PUBLIC

Code Assessment

of the Lido

Smart Contracts

August 23, 2022

Produced for

LIDO

by

CHAINSECURITY

Contents

1 Executive Summary

Dear Lido Team,

Thank you for trusting us to help Lido with this security audit. Our executive summary provides an overview of subjects covered in our audit of the latest reviewed contracts of Lido according to Scope to support you in forming an opinion on their security risks.

Lido implements a staking protocol that allows users to stake their ETH while maintaining liquidity. In addition, it allows users to receive rewards for their staked ETH without running validator nodes. The inverse is true for node operators - they can run validator nodes and receive rewards without having to supply ETH themselves.

The most critical subjects covered in our audit are functional correctness, the trust model, and security of user funds. Security regarding all the aforementioned subjects is high.

The general subjects covered are gas efficiency and access control. Some improvements to gas efficiency can be made.

The documentation provided was detailed and helpful in understanding the complexity of the system.

In summary, we find that the codebase provides a high level of security.

It is important to note that security audits are time-boxed and cannot uncover all vulnerabilities. They complement but don't replace other vital measures to secure a project.

The following sections will give an overview of the system, our methodology, the issues uncovered and how they have been addressed. We are happy to receive questions and feedback to improve our service.

Sincerely yours,

ChainSecurity

1.1 Overview of the Findings

Below we provide a brief numerical overview of the findings and how they have been addressed.

Critical-Severity Findings	0
High-Severity Findings	0
Medium-Severity Findings	0
Low-Severity Findings	9
• Risk Accepted	4
• Acknowledged	5

2 Assessment Overview

In this section, we briefly describe the overall structure and scope of the engagement, including the code commit which is referenced throughout this report.

2.1 Scope

The assessment was performed on the source code files inside the Lido repository based on the documentation files. The table below indicates the code versions relevant to this report and when they were received.

V	Date	Commit Hash	Note
1	19 May 2022	08436ce13d67501fa723169c1dc69fe47b90cde4	Initial Version

The following files were in scope. Various compiler versions were used for different contracts.

Contracts using compiler version `0.4.24`:

- MemUtils.sol
- Pausable.sol
- StakeLimitUtils.sol
- NodeOperatorsRegistry.sol
- LidoOracle.sol
- ReportUtils.sol
- Lido.sol
- StETH.sol

Contracts using compiler version `0.6.12`:

- WstETH.sol

Contracts using compiler version `0.8.9`:

- CompositePostRebaseBeaconReceiver.sol
- DepositSecurityModule.sol
- ECDSA.sol
- LidoExecutionLayerRewardsVault.sol
- OrderedCallbacksArray.sol
- SelfOwnedStETHBurner.sol

The version `0.4.24` was chosen in order to keep compatibility with the Aragon DAO framework.

2.1.1 Excluded from scope

All libraries and contracts imported and used in the contracts, for example Aragon and BytesLib, are not part of this review. Only the contracts listed above are in scope.

2.2 System Overview

This system overview describes the initially received version (Version 1) of the contracts as defined in the Assessment Overview.

Furthermore, in the findings section, we have added a version icon to each of the findings to increase the readability of the report.

As ETHEREUM is implementing a new consensus layer based on Proof-of-stake (Beacon Chain), users are able to stake their ETH in batches of 32 and run validators to receive staking rewards. The staked tokens are locked in a deposit contract and cannot be transferred. At the time of this writing, the tokens also cannot be withdrawn.

Lido offers a staking protocol that allows users to stake their ETH on ETHEREUM's Beacon Chain while maintaining liquidity. It allows any amount of ETH to be sent to the protocol in exchange for staked ETH (stETH) tokens which can be freely traded while they accrue interest. Furthermore, running and maintaining validator nodes is delegated to trusted third party node operators, allowing users to stake their ETH with no effort.

