CERTORA

Lido V2 Smart Contract Security Analysis Report and Formal Verification Properties

Date of report release: 27 April 2023

Table of Contents

Summary

This document describes the specification and verification of the new **Lido V2 protocol with enabled Ethereum withdrawals and modular Staking Router architecture** using the Certora Prover and manual code review findings. The work was undertaken from **01 February 2023** to **25 April 2023**. The commits reviewed and run through the Certora Prover were e575177 (Lido 2.0 beta2), 2bce10d (Lido 2.0 beta3), and the final commit e45c4d6 (Lido 2.0 rc2).

The following contracts list is included in the ***on-chain scope***:

`contracts/0.4.24/lib/Packed64x4.sol`
`contracts/0.4.24/lib/SigningKeys.sol`
`contracts/0.4.24/lib/StakeLimitUtils.sol`
`contracts/0.4.24/nos/NodeOperatorsRegistry.sol`
`contracts/0.4.24/oracle/LegacyOracle.sol`
`contracts/0.4.24/utils/Pausable.sol`
`contracts/0.4.24/utils/Versioned.sol`
`contracts/0.4.24/Lido.sol`
`contracts/0.4.24/StETH.sol`
`contracts/0.4.24/StETHPermit.sol`
`contracts/0.6.11/deposit_contract.sol`
`contracts/0.6.12/interfaces/IStETH.sol`
`contracts/0.6.12/WstETH.sol`
`contracts/0.8.9/interfaces/IStakingModule.sol`
`contracts/0.8.9/lib/Math.sol`
`contracts/0.8.9/lib/PositiveTokenRebaseLimiter.sol`
`contracts/0.8.9/lib/UnstructuredStorage.sol`
`contracts/0.8.9/lib/UnstructuredRefStorage.sol`
`contracts/0.8.9/oracle/AccountingOracle.sol`
`contracts/0.8.9/oracle/BaseOracle.sol`
`contracts/0.8.9/oracle/HashConsensus.sol`
`contracts/0.8.9/oracle/ValidatorsExitBusOracle.sol`
`contracts/0.8.9/proxy/OssifiableProxy.sol`
`contracts/0.8.9/sanity_checks/OracleReportSanityChecker.sol`
`contracts/0.8.9/utils/access/AccessControl.sol`
`contracts/0.8.9/utils/access/AccessControlEnumerable.sol`
`contracts/0.8.9/utils/PausableUntil.sol`
`contracts/0.8.9/utils/Versioned.sol`
`contracts/0.8.9/BeaconChainDepositor.sol`
`contracts/0.8.9/Burner.sol`
`contracts/0.8.9/DepositSecurityModule.sol`
`contracts/0.8.9/EIP712StETH.sol`

`contracts/0.8.9/LidoExecutionLayerRewardsVault.sol`
`contracts/0.8.9/LidoLocator.sol`
`contracts/0.8.9/OracleDaemonConfig.sol`
`contracts/0.8.9/StakingRouter.sol`
`contracts/0.8.9/WithdrawalQueueERC721.sol`
`contracts/0.8.9/WithdrawalQueue.sol`
`contracts/0.8.9/WithdrawalQueueBase.sol`
`contracts/0.8.9/WithdrawalVault.sol`
`contracts/common/interfaces/IEIP712StETH.sol`
`contracts/common/interfaces/ILidoLocator.sol`
`contracts/common/interfaces/IBurner.sol`
`contracts/common/lib/ECDSA.sol`
`contracts/common/lib/Math256.sol`
`contracts/common/lib/MemUtils.sol`
`contracts/common/lib/MinFirstAllocationStrategy.sol`
`contracts/common/SignatureUtils.sol`

The contracts are written in Solidity 0.4.24, 0.6.12, and 0.8.9.

The Certora Prover demonstrated the implementation of the Solidity contracts above is correct with respect to the formal rules written by the Certora team. In addition, the team performed a manual audit of the all the Solidity contracts. During the verification process and the manual audit, the Certora Prover discovered bugs in the Solidity contracts code, as listed below.

