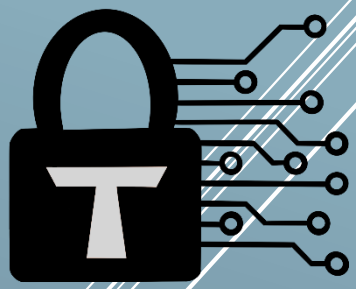
Trust Security



Smart Contract Audit

Lair Finance – Somnia Staking

27/10/25

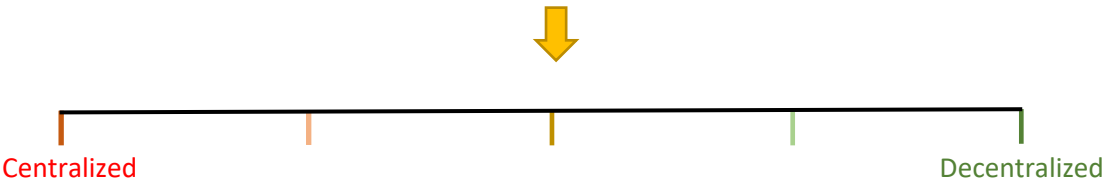
Executive summary

Category	Liquid Staking
Audited file count	9
Lines of Code	590
Auditor	Trust
Time period	20/10/25 – 23/10/25

Findings

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

Centralization score



Signature

Digitally signed by Trust
Date: 2025.10.27 12:40:45 +02'00'

Document properties

Versioning

Version	Date	Description
0.1	23/10/25	Client report
0.2	27/10/25	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

- NodeController.sol
- NodeService.sol
- StakingToken.sol
- INodeService.sol
- INodeController.sol
- IStakingToken.sol
- Validator.sol
- UnStake.sol
- Node.sol

Repository details

- **Repository URL:** https://github.com/bug4city/bughole-lsd
- **Commit hash:** 79d1ac219d12c761b04a5d2f53f68c20c82e1f8b
- **Mitigation review commit hash:** 376e6d3c9bc958810e3694da9ab81743705ba18b

About Trust Security

Trust Security has been established by top-end blockchain security researcher Trust, in order to provide high quality auditing services. Since its inception it has safeguarded over 30 clients through private services and over 30 additional projects through bug bounty submissions.

About the Auditors

Trust has established a dominating presence in the smart contract security ecosystem since 2022. He is a resident on the Immunefi, Sherlock and C4 leaderboards and is now focused in auditing and managing audit teams under Trust Security. When taking time off auditing & bug hunting, he enjoys sharing knowledge and experience with aspiring auditors through X or the Trust Security blog.

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 reasonably documented, but could be slightly improved.
Best practices	**Good**	Project mostly adheres to industry standards. Some deviations from standard practices have been identified.
Centralization risks	**Moderate**	The contract admins are fully trusted.

Findings

High severity findings

TRST-H-1 Undelegations of zero value could cause DoS of key functionality

- **Category:** Functionality issues
- **Source:** NodeService.sol
- **Status:** Fixed

Description

From discussion with the client, the *undelegateStake()* can revert if passed an amount of zero. However, in two locations in NodeService, the call could be made with such calls:

In *withdrawDelegatorRewards()*:

```
if (remainRewardAmount > 0 || remainStakingAmount > 0) {
    node.undelegateStake(validatorAddress,
getNodeTotalNativeAmount());
    node.delegateStake{value:
address(this).balance}(validatorAddress, address(this).balance);
} else if (remainStakingAmount > 0) {
```

In *applyReDelegate()*:

```
node.undelegateStake(validatorAddress, totalAmount);
node.delegateStake{value: msg.value}(validatorAddress,
totalAmount + msg.value);
```

The functionality above will be disrupted and denied due to possibly passing value of zero. In the case of *withdrawDelegatorRewards()*, this occurs periodically and for all NodeService contracts, as part of the preamble for staking and unstaking. As a result, the flaw could result in a wide DoS until the admin somehow restores correct behavior.

Recommended mitigation

Check if the amount passed to *undelegateStake()* is zero, if so, the call should be skipped.