Staked ETH tokens accrue value over time using a rebase mechanism. In periodic updates, rewards gained by the validators are captured by external oracles and published to the Lido smart contracts, increasing all users' balances equally. Currently, only the main staking rewards are accrued but as soon as ETHEREUM merges the execution layer with the beacon chain, additional rewards are collected due to transaction priority fees and Miner Extractable Value (MEV).

Due to network and node operator fees, as well as the fact that rewards are socialized between all participants and there is a waiting time between Beacon chain deposits and completely running validator nodes, rewards are slightly reduced in comparison to individual staking. However, once introduced, execution layer rewards will be reinvested and thus have a compounding effect which is not easily possible in individual staking.

As withdrawing from the ETHEREUM deposit contract is disabled until the Shanghai upgrade, which is planned to follow the Merge, Lido does not support exchanging stETH back to ETH at the time of this writing.

2.2.1 *StETH*

The `StETH` contract represents an `ERC20` token with rebase mechanism. User balances are internally represented as `shares` and their actual balances are a product of the total shares and the amount of ETH per share currently held by active validators, pending validators, and buffered in the Lido contract. As the amount of ETH per share increases due to rewards, the user balances also increase proportionally.

2.2.2 *WstETH*

To mitigate problems with smart contracts that cannot handle rebasing `ERC20` tokens, Lido offers another token contract that represents user balances in constant values. stETH can be sent to the `WstETH.wrap` function in order to utilize this behavior. An `unwrap` function allows to easily convert the tokens back to stETH.

While stETH value remains constant, wstETH value increases over time relative to the stETH rebase amounts.

2.2.3 *Lido*

`Lido` is the main contract of the protocol. It extends the abstract `StETH` contract and features the following functions:

- `submit` allows users to deposit their ETH and receive stETH in return. The deposits are buffered and added to the ETHEREUM deposit contract in bulk by the `depositBufferedEther` function using signing keys that have been previously submitted to the `NodeOperatorsRegistry`.
- `depositBufferedEther` deposits the buffered ETH supplied by users to the ETHEREUM deposit contract. It is called by the `DepositSecurityModule` contract which enables certain security guarantees.
- `handleOracleReport` updates user balances by periodically receiving reports from the `LidoOracle`. During execution, the function also withdraws ETH from the `LidoExecutionLayerRewardsVault` to which node operators are sending the execution layer rewards.

2.2.4 NodeOperatorsRegistry

Lido requires external node operators that run and maintain the validator nodes. The `NodeOperatorsRegistry` contains these operators along with signing keys that are used for deposits in the ETHEREUM deposit contract. For each 32 ETH deposit, one signing key by one operator as well as the protocol's withdrawal key is used. The operator can then set up a validator with the signing key and start validating as soon as the key gets approved.

Signing keys can be added and removed by the DAO as well as node operators themselves. For each operator, a total of the submitted signing keys and the already used signing keys is maintained in the ledger. All operators that still have signing keys left are equally matched to deposits during `Lido.depositBufferedEther` calls.

2.2.5 LidoOracle

`LidoOracle` receives reports about the state of all validators associated with Lido. External oracle providers call the function `reportBeacon` periodically (currently approx. once per day) to inform the protocol about the number of validators as well as the total validator balances. Once a quorum is reached by different oracle providers agreeing on the same state, the report is sent to Lido (`handleOracleReport`) and to some other listeners (currently, only `SelfOwnedStETHBurner`).

2.2.6 DepositSecurityModule

The `DepositSecurityModule` is used to counter some attack vectors of the deposit mechanism. Since the current Beacon Chain specification allows to top up a position with another deposit even when the withdrawal key is different, special measures have to be taken so that nobody can deposit with the same public key but a different withdrawal key before the actual Lido deposit is executed.

`DepositSecurityModule.depositBufferedEther` calls `Lido.depositBufferedEther` after a committee has verified the keys about to be used for depositing have not been used for a malicious pre-deposit. The function is then called with the current Merkle-Root of the ETHEREUM deposit contract (and some other values) to make sure the transaction reverts if state changes happen before the actual on-chain execution.

