shieldify

Lido

Community Staking Module

SECURITY REVIEW

Date: 28 Aug 2024

CONTENTS

1. About Shieldify

Positioned as the first hybrid Web3 Security company, Shieldify shakes things up with a unique subscription-based auditing model that entitles the customer to unlimited audits within its duration, as well as top-notch service quality thanks to a disruptive 6-layered security approach. The company works with very well-established researchers in the space and has secured multiple millions in TVL across protocols, also can audit codebases written in Solidity, Vyper, Rust, Cairo, Move and Go.

Learn more about us at shieldify.org.

2. Disclaimer

This security review does not guarantee bulletproof protection against a hack or exploit. Smart contracts are a novel technological feat with many known and unknown risks. The protocol, which this report is intended for, indemnifies Shieldify Security against any responsibility for any misbehavior, bugs, or exploits affecting the audited code during any part of the project's life cycle. It is also pivotal to acknowledge that modifications made to the audited code, including fixes for the issues described in this report, may introduce new problems and necessitate additional auditing.

3. About Lido and Lido's Community Staking Module

Lido is a liquid-staking protocol built to make staking as secure and decentralized as possible. As a family of open-source, peer-to-system software tools, Lido is currently deployed and functioning on the Ethereum, Solana, and Polygon blockchain networks.

The Lido protocol is governed by the Lido DAO – an independent collection of contributors, staking enthusiasts and governance participants.

Lido's Community Staking Module is the first permissionless one. To make CSM more attractive to the community stakers, the following features were introduced:

- EL rewards and MEV are smoothened with the other modules (e.g. the Curated Module) so CSM Node Operators could potentially gain more stable rewards that are closer to the average MEV ones;
- A reasonably low bond is targeted for Node Operators so it can cover more prospective operators;
- ETH (stETH) is the only token for bond and rewards without any involvement of other assets;
- Node Operators are provided with more friendly UX and pay less gas fees for on-chain operations;
- Node Operators are supposed to gain more rewards than vanilla solo staking;

An extensive overview of Lido'Community Staking Module can be checked here

4. Risk Classification

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

4.1 Impact

- **High** – results in a significant risk for the protocol's overall well-being. Affects all or most users
- **Medium** – results in a non-critical risk for the protocol affects all or only a subset of users, but is still unacceptable
- **Low** – losses will be limited but bearable – and covers vectors similar to griefing attacks that can be easily repaired

4.2 Likelihood

- **High** – almost certain to happen and highly lucrative for execution by malicious actors
- **Medium** – still relatively likely, although only conditionally possible
- **Low** – requires a unique set of circumstances and poses non-lucrative cost-of-execution to rewards ratio for the actor

5. Security Review Summary

The security review was conducted over a span of 23 days and was part of Shieldify's Private Pool competitions. Overall, the code is written to very high standards and implements very solid security practices. A total of 26 researchers participated and identified 2 Medium and 2 Low severity issues.

5.1 Protocol Summary

Project Name	**Lido Finance – Community Staking Module**
Repository	lidofinance/community-staking-module
Type of Project	Community Staking
Audit Timeline	23 days
Review Commit Hash	8ce9441dce1001c93d75d065f051013ad5908976
Fixes Review Commit Hash	N/A

5.2 Scope

The following smart contracts were in the scope of the security review:

File	nSLOC
src/CSModule.sol	1033
src/abstract/CSBondLock.sol	114
src/abstract/CSBondCurve.sol	158
src/abstract/AssetRecoverer.sol	20
src/abstract/CSBondCore.sol	206
src/CSAccounting.sol	340
src/CSFeeOracle.sol	115

src/CSVerifier.sol	259
src/lib/SigningKeys.sol	157
src/lib/proxy/OssifiableProxy.sol	53
src/lib/base-oracle/BaseOracle.sol	268
src/lib/base-oracle/HashConsensus.sol	685
src/lib/TransientUintUintMapLib.sol	36
src/lib/NOAddresses.sol	136
src/lib/utils/PausableUntil.sol	76
src/lib/utils/Versioned.sol	37
src/lib/QueueLib.sol	152
src/lib/UnstructuredStorage.sol	23
src/lib/SSZ.sol	214
src/lib/AssetRecovererLib.sol	65
src/lib/ValidatorCountsReport.sol	20
src/lib/GIndex.sol	87
src/lib/Types.sol	35
src/CSEarlyAdoption.sol	40
src/CSFeeDistributor.sol	128
Total	**4457**

6. Findings Summary

The following number of issues have been identified, sorted by their severity:

- **Medium** issues: **2**
- **Low** issues: **2**

ID	Title	Severity
[M-01]	`CsFeeOracle` Can Be Initialized By Anyone Due To Front-Running	Medium
[M-02]	Loss of Rewards Due to Hardcoded Zero Address as a `_refreral`	Medium
[L-01]	Fee-on-Transfer Tokens Can Cause Accounting Discrepancies in Asset Recovery Functions	Low
[L-02]	Potential Overflow on `_lock()` Method	Low

7. Findings

[M-01] `CsFeeOracle` Can Be Initialized By Anyone Due To Front-Running

Submitted By

Pelz_Dev

Severity

Medium Risk

Description

The `initialize()` function within the `CsFeeOracle.sol` contract is vulnerable to front-running attacks because it is an external function with no access control. This function allows any external account to call it and initialize the contract with arbitrary parameters, potentially resulting in unauthorized initialization.

