PUBLIC

Code Assessment

# of the Staking Router Smart Contracts

February 3, 2023

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 Staking Router according to Scope to support you in forming an opinion on their security risks.

Lido implements a modularization of the current Lido system. This allows Lido to introduce various different staking modules with the Node Operators Registry being just one of these modules. The Staking Router contract is responsible for appropriately distributing the 32 ETH batches and the accumulated rewards among the different modules. To that end, Lido implemented an allocation algorithm.

The most critical subjects covered in our audit are the security of the funds stored in the system, the distribution of the buffered ETH and the rewards to the various modules, the management of the modules, the node operators and the public keys of the validators, the correctness of the allocation algorithm, and the low-level handling of the storage and access control. The most important issue we uncovered relates to incorrectly trimming the array containing the address of the reward recipients. Moreover, we uncovered an important correctness issue in the `MemUtils.memcpy` function which, however, has no impact in the current implementation. All the aforementioned issues have been addressed.

The general subjects covered are upgradeability, the efficiency of the implementation, the documentation and unit testing. We find the security in all aforementioned areas high. The documentation is comprehensive, and the unit testing is extensive.

In summary, we find that the codebase provides a high level of security. Unfixed issues reported by ChainSecurity in previous reports are omitted in this one.

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	1
• Code Corrected	1
Medium-Severity Findings	2
• Code Corrected	2
Low-Severity Findings	10
• Code Corrected	7
• Risk Accepted	2
• Acknowledged	1

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 Staking Router 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	3 January 2023	64661e9e4b93452c6736bfc8d728bed924fe8c40	Initial Version
2	23 January 2023	601161106a8c9832c966deb9dc5773b075ed766b	Second Version
3	31 January 2023	9d1179834c720fd65e05411f384e2dcdd969a4c5	Third Version
4	2 February 2023	d820fddb0b298088db2a4d76fbc6007fdada908d	Fourth Version

For the solidity smart contracts, the compiler versions used by the contracts in scope are `0.4.24` and `0.8.9`.

The contracts in scope under the `contracts/` directory are:

- `0.4.24/`:
  - `BytesLib.sol`
  - `Math64.sol`
  - `Pausable.sol`
  - `SigningKeysStats.sol`
  - `StakeLimitUtils.sol`
  - `nos/NodeOperatorsRegistry.sol`
  - `Lido.sol`
  - `StETH.sol`
  - `StETH.sol`
- `0.8.9/`:
  - `BeaconChainDepositor.sol`
  - `DepositSecurityModule.sol`
  - `LidoExecutionLayerRewards.sol`
  - `StakingRouter.sol`
- `common/lib`:
  - `Math256.sol`
  - `MinFirstAllocationStrategy.sol`

2.1.1 Excluded from scope

Excluded from scope are all the contracts explicitly not mentioned in scope. Moreover, we don't include in the scope all the contracts from the `AragonOS` library. Many of the actions of the contracts in the system can only be executed by authorized users. We trust these users to act to the interest of the system and never maliciously. Finally, the non-fixed issues reported in previous assessments by ChainSecurity of older versions of the system are omitted.

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.

Lido implements a modular extension of the Lido system. In previous versions, validators could register on the *NodeOperatorsRegistry*. Lido now wants to support different validator sets with different implementations of the registry. Thus, they introduced the *Staking Router*.

2.2.1 Staking Router

The *Staking Router* is essentially a module registry. The modules are stored in an array, i.e., each module id is associated with an index of this array. A module can be in one of the following 3 states:

- `Active`: The module is allowed to receive deposits and rewards.
- `DepositsPaused`: The module is not allowed to receive deposits but still receives rewards.
- `Stopped`: The module does not receive deposits or rewards.