2.2.7 LidoExecutionLayerRewardsVault

All execution layer rewards (transaction priority fees and MEV) are sent to the `LidoExecutionLayerRewardsVault` by node operators. When oracle reports are sent to `Lido`, the rewards are reinvested (up to a certain limit per call) into the ETHEREUM deposit contract.

2.2.8 SelfOwnedStETHBurner

The DAO can send stETH to the `SelfOwnedStETHBurner` to distribute value among all holders of stETH by burning the sent tokens. The contract allows to send stETH either for covering losses or for other reasons. The difference is just in the accounting: Both functions `requestBurnMyStETHForCover` and `requestBurnMyStETH` stake stETH in the contract and every time the oracle report is pushed by the `LidoOracle`, the contract burns all stETH up to a certain threshold.

2.2.9 Roles & Trust Model

Lido relies on certain external trusted entities:

- **Oracles** submit reports of the current state of the validators on the Beacon Chain.
- **Node operators** submit public signing keys to be used for deposits and create validator nodes running with these signing keys.
- **Deposit Security Committee** verifies the state of the signing keys and the deposit contract to make sure no malicious pre-deposits have been made.

Oracles and the Deposit Security Committee each consist of multiple trusted entities of which the majority has to agree on certain states in order for state changing actions to be performed. Node operators are approved by the DAO and have to be completely trusted.

Other state changing functions (e.g., parameter changes) are controlled by the DAO.

3 Limitations and use of report

Security assessments cannot uncover all existing vulnerabilities; even an assessment in which no vulnerabilities are found is not a guarantee of a secure system. However, code assessments enable the discovery of vulnerabilities that were overlooked during development and areas where additional security measures are necessary. In most cases, applications are either fully protected against a certain type of attack, or they are completely unprotected against it. Some of the issues may affect the entire application, while some lack protection only in certain areas. This is why we carry out a source code assessment aimed at determining all locations that need to be fixed. Within the customer-determined time frame, ChainSecurity has performed an assessment in order to discover as many vulnerabilities as possible.

The focus of our assessment was limited to the code parts defined in the engagement letter. We assessed whether the project follows the provided specifications. These assessments are based on the provided threat model and trust assumptions. We draw attention to the fact that due to inherent limitations in any software development process and software product, an inherent risk exists that even major failures or malfunctions can remain undetected. Further uncertainties exist in any software product or application used during the development, which itself cannot be free from any error or failures. These preconditions can have an impact on the system's code and/or functions and/or operation. We did not assess the underlying third-party infrastructure which adds further inherent risks as we rely on the correct execution of the included third-party technology stack itself. Report readers should also take into account that over the life cycle of any software, changes to the product itself or to the environment in which it is operated can have an impact leading to operational behaviors other than those initially determined in the business specification.

4 Terminology

For the purpose of this assessment, we adopt the following terminology. To classify the severity of our findings, we determine the likelihood and impact (according to the CVSS risk rating methodology).

- *Likelihood* represents the likelihood of a finding to be triggered or exploited in practice
- *Impact* specifies the technical and business-related consequences of a finding
- *Severity* is derived based on the likelihood and the impact

We categorize the findings into four distinct categories, depending on their severity. These severities are derived from the likelihood and the impact using the following table, following a standard risk assessment procedure.

Likelihood	Impact		
	High	Medium	Low
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

As seen in the table above, findings that have both a high likelihood and a high impact are classified as critical. Intuitively, such findings are likely to be triggered and cause significant disruption. Overall, the severity correlates with the associated risk. However, every finding's risk should always be closely checked, regardless of severity.

5 Findings

In this section, we describe our findings. The findings are split into these different categories:

- Design: Architectural shortcomings and design inefficiencies
- Trust: Violations to the least privilege principle

Below we provide a numerical overview of the identified findings, split up by their severity.

