ZEALYNX SECURITY

Web3 Security & Smart Contract Development

X

shieldify

LIDO FINANCE

Security Assessment

August, 2024 - Prepared by Zealynx Security

ZealynxSecurity@protonmail.com

Secoalba	@Seecoalba
Bloqarl	@TheBlockChainer

Contents

1. About Zealynx

Zealynx Security, founded in January 2024 by Bloqarl and Sergio, specializes in smart contract audits, development, and security testing using Solidity and Rust. Our services include comprehensive smart contract audits, smart contract fuzzing, and formal verification test suites to enhance security reviews and ensure robust contract functionality. We are trusted by clients such as Shieldify, AuditOne, Bastion Wallet, Side.xyz, Possum Labs, and Aurora (NEAR Protocol-based).

Our team believes in transparency and actively contributes to the community by creating educational content. This includes challenges for Foundry and Rust, security tips for Solidity developers, and fuzzing materials like Foundry and Echidna/Medusa. We also publish articles on topics such as preparing for an audit and battle testing smart contracts on our blog, Bloqarl's blog, and Sergio's blog.

Zealynx has achieved public recognition, including a Top 5 position in the Beanstalk Audit public contest, rewarded with $8k. Our ongoing commitment is to provide value through our expertise and innovative security solutions for smart contracts.

2. Disclaimer

A security review of a smart contract can't guarantee there are no vulnerabilities. It's a limited effort in terms of time, resources, and expertise to find as many vulnerabilities as possible. We can not assure 100% security post-review or guarantee the discovery of any issues with your smart contracts.

3. Overview

3.1 Project Summary

An independent security assessment of Lido Finance was conducted by Shieldify in collaboration with Zealynx Security, focusing on the security aspects of the application's implementation.
We performed the security assessment based on the agreed scope, following our approach and methodology. Based on our scope and our performed activities, our security assessment revealed 1 Medium, and 1 Low severity security issues.

3.2 About Lido Finance

Website : Lido Finance
Docs: Documentation, Additional Resources

Lido is the leading liquid staking solution - providing a simple way to get rewards on your digital tokens. By staking with Lido your tokens remain liquid and can be used across a range of DeFi applications, getting extra rewards.

3.3 Audit Scope

The code under review is composed of multiple smart contracts written in Solidity language and includes 3000 nSLOC- normalized source lines of code (only source-code lines).
The core contracts on the scope are the following:

- CMS

4. Audit Methodology:

Approach

During our security assessments, we uphold a rigorous approach to maintain high-quality standards. Our methodology encompasses thorough unit and fuzz testing and meticulous manual security reviews.

Throughout the smart contract audit process, we prioritize the following aspects to uphold excellence:

1. **Code Quality:** We diligently evaluate the overall quality of the code, aiming to identify any potential vulnerabilities or weaknesses.
2. **Best Practices:** Our assessments emphasize adherence to established best practices, ensuring that the smart contract follows industry-accepted guidelines and standards.
3. **Documentation and Comments:** We meticulously review code documentation and comments to ensure they accurately reflect the underlying logic and expected behavior of the contract.

5. Severity Classification

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

6. Executive Summary

Summary of Findings

Vulnerability	Severity
[L-1] Unchecked arithmetic overflow in bond lock amount accumulation	Low
[L-2] Fee-on-transfer tokens can cause accounting discrepancies in asset recovery	Low

7. Findings

7.1. Low Findings

[L-1] Potential Overflow on _lock() Method

Vulnerability Description

There is a potential for overflow in the _lock() function from the CSBondLock contract. The unchecked block in _lock() is problematic because it adds the new amount to any existing locked amount. Even if individual calls don't use extremely large values, repeated calls could potentially lead to an overflow over time.

Impact

The lockBondETH() function in CSModule (which calls _lock() in CSBondLock) is indeed restricted to onlyCSM, meaning it can only be called by the CSM contract itself. This significantly reduces the risk of malicious exploitation. However, there is still theoretical risk even in unintentional circumstances.

