status:** Acknowledged

Description

In *SequencerVCS* contract, after the user deposits the fund from *StakingPool* to *SequencerVCS* and before the deposit controller deposit the fund from *SequencerVCS* to *SequencerVault*, there is a time gap. If the admin of the *lockingPool* in metis side adjusts and reduces the max lock token limit, a certain portion of METIS cannot go to the vault, which results in stuck funds.

Recommended mitigation

In *SequencerVCS.updateDeposit()*, consider adding code to transfer any METIS that is sitting idle in the *SequencerVCS* contract, back to *stakingPool.*

Team response

Acknowledged.

TRST-L-4 Strategy rewards are not updated before updating the fees

- **Category:** Logical flaws
- **Source:** SequencerVCS.sol
- **Status:** Fixed

Description

Before updating the **operatorRewardPercentage** in *SequencerVCS*, *_updateStrategyRewards()* is called to update the rewards based on the old **operatorRewardPercentage**

```
    function setOperatorRewardPercentage(uint256 _operatorRewardPercentage) public
onlyOwner {
        if (_operatorRewardPercentage > 10000) revert InvalidPercentage();

        _updateStrategyRewards();

        operatorRewardPercentage = _operatorRewardPercentage;
        emit SetOperatorRewardPercentage(_operatorRewardPercentage);
    }
```

addFee() and *updateFee()* also change the distribution of rewards, so *_updateStrategyRewards()* should also be called before updating fees to update rewards based on the old fees.

```
    function addFee(address _receiver, uint256 _feeBasisPoints) external onlyOwner {
        fees.push(Fee(_receiver, _feeBasisPoints));
        if (_totalFeesBasisPoints() > 5000) revert FeesTooLarge();
    }
...
    function updateFee(
        uint256 _index,
        address _receiver,
```

```
        uint256 _feeBasisPoints
    ) external onlyOwner {
        if (_feeBasisPoints == 0) {
            fees[_index] = fees[fees.length - 1];
            fees.pop();
        } else {
            fees[_index].receiver = _receiver;
            fees[_index].basisPoints = _feeBasisPoints;
        }

        if (_totalFeesBasisPoints() > 5000) revert FeesTooLarge();
    }
```

Recommended mitigation

It is recommended to call *_updateStrategyRewards()* before the functions *addFee()* and *updateFee()* update fees.

Team response

Fixed.

Mitigation Review

The fix implements the recommendation.

Additional recommendations

TRST-R-1 Make sure to thoroughly test the CCIP sender and receiver.

Currently chain CCIP does not support CCIP bridging from METIS to Ethereum mainnet, so it is recommended to add thorough integration tests for the CCIP sender and receiver when support for METIS is enabled.

TRST-R-2 upgradeVaults() should use different data for vaults

It would be better if the data is of type bytes[] in *upgradeVaults()*, to apply different selectors or parameters for different Vaults when upgrading.

```
    function upgradeVaults(
        uint256 _startIndex,
        uint256 _numVaults,
-       bytes memory _data
+       bytes[] memory _data
    ) external onlyOwner {
        for (uint256 i = _startIndex; i < _startIndex + _numVaults; ++i) {
-           if (_data.length == 0) {
+           if (_data[i-_startIndex].length == 0) {
                vaults[i].upgradeTo(vaultImplementation);
            } else {
-               vaults[i].upgradeToAndCall(vaultImplementation, _data);
+               vaults[i].upgradeToAndCall(vaultImplementation, _data[i-_startIndex]);
            }
        }
        emit UpgradedVaults(_startIndex, _numVaults, _data);
    }
```

TRST-R-3 The total reward ratio should be less than 100%

Currently, **operatorRewardPercentage** is less than 100%, sum of fees in *VCS* is less than 50%, and sum of fees in *StakingPool* is less than 50%. Since they have the same base, which is the reward generated by staking, it seems possible to make the fees greater than the reward. It is recommended that more appropriate limits should be applied to the reward ratios, e.g. 50%, 25%, 25%.

TRST-R-4 Vault.relockRewards() function should not allow anyone to update sequencer operator's nonce

Anyone can call *SequencerVault.relockRewards()* to update and increment the sequencer operator's nonce in *LockingPool*.

```
    function relockRewards() external {
        if (seqId == 0 || getRewards() == 0) revert NoRewards();
        lockingPool.relock(seqId, 0, true);
```

```
    }
    …
    function relock(
        uint256 _seqId,
        uint256 _amount,
        bool _lockReward
    ) external whenNotPaused whitelistRequired {
        Sequencer storage seq = sequencers[_seqId];
        …
        uint256 locked = seq.amount + _amount + _fromReward;
        uint256 nonce = seq.nonce + 1;

        seq.nonce = nonce;
```

The nonce is the sequencer operation number and starts from 1, and used internally by the metis consensus client.

To avoid any integration issues, we recommend adding access control to *SequencerVault.relockRewards()*.

Centralization risks

TRST-CR-1 Dependency on the Deposit controller to manage the fund

The deposit funds go to *SequencerVCS* which depends on deposit controller to deposit the funds into the vault to generate reward yield. Since owner can set any deposit controller, there is a risk of introducing a faulty deposit controller will prevent the staked funds from generating reward yield.

TRST-CR-2 Dependency on the owner to add and upgrade vault

The system highly depends on the protocol owner to create vaults, and the owner can upgrade vaults to any new implementation. Thus, a compromised admin account can lead to severe monetary losses of stakeholders.

Systemic risks

TRST-SR-1 Dependency on Metis Staking to generate stake yield

The protocol deposits funds into Metis's locking pool to generate yield, and the yield is sent to Metis L2 via Metis's standard bridge. Faults in this process may put the funds at risk.

TRST-SR-2 Dependency on CCIP to cross-chain rewards

Yields on Metis L2 are sent back to L1 via Chainlink CCIP. Faults in this process may put yields at risk.

TRST-SR-3 Dependency on offchain infrastructure

The contracts rely on regular calls by offchain bots. The offchain infrastructure that is maintained to run these bots was out of scope for this review. It should be noted that there is an inherent dependency on this infrastructure. If it is compromised or not available, the smart contracts will be affected.