Critical-Severity Findings	0
High-Severity Findings	0
Medium-Severity Findings	0
Low-Severity Findings	9

- Gas Inefficiencies Acknowledged
- Inconsistent Event Order Risk Accepted
- LidoOracle Initialized With Wrong Epoch Acknowledged
- Malicious Node Operators Risk Accepted
- No Events on Important State Changes Risk Accepted
- Typing Errors Acknowledged
- Unused Imports Acknowledged
- memcpy Optimizations Acknowledged
- sharesAmount Can Be Zero Risk Accepted

5.1 Gas Inefficiencies

Design Low Version 1 Acknowledged

We have found some gas inefficiencies that could be optimized:

- `Lido` saves several contract addresses (e.g., the ETH deposit contract) in the storage. Since `Lido` is upgradeable, the mentioned variables could be exchanged with constants or immutables that can be updated with a Proxy upgrade to save storage reads on various interactions.
- `Lido.handleOracleReport` updates the `BEACON_VALIDATORS_POSITION` even when the amount of validators has not changed.
- `StakeLimitUtils.calculateCurrentStakeLimit` performs stake limit calculations even when the `prevStakeBlockNumber` is the current block number. It could just return `prevStakeLimit` in that case.
- `NodeOperatorsRegistry.removeSigningKeys` and `removeSigningKeysOperatorBH` are inefficient if `0 < (totalSigningKeys - 1) - _index < (_amount - 1)` as more swaps from the last to the current position are performed than necessary.
- `NodeOperatorsRegistry._removeSigningKey` assigns the new `totalSigningKeys` value to a value that is loaded from storage, while the same value has already been loaded from storage before (`lastIndex`).

- `NodeOperatorsRegistry.assignNextSigningKeys` calculates `stake + 1 > entry.stakingLimit` while `stake >= entry.stakingLimit` would be sufficient.
- `NodeOperatorsRegistry.assignNextSigningKeys` finds the operator with the smallest stake with the statement `bestOperatorIdx == cache.length || stake < smallestStake`. This can be simplified to `stake < smallestStake` by initially setting `smallestStake` to the maximum value of `uint256`.
- `NodeOperatorsRegistry._storeSigningKey` and `_loadSigningKey` load signatures by iterating over the words of the signature and loading them from the memory location at `add(_signature, add(0x20, i))` on every iteration. This can be simplified to `i` by setting the loop variable to `signature + 32` and the execution condition to `i <= signature + SIGNATURE_LENGTH`.
- `LidoOracle` pushes reports to the `CompositePostRebaseBeaconReceiver` which pushes reports to the `SelfOwnedStETHBurner`. Since the `SelfOwnedStETHBurner` is currently the only receiver, this indirect route is not necessary.
- `SelfOwnedStETHBurner._requestBurnMyStETH` uses `Lido.transfer` and calculates the share amount by calling `Lido.getSharesByPooledEth`. This second call could be avoided by using the `transferShares` function.

Acknowledged:

Lido states:

> Thank you for the suggestions, we will take them into consideration for the next protocol upgrade.

5.2 Inconsistent Event Order

Design | Low | Version 1 | Risk Accepted

The order in which the *Transfer* and *TransferShares* events are emitted is inconsistent. In the `transferShares` function in `StETH`, the *TransferShares* event is emitted first. In all other cases, *Transfer* is emitted first.

Note also that these events are always emitted after calling the `_transferShares` function. To avoid the duplication of emitted events and reduce code size, it would also be possible to emit the events within the `_transferShares` function itself.

Risk accepted:

This change is scheduled for the next update.

5.3 LidoOracle Initialized With Wrong Epoch

Design | Low | Version 1 | Acknowledged

In the `initialize` function of the `LidoOracle`, the `expectedEpoch` is set as follows:

```
uint256 expectedEpoch = _getFrameFirstEpochId(0, beaconSpec) + beaconSpec.epochsPerFrame;
```

However, `_getFrameFirstEpochId` will always return 0 here. So the first expected epoch is set to `beaconSpec.epochsPerFrame`. However, it would make little sense for a member to report an epoch before the contract was deployed. Instead, the `expectedEpoch` could be set to an epoch which occurs after the contract is deployed, for example using `_getCurrentEpochId`.