The *Staking Router*'s interface includes but is not limited to the following functions:

- `addModule`: The Module manager introduces a new module and sets its parameters. The module will be assigned the next available id. The module's status will be set to `Active`.
- `updateStakingModule`: The Module manager updates the parameters of the module.
- `setStakingModuleStatus`: The Module manager can arbitrarily update the status of the module.
- `pauseStakingModule/resumeStakingModule`: The module pauser can pause/resume any module that wasn't already paused/active. Note that there is no dedicated function to stop a module.
- `deposit`: When Lido decides to deposit some ETH to any of the modules, the call goes through the *Staking Router* and in particular the `deposit` function. During this call batches of the 32 ETH are deposited to the deposit contract of the Ethereum Beacon Chain on behalf of the various modules. The allocation of the batches is determined by the allocation algorithm (described later). The deposit only happens for one specific module which is determined by the caller of the `DepositSecurityModule.depositBufferedEther` function. The system then queries the module to return a batch of public keys and signatures to be submitted to the deposit contract.
- `getStakingRewardDistribution`: determines how Lido is going to distribute the received rewards. The rewards are distributed when the Oracle report is updated (roughly once per day).

2.2.2 NodeOperatorsRegistry

Currently, this is the only module available in the Lido system. This is the registry of the node operators. A node operator can handle multiple validator nodes. The 32 ETH batches are distributed to the node operators according to the allocation algorithm discussed later. More details about the *NodeOperatorsRegistry* are discussed in another report.

2.2.3 Lido

This is the core contract of the system. In this contract the available ETH (buffered ETH) to be deposited for the validators is stored. The most important entry points of the contract are:

- `submit`: Any user can call this function submitting some ETH for which they receive stETH ("Staked Ether") in return.
- `deposit`: It can only be called from the `DepositSecurityModule`. Since the amount of ETH to be staked is stored in the Lido contract, it is up to Lido to determine what amount to transfer to the *Staking Router* and eventually to the specific module.
- `handleOracleReport`: It updates the number of beacon validators known to the system and the balance of Lido in the beacon chain. Moreover, it updates and distributes the amount of rewards to the modules, the treasury, and the end users.

2.2.4 DepositSecurityModule

The most important entry point of the contract is `depositBufferedEther`. It is called by any user who has collected a sufficient number of guardian signatures. During the call, sanity checks are performed, and eventually `Lido.deposit` is called.

2.2.5 Reward Distribution and Fees

The system earns ETH rewards for staking. The rewards are the sum of the excessive ETH amount on the beacon chain and the execution layer rewards. The fees are shared between the treasury, the reward recipient of each module and the end users. Both the treasury fees and the module fees for a module are calculated with regard to the total contribution of the module to the total active keys it has. In other words, if a module contributes 1% of the total active keys of the system, then only 1% of the defined module fee and the treasury fee will be deducted. The remaining amount remains in the Lido system until it can be deposited into the Deposit contract.

2.2.6 Allocation algorithm

The algorithm distributes 32 ETH batches to buckets. The algorithm accepts 3 inputs, the current allocation of the buckets, the capacity of each bucket and the number of available batches to be allocated. The allocation algorithm consists of two steps:

1. Starting from the smallest module, the allocation algorithm reserves sufficient ETH to whichever of the below conditions comes first:

    - the module catches up to the next module,
    - all ready validators are covered or,
    - the target share is reached.

2. If two or more modules are equal in size and still have ready unused keys and some room until the target share, deposit ether is allocated evenly between them until reaching the capacity or there is no more ether to allocate.

2.2.7 Trust Model and Roles

The roles defined by the Staking Router are the following:

- Module Manager: They are allowed to add a module, set the staking module status to active and inactive and update the staking module parameters.
- Module Pauser: They are allowed to pause a module, i.e., call `StakingRouter.pauseStakingModule`. This is the `DepositSecurityModule`.
- Module Resumer: They are allowed to resume a module, i.e., call `StakingRouter.resumeStakingModule`. This is the `DepositSecurityModule`.
- Withdrawal Credentials Setter: They are allowed to set the withdraw credentials using `StakingRouter.setWithdrawalCredentials`. These credentials are used when the system deposits to the Ethereum staking contract.

The roles defined by the Node Operators Registry are the following:

- Adder: They are allowed to add new node operators to the registry.
- Activator/Deactivator: They are allowed to activate/deactivate a non-active/active node operator.
- Name/Reward Setter: They are allowed to set a name/reward-address for a specific operator. Each node operator may define a different name/reward setter.
- Staking Limit Setter: They are allowed to set the staking limit for a specific node operator and to a specific value.
- (Unsafe) Exited Validators Key Count Setter: They are allowed to (unsafely) set the number of exited nodes for *any* operator.
- Ready to Deposit Keys Invalidator: They are allowed to invalidate the pending keys for all the node operators.
- Validators' Keys Requester: This should be the Staking Router. It updates the state of the registry relevant to the public keys of node operators for which the 32 ETH have been deposited to the Deposit contract.
- Signing Keys Manager: They are allowed to add or remove singing keys to or from a specific node operator. Each node operator may define a different manager for that task.