Summary of findings

The table below summarizes the issues discovered during the audit, categorized by severity.

	Total discovered	Total fixed	Total acknowledged
Critical	2	2	0
High	5	1	4
Medium	10	7	3
Low	5	3	2
Informational	1	1	0
Total	**23**	**14**	**9**

We have also manually reviewed the following ***off-chain*** Python source files commit 833bf94, in the context of their interaction with the on-chain contracts in the scope:

```
./services/withdrawal.py
./services/exit_order.py
./services/validator_state.py
./services/bunker.py
./services/prediction.py
./services/safe_border.py
./variables.py
./providers/consensus/client.py
./providers/consensus/typings.py
./providers/keys/client.py
./providers/keys/typings.py
./providers/http_provider.py
./modules/ejector/ejector.py
./modules/ejector/typings.py
./modules/ejector/data_encode.py
./modules/submodules/oracle_module.py
./modules/submodules/typings.py
./modules/submodules/consensus.py
./modules/submodules/exceptions.py
./modules/accounting/extra_data.py
./modules/accounting/typings.py
./modules/accounting/accounting.py
./metrics/prometheus/basic.py
./metrics/logging.py
./metrics/healthcheck_server.py
./utils/validator_state.py
./utils/events.py
./utils/abi.py
./utils/types.py
./utils/slot.py
./utils/dataclass.py
./utils/blockstamp.py
./typings.py
./web3py/typings.py
./web3py/extensions/keys_api.py
./web3py/extensions/contracts.py
./web3py/extensions/lido_validators.py
./web3py/extensions/consensus.py
./web3py/extensions/tx_utils.py
./web3py/contract_tweak.py
./web3py/middleware.py
```

`./constants.py`
`./main.py`

The off-chain code is written in Python 3.11.

Main Issues Discovered

Cr-01: Withdrawal Finalization Affected by other users in batch (`DiscountFactor` issue)

Severity: Critical
Category: Logic flaw
File(s): `WithdrawalQueue.sol` in the original code [WithdrawalQueue.sol#L189]
Bug description: If there are two users in the same batch who entered the queue with different share rates, and later there is a slash (the finalization share rate has decreased for one of them, but still above the others), the finalized amount will be weighted incorrectly. This could unjustifiably lead to ETH losses for users.
Exploit scenario: When there's an expected slash, and there are several requests in the queue with a lower rate than the current rate, any user is incentivized to exit immediately (before the report of the slashing), and lose less at the expense of other users who were in the queue beforehand.
Numerical Example:
user1 has a request in the queue with a **shareRate** of 1 * 10^27 with 10^18 shares (1 ETH) and
user2 has a request in the queue with a **shareRate** of 2 * 10^27 with 10^18 shares (2 ETH). Both are in the same **finalizationBatch**.

Let's look at two cases: (with floats to make this more readable)

a. In the happy flow, there is no slashing → **shareRate** == 2 or more so
   user1 receives 1 ETH
   user2 receives 2 ETH

b. In the case where slashing occurs → the **shareRate** < 2,
   For example, **shareRate** == 1.4 (less than the average of the two shareRates), then when finalizing the batch:
   user1 will receive 0.93333 ETH
   user2 will receive 1.866666 ETH

   But if finalizing in different batches:
   user1 will receive 1 ETH (**>** 0.93333 ETH **!**)
   user2 will receive 1.4 ETH (**<** 1.866666 ETH **!**)

   Explanation:
   2 shares shareRate of 1.4 → 2 * 1.4 = 2.8 < 3,
   since
   discountFactor = _amountOfETH * E27_PRECISION_BASE / stETHToFinalize
   where
   _amountOfETH == 2.8*10^18
   stETHToFinalize == 2*10^18
   Therefore:
   discountFactor = 0.933333*10^27

In the second case (b), user1 “protects” user2 from losses, user1 even loses ETH and user2 earns ETH at user1 expense.