Acknowledged:

The `initialize` function can't be called again on the Lido contract, which is already deployed. Therefore this is only an issue if a redeployment becomes necessary.

5.4 Malicious Node Operators

Trust **Low** **Version 1** **Risk Accepted**

Node operators are trusted entities in the Lido ecosystem. They are responsible for correctly running the validators as well as distributing MEV rewards to the contract. They can only be incentivized to behave decently so it is possible that certain economic opportunities could incentivize them to behave maliciously.

For example, it can be hard to verify the amount of MEV rewards node operators generate with the nodes they are running. Malicious node operators could choose to not distribute these rewards but instead pocket them themselves.

Furthermore, and most importantly, node operators have no ownership of the ETH that are locked in their validators. This means that whatever incentive they have to run the nodes benevolently could be offset by a more financially lucrative incentive. One example could be a short position in stETH that becomes profitable. As the staked ETH are not owned by the operators, this is very much possible due to slashing as can be seen in the following example (assuming the Merge has already happened and according to current spec):

- A malicious node operator executes 2 attestations to the same target on all of their controlled validator nodes. At the time of this writing, single validators run up to ~8,000 nodes of the ~400,000 nodes currently on the Beacon Chain).
- Each node gets slashed by 1 ETH, reducing the amount of ETH in the protocol by ~8,000 or ~0.1% of Lido's total supply.
- After 18 days, the validators get slashed again based on the amount of validators that have been slashed in the previous 16 days: Each validator loses ~1.8 ETH.
- In total, the supply of Lido drops by ~0.5%.

If 2 node operators collude, the total supply drops by ~1.7%. If 3 operators collude, it drops by ~3.6%.

Depending on the market reaction, the value of stETH could decrease dramatically following these events, making a decently sized short position in stETH (or more likely wstETH) profitable.

Risk accepted:

Lido states:

> The risk is mitigated by maintaining healthy validators set with monitoring and DAO governance processes. There is a set of policies and management actions:
>
> - onboarding new NOs to decentralize further;
> - limiting the stake amount per single NO;

- developing dashboards and tools to monitor network pasticipation performance (now open-sourced https://github.com/lidofinance/ethereum-validators-monitoring)
- developing dashboards and tools to monitor MEV and priority fee distribution (approaching testing stage for the upcoming Merge)

Despite the fact that Ethereum staking is not delegation-friendly, Lido DAO already has on-chain levers to address malicious NO behavior: excluding them from the new stake, disabling fee distribution, excluding them from the set, considering penalties on other chains if applicable, and so on. Once and if withdrawal-credentials initiated exits are implemented, there will appear additional on-chain enforcement mechanics which would allow building more permissionless schemes for the validators set.

5.5 No Events on Important State Changes

Design | Low | Version 1 | Risk Accepted

The `DepositSecurityModule` does not emit events on the following important state changes:

1. When the owner calls `setLastDepositBlock`.
2. When `depositBufferedEther` is called.

Risk accepted:

Lido states:

- `setLastDepositBlock` will be used only if re-deploy is needed, so we may add the event for future versions.
- `depositBufferedEther` emits the Unbuffered event in the Lido contract which is still enough for indexers, though, will consider the change if an upgrade is needed.

5.6 Typing Errors

Design | Low | Version 1 | Acknowledged

- `NodeOperatorsRegistry._loadOperatorCache` returns an error message with a typing error: `INCOSISTENT_ACTIVE_COUNT`.
- `Lido._setProtocolContracts` emits the event `ProtocolContactsSet`.

Acknowledged:

This will be fixed in the next major protocol upgrade.