All the aforementioned roles are considered trusted by the system and assumed to act maliciously.

2.2.8 Version 3

In the third iteration of the system, the access control for the adding or removing signing keys to/from the `NodeOperatorsRegistry` has been further unified.

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 any open findings. Findings that have been resolved have been moved to the Resolved Findings section. The findings are split into these different categories:

- Design: Architectural shortcomings and design inefficiencies
- Correctness: Mismatches between specification and implementation

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	3

- Excessive Memory Allocations Risk Accepted
- Rounding Errors Risk Accepted
- Unnecessary Reduction of Vetted Key Count Acknowledged

5.1 Excessive Memory Allocations

Design Low Version 1 Risk Accepted

The `slice` function of the *BytesLib* library is used in many places. In some cases, it is used inside a loop. This allocates new memory on every loop iteration, even if the memory is no longer needed after the iteration finishes. Hence, it results in a much larger allocation of memory than is actually needed.

For example, we can examine the following loop from the `_addSigningKeys` function of the *NodeOperatorsRegistry*:

```
for (uint256 i = 0; i < _keysCount; ++i) {
    bytes memory key = BytesLib.slice(_publicKeys, i * PUBKEY_LENGTH, PUBKEY_LENGTH);
    require(!_isEmptySigningKey(key), "EMPTY_KEY");
    bytes memory sig = BytesLib.slice(_signatures, i * SIGNATURE_LENGTH, SIGNATURE_LENGTH);

    _storeSigningKey(_nodeOperatorId, totalSigningKeysCount, key, sig);
    totalSigningKeysCount = totalSigningKeysCount.add(1);
    emit SigningKeyAdded(_nodeOperatorId, key);
}
```

Here, the memory is only temporarily needed, it holds the values copied from the `_publicKeys` and `_signatures` arrays. As soon as the loop iteration finishes, the `key` and `sig` arrays are no longer needed. However, the memory they used remains allocated.

Instead, the arrays could be declared outside the loop, and the values copied into them on each iteration. This would cut the total memory usage of the `_addSigningKeys` function in half. As memory has a quadratic cost, this could significantly reduce transaction costs.

The same optimization could be applied to the `_makeBeaconChainDeposits32ETH` function of the *BeaconChainDepositor* contract, which also uses `BytesLib.slice` in a loop.

Similarly, the `getSigningKeys` and `_loadAllocatedSigningKeys` functions in the *NodeOperatorsRegistry* call `_loadSigningKey` in a loop. This allocates new memory in each iteration, despite each value only being needed for the duration of a single iteration.

Changes in Version 3

In Version 3, The declarations of the bytes arrays in `_addSigningKeys`, `getSigningKeys` and `_loadAllocatedSigningKeys` were moved outside the loop. However, they are still re-allocated in every loop iteration. `_makeBeaconChainDeposits32ETH` was not changed.

For `_addSigningKeys` and `_makeBeaconChainDeposits32ETH`, the issue is that `BytesLib.slice` allocates a *new* bytes array every time it's called. In the case of `getSigningKeys` and `_loadAllocatedSigningKeys`, the culprit is the call to `_loadSigningKey`, which also allocates a *new* bytes array. Hence, the memory allocations still occur and the transaction costs remain the same.

Risk accepted

Lido responded:

> Will be fixed in the next major protocol upgrade.

5.2 Rounding Errors

Design Low Version 1 Risk Accepted

1. In the `_getKeysAllocation` function, a number of `_keysToAllocate` is provided. However, it's not guaranteed that all the keys will be allocated, as rounding errors may lead to an incomplete allocation of the keys. For example, with two modules each with a `targetShare` of 50%, a `_keysToAllocate` of 3 results in both modules being allocated just one key, as their `targetKeys` values will be rounded down to 1.
2. The `getRewardsDistribution` calculates a `perValidatorReward` as the total rewards divided by the total number of validators. Then it multiplies it by the number of validators per node operator. However, this leads to excessive rounding errors, as the division is performed before the multiplication. Instead, the multiplication should be performed first in order to reduce the amount of reward that is not distributed due to rounding errors.