Implications: Users in the protocol and in the queue lose value unjustifiably. Other users gain more value than they should.
Lido’s response: ***Fixed*** in commit 336b6f3. The withdrawals finalization process was changed profoundly to remove the discount factor in favor of the share rate batch-wise calculation approach.
Certora’s response: After the issue was fixed in commit 336b6f3 with the removal of the discount factor, a related issue M-04 arose in this component.

Cr-02: Miscalculation in `PositiveTokenRebaseLimiter` can lead to an incorrect *`shareRate`*

Severity: Critical
Category: Logic/Calculation flaw
File(s): `PositiveTokenRebaseLimiter.sol`
Bug description: Due to a miscalculation in the raising and consuming functions of the rebase limit library, the limit on the amount of shares that can be burnt during the handling of the report may be larger or lower than intended. In the case where the limit is small, the consequence is loss of value to the existing users. This can be influenced by entering (via *`submit()`*), which opens up the vector to anyone with sufficient funds.
Exploit scenario: As shown in the numerical example below, first intentionally get to the presented edge case by making a deposit and withdrawal request of a large enough amount. Then after the report, buy a lot of stETH at a discount at the expense of those who were already in the protocol.
Numerical example:
Let’s assume all the vaults have enough ETH.

Initial shares: 960,000
Total available ETH in Lido: 960,000
Rebase daily limit: 0.1% implying 1.001^365 = 1.44, i.e. 44% APY

User A deposits 40,000 ETH and immediately asks to withdraw it (therefore we know there are enough funds to supply the user request)

Shares: 1,000,000
Available ETH in Lido: 1,000,000 ETH
shareRate: 1:1

Once the oracle report is submitted, the following rebase will occur:

- 1,000 ETH comes from the vault. (assuming the vault has more than 1000 ETH)

- In function *`consumeLimit()`* line 106:

_limiterState.accumulatedRebase = _limiterState.rebaseLimit; (0.001)

- Then in function *raiseLimit()* lines 78-82:
  _limiterState.rebaseLimit = 0.041

- So now in function *getSharesToBurnLimit()* in line 129, the calculation result:
  maxSharesToBurn = (1,000,000 * 0.04) / (1 + 0.04) = 38,461.54

- And also there are (40,000 - 38,461.54) = 1,538.46 shares waiting in the `Burner` to be burned later

As a result:
New shares: 1,000,000 - 38,461.54 = 961,538.46
New available ETH in Lido: 1,000,000 - 40,000 + 1,000 = 961,000 ETH
New shareRate: 961,000 / 961,538.46 = **0.99944 < 1**

Implications: A malicious user could get stETH at a discount at the expense of all other users, thus causing the others to lose funds.
Lido's response: ***Fixed*** in commit b748768 (limiter takes into account total pooled ether amount accumulated during the token rebase operations)

H-01: *NodeOperator* can DoS any new deposits via Frontrun

Severity: High
Category: Frontrun -> DoS
File(s): `NodeOperatorRegistry.sol`
Bug description: To deposit, the Guardians sign approval for a given nonce state of a staking module. Any single operator in this implementation can frontrun a deposit call to increase the nonce.
Called internally by *_increaseValidatorsKeysNonce()*, which can be called by the operator from the external *addSigningKeys(), removeSigningKey(),* or *removeSigningKeys().*
Exploit scenario: A node operator who knows that the Guardians got a sign request for a deposit could continuously add signing keys to prevent them from getting a consensus (enough signatures).
Implications: If run continuously, it would prevent new validator participation, which would decrease rewards (and thus capital efficiency).
Lido's response: *Acknowledged*. Any misbehaved node operator might be deactivated by the Lido DAO, which in particular results in an inability to add/remove signing keys by the inactive node operator (this feature was implemented in commit 0628547)