Team response

Fixed.

Mitigation review

Fixed as recommended.

TRST-H-2 Lack of proper accounting in certain functions will lead to freeze of rewards

- **Category:** Accounting flaws
- **Source:** NodeService.sol

- **Status:** Fixed

Description

The Staking interface describes that *undelegateStake()* would send the undelegated stake plus accrued rewards of the delegator. However, in *_claim()*, *redelegate()*, *applyReDelegate()*, the functions do not register the dispatched rewards in the **remainRewardAmount** and **totalStakingAmount**. Since the reported value of the NodeService is calculated as **totalStakingAmount – totalUnStakingAmount**, the rewards would be stuck in the contract and never redeemable by users.

Recommended mitigation

In the functions discussed, wrap the *undelegateStake()* with a balance before/after check, and increase the staked amount accordingly. Note that the rewards should either be accounted for later staking, or staked immediately, in this case *undelegateStake()* should remove the entire holding to avoid *delegateStake()* from reverting.

Team response

Fixed.

Mitigation review

The issue has been addressed by making sure all code paths begin with distribution of rewards, therefore undelegation will only remove the principal amount.

TRST-H-3 Fee accrual on rewards is accounted incorrectly, leading to various financial inaccuracies

- **Category:** Accounting issues
- **Source:** StakingToken.sol
- **Status:** Fixed

Description

In *changeRatio*(), the current ratio is determined using the existing staking, new rewards, and total amount of shares. Then, this ratio is used to mint fee shares to the treasury:

```
uint256 currentRewardStakingToken = getRatioStakingTokenByNativeToken(fee);
_mint(treasuryAddress, currentRewardStakingToken);
```

The issue is that the code intends to mint shares redeemable for **fee** amount of tokens, but the minting operation inflates the token supply, so the result is an amount that redeems for less than **fee**. Secondly, since the mint operation occurs after ratio, there's a situation of ratio being temporarily incorrect – the sum of all shares multiplied by the ratio results in more than the available native in the contract. The ratio will be corrected at the next *distibributeReward()*, in any *unstake()*, *stake()* and *claim()*. However if logic is executed between the *changeRatio()* and the next distribution, it will be done under wrong ratio. So, if the *changeRatio()* is done at the start of *unstake()* or *stake()* call, user will receive more native tokens, or receive less shares respectively. An unfortunate user would in essence have their holdings suffer inflation from rewards from before they ever entered the staking.

Recommended mitigation

The math should be refactored to calculation that the minted shares line up with the fee. The ratio should monotonically increase to prevent any abuse.

Team response

Fixed.

Mitigation review

Issue has been addressed by removing the minting code from the StakingToken and minting native tokens in the NodeService.

TRST-H-4 The unstaking node selection logic skips inactive nodes, leading to unstaking DoS

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

Description

In *unstake()* of the NodeController, nodes are selected using *unstakeNode()* to get the node with most stakes, which is done in a loop. However, the logic skips all elements prior to the first active node, after which any node activity is accepted. The intention is for *unstake()* to be available for all active and inactive nodes, otherwise users would not be able to withdraw.

Recommended mitigation

Start processing the loop from the first node.

Team response

Fixed.

Mitigation review

Fixed as recommended.

Medium severity findings

TRST-M-1 Users could bypass fees due to rounding down logic

- **Category:** Rounding errors
- **Source:** StakingToken.sol
- **Status:** Fixed

Description

The StakingToken charges fees on staked, unstaked, and rewarded amounts. For example, the logic below charges the deposit fee:

```
uint256 fee = realNativeTokenAmount.mul(depositFeeRatio).div(PERCENT);
if (fee > 0) {
    realNativeTokenAmount = realNativeTokenAmount.sub(fee);
    (bool success, ) = treasuryAddress.call{value: fee}("");
    require(success, "StakingToken::stake: failed to send fee to treasury");
}
```

Since the fee is rounded down, users can theoretically bypass it by specifying a **realNativeTokenAmount** which satisfies **realNativeTokenAmount * depositFeeRatio < 10000** , repeating as desired to achieve the full intended staking amount. The effectiveness of this method depends on the size of the fee ratio and the gas costs, as it would likely not be viable to perform it when gas costs are high.