Risk accepted:

Lido responded:

> This is an expected behavior and impact is tolerable. Will be fixed in the next major protocol upgrade.

5.3 Unnecessary Reduction of Vetted Key Count

Design Low Version 1 Acknowledged

When deleting a key in the *NodeOperatorsRegistry*, the `vettedKeysCount` is set to the index of the deleted key (if the deleted key is vetted). This is done because deleting a key actually swaps it with the last key, then deletes it. Hence, if the last key is swapped to a smaller index, it might otherwise become vetted. However, if the last key has also already been vetted, this could be unnecessary - instead the vetted key count could simply be reduced by one.

Acknowledged

Lido acknowledges the issue and states the following:

> Thank you for this finding. We won't fix this issue in current version but will address it later.

6 Resolved Findings

Here, we list findings that have been resolved during the course of the engagement. Their categories are explained in the Findings section.

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

Critical-Severity Findings	0

High-Severity Findings	1

- Staking Rewards Incorrect Trimming Code Corrected

Medium-Severity Findings	2

- Dummy Iterations Can Be Avoided Code Corrected
- MemUtils.memcpy Mask Is Wrong Code Corrected

Low-Severity Findings	7

- Code Redundancies Code Corrected
- Incorrect Comments Code Corrected
- Interface Issues Code Corrected
- Issues With Events Code Corrected
- Non-existent Modules Are Active by Default Code Corrected
- Sanity Checks Missing Code Corrected
- Visibility Can Be Reduced Code Corrected

6.1 Staking Rewards Incorrect Trimming

Correctness High Version 1 Code Corrected

In the *StakingRouter*, the `getStakingRewardsDistribution` assigns fees to recipient addresses. It then trims the resulting arrays so that there are no recipients that receive an amount of zero. However, this trimming does not work correctly.

The `recipients` and `moduleFees` arrays are filled left-to-right, using the same index as the `modulesCache`. This means that the "empty" entries, where the module fees are zero, occur non-deterministically throughout the array. Hence, when the last entries are trimmed, some modules that would otherwise receive fees are not being returned. Likely, the intent was to use the `rewardedModulesCount` as an index for these arrays, so that modules which receive no fee would not increment the index.

As such, the results returned by `getStakingRewardsDistribution` have the potential to be very incorrect, distributing far less fees than intended, and only distributing fees to the recipients "lucky" enough to be stored early in the `modulesCache` array.

Code corrected

The issue was fixed by using the `rewardedModulesCount` as an index for the resulting arrays, thereby ensuring that the modules receiving rewards are stored contiguously at the start of the returned arrays. Thus, the trimming is now done correctly.

6.2 Dummy Iterations Can Be Avoided

Design Medium Version 1 Code Corrected

The *MinFirstAllocationStrategy* features an algorithm which allocates items to buckets with different capacities. In some cases, the algorithm may do many unnecessary dummy iterations, which each allocate a single item to a bucket. For example:

1. Let's start with the scenario where we have 5 empty buckets, each with a very large capacity.
2. Now, call the `allocate` function, with an `allocationSize` of 24.
3. In the first five iterations, each bucket will be allocated 4 items, as `24 / 5` is rounded down to 4.
4. Now, an additional four "dummy" iterations have to be performed, which allocate an additional single item to the first four buckets.

These dummy iterations could be avoided, for example, by rounding up instead of down when the `allocationSize` is not divisible by the number of best candidates. This would result in the first four iterations allocating 5 items each, with the last iteration having only four items left and allocating them all to the last bucket.

Note that when a deposit occurs, this algorithm is run *once for each staking module*. Hence, as this algorithm is executed many times, reducing the number of iterations could reduce business costs, especially when more staking modules are added in the future.