Impact

An attacker could front-run the intended owner and initialize the contract with their own parameters. This could lead to unauthorized control over the contract's administrative functions, misdirecting fees, altering consensus mechanisms, or other critical aspects of the contract's behaviour.

Location of Affected Code

File: CSFeeOracle.sol#L86

```
function initialize(
  address admin,
  address feeDistributorContract,
  address consensusContract,
  uint256 consensusVersion,
  uint256 _avgPerfLeewayBP
) external {
if (admin == address(0)) revert ZeroAdminAddress();

_grantRole(DEFAULT_ADMIN_ROLE, admin);

BaseOracle._initialize(consensusContract, consensusVersion, 0);
/// @dev _setFeeDistributorContract() reverts if zero address
_setFeeDistributorContract(feeDistributorContract);
_setPerformanceLeeway(_avgPerfLeewayBP);
}
```

Recommendation

Implement access control for the `initialize` function to ensure that only authorized accounts can call it.

Team Response

Acknowledged, it won't be fixed for the time being.

[M-02] Loss of Rewards Due to Hardcoded Zero Address as a `_refreral`

Submitted By

atharv_181

Severity

Medium Risk

Description

Node Operators won't receive referral rewards for depositing ETH into Lido. When staking ETH with Lido and depositing `stETH` into the bond, the process calls `CSBondCore::_depositETH()` to deposit ETH into Lido. However, since the referral address is passed as zero, no referral rewards are earned.

The Lido Rewards Share Program started a referral initiative to increase participation in staking via Lido. Participants can refer others to Lido, earning a share of rewards based on the staked amounts of those they refer.

Participants can earn referral rewards proportional to the staked amount of those they refer. The exact earnings depend on how much ETH the referred users stake and the duration they keep their assets staked.

Let's say prior to enrollment, the amount of ETH staked through Participant A's products and services is 60,000 ETH and the Committee agreed to a 25% Rewards Share Percentage.

If as a result of the 60,000 ETH stake a middleware usage fee in the amount of 1,000 ETH is programmatically collected, Participant A would receive 250 ETH as their share (25% of 1000 ETH). This reward-sharing continues for each ETH staked, starting from the date each ETH is staked, and generally lasts for 12 months. If Participant A decides to stake additional ETH during this period, each new ETH will also earn rewards for 12 months from the date it was staked.

Hardcoding the referral address to `address(0)`, the reward amount is lost.

More info about the referral program - Lido Referral Program

Transaction with Referral Address Provided - Transaction Link

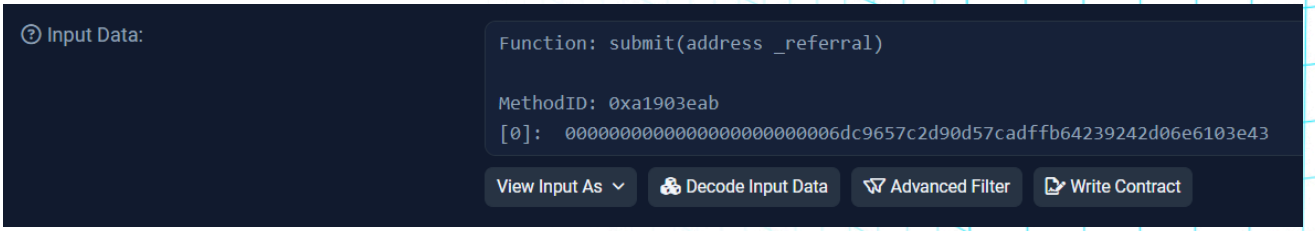

Figure 1: Input Data

Location of Affected Code

File: CSBondCore.sol#L125

```
function _depositETH(address from, uint256 nodeOperatorId) internal {
    if (msg.value == 0) return;
    uint256 shares = LIDO.submit{ value: msg.value }({
        _referal: address(0)
    });
    _increaseBond(nodeOperatorId, shares);
    emit BondDepositedETH(nodeOperatorId, from, msg.value);
}
```

Recommendation

Referrals should be distributed to all node operators proportionally to bond shares.

Team Response

Acknowledged, it won't be fixed for the time being.

[L-01] Fee-on-Transfer Tokens Can Cause Accounting Discrepancies in Asset Recovery Functions

Submitted By

bloqarl

Severity

Low Risk

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.

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

```
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);
}
```

Team Response

Acknowledged, the amount in the event properly reflects the amount of tokens transferred from the contract.

[L-02] Potential Overflow on `_lock()` Method

Submitted By

bloqarl

Severity

Low Risk

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

```
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;
}
```

Team Response

Will be fixed, the `unchecked` block will be removed to ensure native solidity revert in case of overflow.

shieldify

Thank you!