H-02: Double use of validator *(pub_key,signature)* pairs

Severity: High
Category: Logic flaw
File(s): `DepositSecurityModule.sol`, `StakingRouter.sol`, `NodeOperatorsRegistry.sol`, (Off-chain accounting oracles and Guardians)
Issue description: There could be a case where a validator's key is used twice - either via two different node operators, or because it was deposited outside of Lido. Either case can lead to a scenario where the oracles would report the state of this key twice (or once where they shouldn't have at all). This would lead to a wrong state (count more rewards than there actually are, allocate operator rewards incorrectly, etc.).
In the case of exits, the wrong state may also lead to unexpected reverts since an operator may have more exited keys than they have deposited.
Exploit scenario: Described above. Requires being a malicious node operator.
Implications: Count more rewards than there are, allocate operator rewards incorrectly, DOS some functionality.
Lido's response: *Acknowledged*. The validator's keys validation process, including keys uniqueness, happens off-chain. Given an attempt to upload an already added or pre-deposited validator key, Lido's key validation system triggers an alarm to prevent malicious keys from becoming depositable (via deactivation of the node operator or removing the malicious key). Moving a validator's key check on-chain is a desired feature, but unfortunately, it can't be implemented until the beacon chain state root is exposed on the execution layer.

H-03: Continuous DoS by reaching stake limit

Severity: High
Category: Logic flaw
File(s): `Lido.sol`, `StakeLimitUtils.sol`
Issue description: A malicious user with sufficient funds could keep the stake at its upper limit, thus keeping other users from submitting ETH to the protocol. This attack is viable since the attacker can always request the ETH back via the withdrawal mechanism. This means the amount needed to sustain such an attack for an extended period of time is large but not unrealistic.
Exploit scenario: The attacker initially stakes *maxStakeLimit* ETH, and then *maxStakeLimit/blocksPerDay* ETH every new block, thus keeping the stake at its limit at all times. The minted stETH is immediately sent back to the withdrawal queue (by creating a withdrawal request), which will eventually release the ETH back to the attacker. Lido team estimated the amount of funds needed to sustain the attack at 150k+666*23 ETH, or 226 ETH/day using flash loans.
Implications: DoS for the ETH's submission mechanism.
Lido's response: *Acknowledged*. The Lido DAO governance can change the following parameters of the protocol that stored on-chain and enforced by the smart contracts:

- max stake limit amount and its restoration speed (*Lido.setStakingLimit* requires the assigned permission `STAKING_CONTROL_ROLE`)

- minimum delay before the withdrawal request can be finalized (`OracleReportSanityChecker.setRequestTimestampMargin` requires the assigned permission `REQUEST_TIMESTAMP_MARGIN_MANAGER_ROLE`)
  Therefore, if an attack was detected via the continuous monitoring and alerting tools, changing these params (increasing the minimum delay and, optionally, max stake limit or its restoration speed) on behalf of the Lido DAO governance mitigates the issue by raising costs for the attacker and minimizes the attack impact by delaying withdrawals finalization (loss of the missed rebases for the attacker) while allowing other users to stake their funds by increasing max stake limit.

H-04: A node operator whose reward address is Lido would DoS the *extraReport*

Severity: High
Category: Logic flaw
File(s): `NodeOperatorRegistry.sol`
Issue description: If a node operator is added (via *addNodeOperator()*) whose reward address is the Lido contract address, then the *submitReportExtraDataList()* function reverts (because *transferShares()* reverts). Also, a node operator can change its reward address via *setNodeOperatorRewardAddress()*.
Exploit scenario: A new `NodeOperator` can set its `rewardAddress` to be Lido contract, thus DoSing the extra reports.
Implications: DoS the node operator rewards and penalty mechanisms, destroying their incentives.
Lido's response: ***Fixed*** in commit 03c5280.