It is common practice for protocols to round in favor of the fee recipient in order to avoid gaming the fees as above.

Recommended mitigation

Round up when calculating fees in *_stake()*, *_unStake()*, *changeRatio()*.

Team response

Fixed.

Mitigation review

Issue was fixed as suggested.

TRST-M-2 Redelegate functionality is broken due to flaw in applyReDelegate()

- **Category:** Functionality issues
- **Source:** NodeService.sol
- **Status:** Fixed

Description

In *applyReDelegate()*, the contract undelegates all existing stake and then calls *delegateStake()*, intending to pass entire previous value and the new native value. However, the amount passed is not in sync with the value:

```
node.delegateStake{value: msg.value}(validatorAddress, totalAmount +
msg.value);
```

Presumably, the underlying logic will revert as the requested amount does not match the value passed. This means any *reDelegate()* call will fail, which is important logic for ensuring efficient distribution.

Recommended mitigation

Change the **value** parameter to include **totalAmount**.

Team response

Fixed.

Mitigation review

The contract now includes the entire amount in the call.

TRST-M-3 Reward accounting error in *withdrawDelegatorRewards()* could lead to temporary freeze of funds

- **Category:** Accounting issues
- **Source:** NodeService.sol
- **Status:** Fixed

Description

In *withdrawDelegatorRewards()*, it stores the last claimed amount in **rewardAmount**. Note that even if such amount is not zero, the delegation logic may not activate, since **rewardAmount** does not accrue to **remainRewardAmount** or **remainStakingAmount:**

```
if (remainRewardAmount > 0 || remainStakingAmount > 0) {
    node.undelegateStake(validatorAddress, getNodeTotalNativeAmount());
    node.delegateStake{value: address(this).balance}(validatorAddress,
address(this).balance);
} else if (remainStakingAmount > 0) {
    node.delegateStake{value: remainStakingAmount}(validatorAddress,
remainStakingAmount);
}
```

Thus the rewards will be accounted in **totalStakingAmount**, but not actually staked. This could lead to a temporary freeze of funds, because the StakingToken ratio is updated through **totalStakingAmount**, but the funds are not available to be unstaked, so if enough unstakes occur, the matching native to be withdrawn will revert. Note that it is not a permanent freeze of funds because at some point **remainRewardAmount** or **remainStakingAmount** would be positive from events like new staking thus the entire **address(this).balance** will be sent to the *delegateStake()*, and accounting will be correct for the unaccounted portion.

Recommended mitigation

The logic should be refactored to restake any new rewards in *withdrawDelegatorRewards()*.

Team response

Fixed.

Mitigation review

The code was refactored and any spare balance will be restaked in the function.

TRST-M-4 Redelegation may fail due to double-counting of pending unstakes

- **Category:** Accounting issues
- **Source:** NodeService.sol
- **Status:** Fixed

Description

When unstaking, the NodeService adds the amount to both **totalUnStakingAmount** and **remainUnStakeAmount**. The former is part of the *getTotalNativeAmount()* calculation:

```
function getTotalNativeAmount() public view returns (uint256) {
    return totalStakingAmount - totalUnStakingAmount;
}
```

Meanwhile, the following check exists in *reDelegate()*:

```
require(amount + remainUnStakeAmount <= getTotalNativeAmount(),
"NodeService:: not enough withdrawable amount after unStake : ");
```

The intention is to maintain sufficient amount in the contract to fulfill pending unstakes. However, it ends up applying the amount to both sides of the inequality.

Recommended mitigation

Remove **remainUnStakeAmount** from the calculation above.

Team response

Fixed.

Mitigation review

The **remainUnStakeAmount** variable has been removed.

Low severity findings

TRST-L-1 Rewards are not accrued when total supply is zero

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

Description

In *changeRatio()*, the token should accrue the **currentReward**, however if there are no supply units, the code does an early exit. At this point, the **currentReward** will never be notified again, although they remain available on the NodeService contracts. Thus, the value will be leaked.