Location of Affected Code

File[CSBondLock.sol](https://github.com/lidofinance/community-staking-module/blob/8ce9441dce1001c93d75d065f051013ad5908976/src/abstract/CSBondLock.sol)

```
function _lock(uint256 nodeOperatorId, uint256 amount) internal {
  ...
  unchecked {
      if ($.bondLock[nodeOperatorId].retentionUntil > block.timestamp) {
          amount += $.bondLock[nodeOperatorId].amount;
      }
      _changeBondLock({
          nodeOperatorId: nodeOperatorId,
          amount: amount,
          retentionUntil: block.timestamp + $.bondLockRetentionPeriod
      });
  }
}
```

Recommendation

Consider applying either of the following mitigations:

- Add an explicit check for overflow:

```
if ($.bondLock[nodeOperatorId].retentionUntil > block.timestamp) {
    require(amount <= type(uint256).max - $.bondLock[nodeOperatorId].amount, "Lock amount too high");
    amount += $.bondLock[nodeOperatorId].amount;
}
```

- Add an upper limit to the amount that can be locked, based on realistic maximum values for the protocol:

```
if ($.bondLock[nodeOperatorId].retentionUntil > block.timestamp) {
+   require(amount <= MAX_LOCKABLE_AMOUNT, "Lock amount exceeds maximum");
    amount += $.bondLock[nodeOperatorId].amount;
}
```

[L-2] Fee-on-Transfer Tokens Can Cause Accounting Discrepancies in Asset Recovery Functions

Vulnerability Description

The `AssetRecovererLib` library, which is used by `CSAccounting`, `CSFeeDistributor`, and potentially other contracts inheriting from `AssetRecoverer`, contains a `recoverERC20()` function that is vulnerable to fee-on-transfer token accounting issues.

This function assumes that the full amount specified will be transferred, which may not be the case for tokens that implement a fee-on-transfer mechanism.

```
function recoverERC20(address token, uint256 amount) external {
    IERC20(token).safeTransfer(msg.sender, amount);
    emit IAssetRecovererLib.ERC20Recovered(token, msg.sender, amount);
}
```

Impact

If a fee-on-transfer token is recovered using this function, the actual amount transferred will be less than the amount specified. This discrepancy could lead to:

- Incorrect accounting of recovered assets
- Potential loss of funds if the contract's balance is used for future calculations or operations
- Inconsistencies between on-chain state and off-chain records
- Misleading event emissions, as the emitted amount may not reflect the actual transferred amount

While the impact is generally low due to the restricted access of these functions (only callable by a recoverer role), it still presents a risk, especially if these contracts interact with or recover a wide range of tokens in the future.

Proof of Concept:

1. Assume a fee-on-transfer token that takes a 1% fee on each transfer.
2. The contract has a balance of 1000 of these tokens.
3. A recoverer calls recoverERC20 with an amount of 1000.
4. Only 990 tokens are actually transferred due to the fee.
5. The ERC20Recovered event is emitted with an amount of 1000, which is incorrect.

Location of Affected Code

File:[AssetRecovererLib.sol](https://github.com/lidofinance/community-staking-module/blob/8ce9441dce1001c93d75d065f051013ad5908976/src/lib/AssetRecovererLib.sol)

```
function recoverERC20(address token, uint256 amount) external {
    IERC20(token).safeTransfer(msg.sender, amount);
    emit IAssetRecovererLib.ERC20Recovered(token, msg.sender, amount);
}
```

Recommendation

To mitigate this issue, implement a balance check before and after the transfer to determine the actual amount transferred:

```
function recoverERC20(address token, uint256 amount) external {
    uint256 balanceBefore = IERC20(token).balanceOf(address(this));
    IERC20(token).safeTransfer(msg.sender, amount);
    uint256 balanceAfter = IERC20(token).balanceOf(address(this));
    uint256 actualAmountTransferred = balanceBefore - balanceAfter;
    emit IAssetRecovererLib.ERC20Recovered(token, msg.sender, actualAmountTransferred);
}
```