Code corrected:

The allocation algorithm was updated. Now, when there are more than one best candidates, the ceil of the division of the `allocationSize` by `bestCandidatesCount` is used.

```
allocated = Math256.min(
    bestCandidatesCount > 1 ? Math256.ceilDiv(allocationSize, bestCandidatesCount) : allocationSize,
    Math256.min(allocationSizeUpperBound, capacities[bestCandidateIndex]) - bestCandidateAllocation
);
```

6.3 MemUtils.memcpy Mask Is Wrong

Correctness Medium Version 1 Code Corrected

The `memcpy` function in *MemUtils* copies a segment of memory from one location to another. It does so in 32-byte chunks. However, if the length of the segment to copy is not divisible by 32, it does an additional copy with a masked value so that no memory values are overwritten when they shouldn't be. However, this is done incorrectly.

```
function memcpy(uint256 _src, uint256 _dst, uint256 _len) internal pure {
    assembly {
        // while al least 32 bytes left, copy in 32-byte chunks
        for { } gt(_len, 31) { } {
            mstore(_dst, mload(_src))
            _src := add(_src, 32)
            _dst := add(_dst, 32)
            _len := sub(_len, 32)
        }
        if gt(_len, 0) {
            // read the next 32-byte chunk from _dst, replace the first N bytes
            // with those left in the _src, and write the transformed chunk back
            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))
        }
    }
}
```

The `mask` value is calculated incorrectly. The `shl` instruction for Solidity assembly is specified as follows:

> `shl(x, y)`: Logical shift left of y by x bits.

Hence, the mask is not shifting `1` to the left by some amount. Instead, `mul(8, sub(32, _len))` is being shifted to the left by 1 bit. As a result, the mask being calculated will always copy at least 30 bytes (since `2*8*(32-1)-1<512`), with the remaining two bytes being partially or not at all copied, depending on the exact value of `_len`. In the case of `_len % 32 == 16`, exactly 31 bytes are copied, meaning an additional 15 bytes after the end of the memory segment are copied to the destination.

This is the case when dealing with public keys. In the `_loadAllocatedSigningKeys` and `getSigningKeys` functions of the *NodeOperatorsRegistry*, when the public keys are copied, the length will be 48 and hence not divisible by 32. So additional bytes are copied.

However, it happens to be the case the when `_loadSigningKey` is called, the allocated memory for the public key is directly followed by the signature. Hence, the 15 extra bytes which are copied are the first 15 bytes from the length field of the signature's bytes array. Because this length is always 3, these bytes are guaranteed to be zero.

Additionally, as the destination of the memory copy is filled left-to-right, for all but the last copy operation the extra bytes that are written are immediately overwritten by the following value. Only the last copy writes 15 bytes of zeroes past the end of the `publicKeys` bytes array. But again, due to the order of memory allocation, this array is followed in memory by the `signatures` bytes array. So, the extra bytes happen to overlap with the length field of the `signatures` array. As the length of this array is (hopefully) less than `2 ** (17 * 8)`, the extra bytes that are overwritten are already set to zero anyway.

All in all, this means that the `memcpy` function works correctly, but *only due to the order of memory allocations*, which happen to guarantee that the extra bytes which are copied to and from are always zero.

Code corrected

The argument order of the `shl` expression in the mask calculation was changed.

```
let mask := sub(shl(mul(8, sub(32, _len)), 1), 1)
```

6.4 Code Redundancies

Design Low Version 1 Code Corrected

In the current implementation there are a few code redundancies which can be improved. In particular:

1. In `NodeOperatorsRegistry.activateNodeOperator`, the `ACTIVE_OPERATORS_COUNT_POSITION` is increased using `SafeMath`. However, it is impossible for this value to overflow.