5.7 Unused Imports

Design | Low | Version 1 | Acknowledged

`Lido` imports `SafeMath64.sol` which is not used in the contract.

Acknowledged:

This will be fixed in the next major protocol upgrade.

5.8 `memcpy` Optimizations

Design | Low | Version 1 | Acknowledged

The `memcpy` function in the `MemUtils` library is quite critical for gas costs. It is called by `copyBytes`, which is in turn called from within nested loops in `NodeOperatorsRegistry.assignNextSigningKeys`. The function itself also contains a loop, for a total of three nested loops.

While it is already written in assembly, it can be optimized further. First, let's take a look at the loop.

```
for { } gt(_len, 31) { } {
    mstore(_dst, mload(_src))
    _src := add(_src, 32)
    _dst := add(_dst, 32)
    _len := sub(_len, 32)
}
```

As it stands, there are three variables which are modified per loop iteration. Ideally, one would only change one variable per iteration, and use a loop bound based on this variable. However, this change would add additional overhead outside the loop, which may not pay off in general. Currently, the loop is only executed 1-3 times per call, as the `_len` parameter is only ever 48 or 96. One may also consider creating functions specifically for copying byte arrays of length 48 and 96, as this would allow a complete unrolling of the loop.

After the loop, the following code is executed:

```
if gt(_len, 0) {
    let mask := sub(shl(1, mul(8, sub(32, _len))), 1) // 2 ** (8 * (32 - _len)) - 1
    let srcMasked := and(mload(_src), not(mask))
    let dstMasked := and(mload(_dst), mask)
    mstore(_dst, or(dstMasked, srcMasked))
}
```

- As `_len` is a `uint256`, it is more efficient to just check the condition `if _len {`.
- The mask could also be written as `shr(0xff..ff, shl(3, _len))` or `shr(not(0), shl(3, _len))`.

Acknowledged:

Lido states:

> We decided to leave the assembly code as is to prevent possible peculiarities.

5.9 `sharesAmount` Can Be Zero

Design | Low | Version 1 | Risk Accepted

Due to rounding errors, the value returned by `getSharesByPooledEth` can be zero. In the function `_submit` in `Lido`, the check `sharesAmount == 0` is made. This is assumed to hold either on the first deposit, or in the case of a complete slashing. However, this can also occur if rounding errors lead to `getSharesByPooledEth` returning 0. Thus, a user would receive a disproportionate amount of shares, as they would get a 1:1 rate of ETH to StETH, despite the share value being lower. Note that with the current state of the live contracts, this can only occur if `msg.value == 1`.

Risk accepted:

Lido is aware of rounding errors, however chooses not to fix them as they are difficult to correct without sacrificing gas efficiency or backwards compatibility.

6 Notes

We leverage this section to highlight further findings that are not necessarily issues. The mentioned topics serve to clarify or support the report, but do not require an immediate modification inside the project. Instead, they should raise awareness in order to improve the overall understanding.

6.1 Deposits Can Be Blocked by Node Operators

Note Version 1

`DepositSecurityModule` requires the `keysOpIndex` to not change between creation of a `depositBufferedEther` transaction and its execution:

```
uint256 onchainKeysOpIndex = INodeOperatorsRegistry(nodeOperatorsRegistry).getKeysOpIndex();
require(keysOpIndex == onchainKeysOpIndex, "keys op index changed");
```

Since the `keysOpIndex` can be changed by node operators using `addSigningKeysOperatorBH` or `removeSigningKeysOperatorBH`, malicious node operators can delay depositing even when they are not activated. The only way to counter this problem is to change the `rewardAddress` of such node operators.

6.2 No Quorum Sanity Checks

Note Version 1

`LidoOracle` and `DepositSecurityModule` allow the addition of members / guardians and the setting of a quorum that has to be reached by these entities. The quorum can however be set to any value (except for 0 in the case of `LidoOracle`) independently of the number of members / guardians.