Recommended mitigation

Consider adding the rewards to **totalStaking**, then the next time supply will go above zero, the holders could enjoy a share of the rewards sent. Make sure to skip the ratio calculation as the denominator is zero.

Team response

Fixed.

Mitigation review

The code has been changed and there is no early return for the **totalSupply() == 0** case. However, a new issue arises since later in the function, there is an operation with **totalSupply()**, causing a division by zero panic. It's possible to have a positive reward while supply is zero, since for example a user could donate some assets to the contract, which accrue new rewards in *withdrawDelegatorRewards()*. In this case, all future activity in the contract will be denied as the distribution will revert. This could happen during the natural lifecycle of the contract when there are no tokens staked, or as a DoS vector during setup of the contracts, right after deployment.

It is recommended to explicitly handle the **totalSupply() == 0** case.

Team response

Fixed.

Mitigation review #2

Issue has been addressed as suggested.

TRST-L-2 Rounding in favor of user could introduce insolvency and breaks ratio invariants

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

Description

During staking, the *getRatioStakingTokenByNativeToken()* function is used to get the matching token shares to mint by applying the ratio.

```
function _getRatioStakingTokenByNativeToken(uint256 amount) private view
returns (uint256) {
    return ratio == 0 ? 0 : amount.mul(decimalsOffset).div(ratio);
}
```

In *changeRatio()*, it is observed that **ratio** is rounded down.

```
ratio = totalStaking.mul(decimalsOffset).div(totalSupply());
```

Since the **ratio** is rounded down, and the native -> shares conversion divides by it, the overall calculation rounds up (even though the statement doesn't use *divUp()*). This creates a weak spot where attacks may abuse the rounding to get more shares than they should. Also, when giving users even a small discount, the invariant where ratio only increases (token is appreciating) is broken.

Recommended mitigation

In the calculation above, consider using **ratio+1** to complete the conversion, to ensure the ratio is acting slightly against the user instead of in their favor. Even though it could be inaccurate, it maintains safety for the protocol.

Team response

Fixed.

Mitigation review

Issue has been addressed as suggested.

TRST-L-3 Fees can be changed instantly and to the dissatisfaction of users with transactions yet to be inserted

- **Category:** Time-sensitivity issues
- **Source:** StakingToken.sol
- **Status:** Fixed

Description

The fee parameters of the StakingToken are configurable by an admin. A user calling *stake()* or *unstake()* could be surprised by a change of the fee parameters, which can take place from the moment the transaction is in the air, until it is executed in a block.

Recommended mitigation

The *stake()* and *unstake()* functions should receive slippage parameters and revert if user is getting less tokens back than they expected. They should also pass a deadline parameter, which is considered a best practice and a secondary slippage protection parameter.

Team response

Fixed.

Mitigation review

Issue has been addressed by introducing slippage parameters.

TRST-L-4 Unstaking node selection is very gas inefficient and could lead to DoS

- **Category:** Gas-related issues
- **Source:** NodeController.sol
- **Status:** Acknowledged

Description

During unstaking, a loop is executed as long as there is a remaining amount to unstake. In each iteration, *unstakeNode()* itself loops over potentially all nodes to get the one with highest availability. Since the algorithm is $O(n^2)$, there is a risk that with even a relatively small number of nodes, the gas expense could be too high to fit in a block, and make it impossible to finish the *unstake()* logic. It should be considered that the system is permissionless and any user could manipulate the available amounts in each node, thus providing a vector to block users from unstaking.

Recommended mitigation

Consider refactoring the algorithm to O(n) complexity as commonly done in smart contract logic.

Team response

“Currently, there are no major issues. However, if frequent errors occur due to the addition of many nodes, we plan to modify the system to allow direct unstake requests for nodes.”

Additional recommendations

TRST-R-1 Treasury pays fees to itself in a perpetual loop

During *unstake()* and *changeRatio()*, the treasury receives shares, however to access their value, they would need to be unstaked, which would trigger another round of shares to be minted. This circular behavior does not lead to lost value, but is counterproductive, and it is best to skip fee logic for the treasury address.

TRST-R-2 Improve contract readability

Some variables, like **decimalsOffset**, are fixed, so they should be capitalized and marked as constant.