2. In `NodeOperatorsRegistry.distributeRewards`, `recipients.length` is guaranteed to equal `shares.length` as it constructed by `getRewardsDistribution` that way. Hence, the relevant assertion will always be true.
3. In `MinFirstAllocationStrategy.allocateToBestCandidate`, the condition `allocationSize == 0` is checked after the first loop, despite not being affected by it. Instead, this condition could be checked at the start of the function. Alternatively, as the function is only ever called from `allocate`, it is actually guaranteed the `allocationSize` is never equal to zero. Hence, the check could also be omitted fully.
4. In `StakingRouter.deposit`, it is ensured that `keysToDeposit` does not exceed `maxDepositableKeys`, which is calculated by querying the *Lido* contract and querying its buffered Ether amount. However, it is already guaranteed that `_maxDepositsCount` fulfills this condition, as only the *Lido* contract can call this function, and it performs checks before making this call.
5. The *StakingRouter* has two functions, `_getStorageStakingModulesMapping` and `_getStorageStakingIndicesMapping`, which are always called with the same constant arguments. Instead of passing these arguments to the functions, the functions could inline the values in order to reduce unnecessary complexity and allow for more optimal bytecode.
6. In the *StakingRouter*'s `deposit` function, a `StakingRouterETHDeposited` event is emitted with the parameter `_getStakingModuleIdByIndex(stakingModuleIndex)`. However, the staking module's ID was already passed as an argument to the function, so it doesn't need to be looked up. Instead, `_stakingModuleId` could be used as a parameter to this event.
7. `NodeOperatorsRegistry.addNodeOperator` will always emit a `NodeOperatorAdded` event with 0 staking limit. As there are no occasions where the event is emitted with a non-zero staking limit, this field contains essentially no information and is redundant.
8. In `NodeOperatorsRegistry._getSigningKeysAllocationData`, the cells of `nodeOperatorIds` and `exitedSigningKeysCount` arrays are potentially written redundantly in the case where `depositedSigningKeys` is equal to `vettedSigningKeysCount`, as they may be overwritten in a following iteration.
9. In `NodeOperatorsRegistry.getRewardsDistribution`, a non-empty `recipients` array could be returned even though the `shares` array contains only 0s. This corner case can happen if all the node operators are active but all their validators have exited.
10. `NodeOperatorsRegistry.getValidatorsKeysStats` makes redundant use of `SafeMath` even though, should the desired `exitedKeys <= depositedKeys <= vettedKeys` invariant hold, the subtractions should never underflow.

Code corrected:

Most of the redundancies reported have been fixed.

1. `SafeMath` is not used any more.
2. The corresponding assertion has been removed.
3. The check was moved at the beginning.

5. Now, the two functions do not accept any argument.
6. `_stakingModuleId` is now used.

8. The array elements are now written after the `depositedSigningKeysCount == vettedSigningKeysCount` check.

10. SafeMath is no longer used in the `getValidatorsKeysStats` function.

Acknowledged:

4. Lido acknowledges the additional cost associated with the redundant check.

7. This is intended behavior in order to keep backwards compatibility.

9. The returning of empty `shares` entries is intended behavior.

6.5 Incorrect Comments

Correctness Low Version 1 Code Corrected

The code contains some inaccurate comments:

1. The doc comment for the `_stakingModuleAddress` of the `addModule` function is as follows:

   ```
   /**
    * ...
    * @param _stakingModuleAddress target percent of total keys in protocol, in BP
    * ...
    */
   ```

   This is incorrect and describes a different parameter.

2. Point 4. in the comment in `MinFirstAllocationStrategy.allocateToBestCandidate` is not updated. In the current implementation, `DivCeil(allocationSize, count)` is used instead of integer division.

3. In `NodeOperatorsRegistry`, `addSigningKeys` enforces the same access control as `addSigningKeysOperatorBH`. The same holds for `removeSigningKeys` the same as `removeSigningKeysOperatorBH`. This implies that the `BH` versions of the functions are pointless and can be considered deprecated. However, they are not marked as such.

Code corrected:

Comments 1. and 2. have been fixed. 3. is left as is, the functions exist for backwards compatibility and are hence not deprecated.

6.6 Interface Issues

Correctness Low Version 1 Code Corrected

The interfaces for certain contracts were defined multiple times, once for version `0.4.24` and once for version `0.8.9`. However, some of the duplicate interfaces have outdated or clashing definitions:

1. The `0.8.9` *ILido* interface defines the functions `updateBufferedCounters` and `getLastReportTimestamp`, which are not implemented by the *Lido* contract.

2. The `0.4.24` *IStakingRouter* interface defines a function `getStakingModuleMaxDepositableKeys` which takes a parameter with the incorrect name `_stakingModuleId`. The `0.8.9` interface was changed to take the `_stakingModuleIndex` as a parameter instead.

Moreover, there are some mismatches between the functions defined in the interface and the ones actually implemented. The `0.4.24/ILido` interface defines the following function signatures:

```
function getFee() external view returns (uint16 feeBasisPoints);
function getFeeDistribution() external view returns (uint16 modulesFeeBasisPoints, uint16 treasuryFeeBasisPoints);
```

However, the *Lido* contract implements these functions with different return types:

```
function getFee() public view returns (uint96 totalFee) {
    // ...
}

function getFeeDistribution() public view returns (uint96 modulesFee, uint96 treasuryFee) {
    // ...
}
```

Code corrected

Regarding mismatches between different versions:

1. The unimplemented functions were removed from the `0.8.9` *ILido* interface.
2. The `0.8.9` version of the *IStakingRouter* interface was changed, the parameter for the `getStakingModuleMaxDepositableKeys` is now `_stakingModuleId` instead of `_stakingModuleIndex`.

Regarding mismatch of the *Lido* contract between interface and implementation:

The interface was corrected to match the implementation.

6.7 Issues With Events

Design | Low | Version 1 | Code Corrected

In the implementation there are some issues with the events:

- In `NodeOperatorsRegistry.addNodeOperator`, two events are emitted. Namely, `NodeOperatorAdded(id)` and `NodeOperatorAdded(id, _name, rewardAddress, 0)`. The second event contains strictly more information than the first one.
- In `NodeOperatorsRegistry.activateNodeOperator`, the `NodeOperatorActiveSet` event is not emitted even though it is defined.
- In `NodeOperatorsRegistry.deactivateNodeOperator`, the `NodeOperatorActiveSet` event is not emitted.
- In `NodeOperatorsRegistry.distributeRewards`, the `RewardsDistributed` event uses `idx` which is not the id of the node operator but the index of the operator in the `recipients` array. Note that this index can vary based on the status of the various operators. Moreover, the event is emitted even if `shares[idx]` is 0.
- In `DepositSecurityModule._setGuardianQuorum`, the `GuardianQuorumChanged` event will be emitted even if the quorum has not changed.
- In `StakingRouter.setStakingModuleStatus`, the `StakingModuleStatusSet` event will be emitted even if the status of the module has not changed.

Code corrected

The issues have been remedied in the following ways:

- NodeOperatorsRegistry.addNodeOperator only emits the event with more information. The definition of the other event was removed from the IStakingModule interface.
- NodeOperatorsRegistry.activateNodeOperator now emits a NodeOperatorActiveSet event.
- NodeOperatorsRegistry.deactivateNodeOperator now emits a NodeOperatorActiveSet event.
- The RewardsDistributed event was changed to take the rewards address as a parameter. NodeOperatorsRegistry.distributeRewards emits the event with recipients[idx] as the address. Additionally, the transfer and event emission are omitted if the shares amount is 0.
- DepositSecurityModule._setGuardianQuorum no longer emits an event if the quorum has not changed.
- StakingRouter.setStakingModuleStatus now reverts if the new status is the same as the old one.

6.8 Non-existent Modules Are Active by Default

Design Low Version 1 Code Corrected

Should StakingRouter.getStakingModuleByIndex be called with a non-existent index, it will return an empty StakingModule. the StakingModule struct has a status field of type IStakingRouter.StakingModuleStatus. Since the default value of the enum is Active, any uninitialized staking module will be considered active. This could be a potential problem for other contracts calling getStakingModuleByIndex.

Code corrected

The StakingRouter.getStakingModuleByIndex function was removed. Additionally, the StakingRouter.getStakingModuleMaxDepositableKeys function was modified to take an ID as an argument instead of an index. However, it is worth noting that the default value of the StakingModuleStatus is still Active.

6.9 Sanity Checks Missing

Design Low Version 1 Code Corrected

In the current implementation, a few sanity checks are missing. More specifically:

- In NodeOperatorsRegistry.removeSigningKeys, _fromIndex and _keysCount are checked to be less than UINT64_MAX. However, these checks do not suffice, as it's important that the sum of the two inputs is also less than UINT64_MAX (and strictly greater than 0). Note that in any of these cases, SafeMath calculations taking place later will revert.
- In NodeOperatorsRegistry.removeSigningKeys, _keysCount is not checked to be non-zero. This is not symmetric to the addSigningKeys case where such a check is performed. Moreover, should _keysCount equal 0 the call will have no effect, yet the ValidatorsKeysNonce will increase.
- In theory, the NodeOperatorsRegistry can hold up to UINT64_MAX different keys for each operator ranging from indices 0 up to UINT64_MAX - 1. This means that _index cannot be equal to UINT64_MAX. However, this is allowed from removeSigningKey and

removeSigningKeyOperatorBH. The inequality of the sanity check in these cases should be strict.