TRST-R-3 Remove repeated modifiers throughout the codebase

Throughout the codebase, there are many redundant modifiers used. For example, *whenNotPaused()* is performed both in external and internal functions. The *validMoreThanEqualZero()* modifier should be removed as it can never revert. Consider going over the entire codebase and removing all redundant instances to save gas and improve clarity.

TRST-R-4 Avoid CEI violations

The code is overall protected from reentrancy using **nonReentrant** guards in all contracts. However, to protect from potential mistakes or from cross-contract reentrancy attacks, it is important to maintain correct CEI behavior whenever possible. In *_claim()*, it can be observed that the native token is first sent to the untrusted account, and then marked as completed. It is recommended to call the user's code only at the very end of the function.

TRST-R-5 Remove no-ops

There are several instances of code which doesn't have any effect:

- When claiming, the **userUnstakeInfo[account][index].state** is set twice.
- In *getUnStakeRequestInfos(),* the check below can never be reached because **i < end** and **end** can't be higher than array **length**.
  ```
  if (i >= userUnStakeInfo[account].length) {
  ```
- The line below can never revert:

```
require(stakingOf(account) >= nativeTokenAmount,
"StakingToken::unStake: insufficient nativeTokenAmount");
```

TRST-R-6 Standardize page access between contracts

In the NodeService, accessing pages via *getUnStakeRequestInfos()* is zero-based, and in NodeController, *getUnStakeInfos()* is one-based.

TRST-R-7 Remove double-calls to distributeReward()

Since all *stake()* and *unstake()* actions are initiated at the StakingToken which calls *distributeReward()*, it is not needed to call the internal distribution function in the NodeController.

TRST-R-8 Bias towards lower indexes should be corrected or documented

In *selectStakeNode()* and *unstakeNode()*, the lower index is always preferred in case the amount compared is equal to another node. This should either be documented, or randomized, in order to maintain fairness between the nodes.

TRST-R-9 Avoid possible share manipulation attacks by minting dead shares

It is a very common protection measure in DeFi applications that mint share tokens to mint dead (inaccessible) shares when the contracts are initialized – to prevent share inflation attacks which can occur when the share amount is very small. It is recommended to do so for StakingToken to avoid possible share-related exploits around the **ratio** parameter.

TRST-R-10 Try/catch statements could expose controlled-gas vectors

The NodeService uses try/catch to handle errors from *getDelegationByValidator()* and *getDelegatedStakerRewards()* APIs. When using such statements it is important to consider the impact of user passing a gas allowance that is too smaller to execute the statement, yet enough so that the outer context can finish execution (using the 63/64 rule). In this case, the internal interface is out of scope, but if spending is significant, attacker could forge a malicious delegation amount or reward amount of zero. Consider requiring a sensible amount of gas to be available before calling the external interface, to ensure it does not revert due to OOG. Also consider if any other error should always be treated as an amount of zero, or if some errors should be considered critical and force a revert at the dApp context.

TRST-R-11 Contracts should not have receive() function by default

The NodeService requires a *receive()* function in order to receive rewards and undelegated amounts. However, the NodeController and StakingToken do not need to have such a function as they do not expect to receive tokens (those are permanently stuck). It is recommended to remove the *receive()* implementation to protect from users mistakenly sending native coins to the contract.

Centralization risks

TRST-CR-1 Admin is fully trusted

The protocol should be considered fully trusted. A malicious admin is able to update critical addresses in the protocol contracts, causing freeze or loss of funds.

Systemic risks

TRST-SR-1 Integration Risks

The contract integrates with the ISomniaStakingV2 contract. Any issue with the contract or how it is expected to operate with the dApp could lead to functionality problems. The source of the contract has not been made available for the duration of the audit.

TRST-SR-2 Delegation inefficiencies

The algorithms for choosing staking and unstaking nodes are simplistic. It is acknowledged that attackers could manipulate the selection of such nodes whilst paying the relevant costs for interaction. The admin can call *redelegate()* to restore correct allocations, but it should be understood that this could be a never-ending cycle of fixing and unfixing of the state.