- In StakingRouter.addModule, the _stakingModuleAddress argument is not checked to be non-zero. As this address cannot be changed later, it may be important to sanity-check the value before setting it.
- In StakingRouter.addModule, the _name argument is not sanitized to be at most 64 bytes while such a check exists for the name argument in the NodeOperatorsRegistry.
- In NodeOperatorsRegistry._addSigningKeys, while the public keys are checked to be non-empty, there is no such a check for the signatures.

Code corrected

NodeOperatorsRegistry.removeSigningKeys was changed in the following ways:

- The sum of _fromIndex and _keysCount is checked to be less than UINT64_MAX. Note that the check for _fromIndex < UINT64_MAX is now redundant, as the sum of two unsigned integers is always greater or equal than both summands (assuming no overflow occurs). Note also that the conversion uint256(_fromIndex) is redundant.
- _keysCount is checked to be non-zero in the _removeUnusedSigningKeys function, in order to not emit redundant events.

The *NodeOperatorsRegistry* was modified so that all checks comparing the index to UINT64_MAX are now done with a strict inequality.

StakingRouter.addModule was modified as follows:

- It was renamed to addStakingModule in Version 2.
- The _stakingModuleAddress is checked to be non-zero.
- The _name parameter is checked to be between 1 and 32 bytes in length.

Acknowledged

Regarding the missing check for signatures, Lido states

> Will be fixed in the next major protocol upgrade.

6.10 Visibility Can Be Reduced

Design Low Version 1 Code Corrected

Visibility of certain functions can be restricted in order to reduce gas costs. The following functions' visibility could be changed from public to external:

Lido:

1. getCurrentStakeLimit
2. getBeaconStat
3. getFee
4. getFeeDistribution

StETH:

1. getTotalShares
2. sharesOf

3. `transferShares`

StakingRouter:

1. `getKeysAllocation`

Lastly, it may make sense to reduce the `allocateToBestCandidate` function of the *MinFirstAllocationStrategy* library to `private` instead of `internal`.

Code corrected:

The visibility of the functions was changed as suggested, except for the `allocateToBestCandidate` function.

7 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.

7.1 Discrepancy in Event Argument

Note Version 1

When Lido.submit is called, a Transfer event is emitted by the _emitTransferAfterMintingShares function. This event contains a field which corresponds to the amount of ETH used to mint the respective amount of shares the user received. It is likely, however, that the amount is not exactly the same as the one sent by the user since the calculation of the shares amount to be minted can incur some rounding errors. For example, assume that a user submits 1234 wei, the total number of shares is 10 and pooled ether amount is 1000 wei. Then the user will mint 1234*10/1000 = 12 shares and the argument in the Transfer event will be 1200.

7.2 Misleading Calculation in _deleteSigningKey

Note Version 1

The _deleteSigningKey function deletes a (key, signature) pair from storage by writing zeroes to the bytes where they were stored.

```
function _deleteSigningKey(uint256 _nodeOperatorId, uint256 _keyIndex) internal {
    uint256 offset = _signingKeyOffset(_nodeOperatorId, _keyIndex);
    for (uint256 i = 0; i < (PUBKEY_LENGTH + SIGNATURE_LENGTH) / 32 + 1; ++i) {
        assembly {
            sstore(add(offset, i), 0)
        }
    }
}
```

The number of bytes to be zeroed out is calculated in the expression (PUBKEY_LENGTH + SIGNATURE_LENGTH) / 32 + 1. However, this calculation relies on the fact that PUBKEY_LENGTH is *not* divisible by 32, and that SIGNATURE_LENGTH is. The calculation is misleading as it is not correct in the general case, where the lengths could assume any value.

7.3 Staking Modules Can DOS

Note Version 1

The staking router makes external calls to the staking modules in several situations. For example, when making a deposit, it calls into all active staking modules to query their validator key stats. This means that a revert by any single module prevents the staking router from depositing to any other module.

While the modules are trusted in general, it should be ensured in their development that these calls do not propagate the potential denial of service attack one level deeper, e.g. to the users running validators in the case of some distributed validator technology.