H-05: Wrong batching in the oracle report could permanently lock funds in the `withdrawalQueue` contract

Severity: High
Category: Logic flaw
File(s): `NodeOperatorRegistry.sol`
Issue description: In the latest version released, the Lido contract trusts the oracles to provide accurate information related to the batches. Their reported data is not verified on-chain. As a result, malformed batches may cause a permanent lock of funds in the `withdrawalQueue`.
Exploit scenario: Numerical example:
There are two requests:
Request #1: ratio 1:1, 1 share,
Request #2: ratio 1:2, 1 share

The simulated shareRate is 1.5

If the oracles put both of them in the same batch (which they should not) then the *prefinalize()* function will calculate that 3 eth is needed (when only 2.5 eth is needed)

In that case, 3 eth will be sent to the withdrawalQueue with a shareRate of 1.5 and only 2.5 eth can be claimed.

As a result, 0.5 eth will be locked forever in the withdrawalQueue.
Implications: Some of the funds that should've been split among the other users would actually be sent and locked forever in the withdrawalQueue.
Lido's response: *Acknowledged*. Even if extra funds had been locked on the WithdrawalQueue contract, it wouldn't mean that funds are locked **permanently**. The WithdrawalQueue contract is upgradeable. Therefore, to mitigate the issue, it's possible to change the contract implementation in a way that allows locked funds to be added back to the protocol total pooled ether via the Lido DAO governance voting process (the proxy admin is the Lido DAO Agent contract). It was decided not to introduce the recovery implementation upfront to minimize overall code complexity and audit scope. We believe that strict on-chain validation of the batches increases the complexity and implementation risks of the finalization method dramatically (see the PoC) because needs historical token rebase data and extrema tracking for the share rate changes.
The finally chosen solution still provides guarantees that:

- everyone will be able to claim their ETH once the request is finalized
- malicious/incorrect Oracle report can't lock ETH more than the total stETH of the underlying pending batches

As was said above, the locked funds could be returned back to the protocol TVL via the WithdrawalQueue implementation upgrade as a mitigation measure. Moreover, Lido has a general long-term plan to mitigate oracle trust issues by introducing the trustless zk-based oracle implementation that will close this and other issues requiring oracles to collude when exploit them.

M-01: *ExtraData* and main *Report* are tied but can have mismatching data

Severity: Medium
Category: other
File(s): AccountingOracle.sol
Property violated: cannotSubmitNewReportIfOldWasNotProcessedFirst
Bug description: There are two reports, the main accounting report and the *extraData* report. The two reports depend on each other's timing and content, but are not as strongly tied on-chain.
Examples: the *extraData* can be skipped, the amounts reported exited validators can be incoherent, etc.
Exploit scenario: If there's a way to manipulate the oracles, such a mismatch (or the missing of a report) could happen. At that point, the report relies on the oracles being smart enough to correct this.

Implications: Incorrect reward distribution among validators, or missing it entirely in the case of missing a report. There could be issues around reward distribution fairness due to this as well.
Lido's response: *Acknowledged*. As a part of the Lido protocol, Oracles are a trusted code. Thus, a malicious quorum of Oracles is currently a case that the protocol can't handle by design. Oracles report data based on the on-chain state at the reference slot. The next reference slot is guaranteed to occur after all state changes made by the previous Oracle report. Thus, if data in the previous report is not complete, oracles will be able to calculate the proper diff and submit the complete data. In case of an error in the Oracle code, there is a function `StakingRouter.unsafeSetExitedValidatorsCount()` for manually correcting the wrong data by a DAO vote. It allows manually setting exited and stuck validator counts for any module and node operator.

M-02: Sum of exited keys and target keys may overflow

Severity: Medium
Category: DoS / unexpected revert
File(s): `NodeOperatorsRegistry.sol`
Property violated: `TargetPlusExitedDoesntOverflow`
Bug description: By setting the target validators for any node operator too high, the sum of the target validators and the exited keys count can overflow and revert due to safe math checks, leading to unexpected revert.
Exploit scenario: Calling `updateTargetValidatorsLimits()` with a relatively large number for the target limit may lead to an unexpected revert in the future if the exited keys increase for that node operator, given that at some point the operator penalty is cleared. The check for overflow is included in `_applyNodeOperatorLimits()`, which is called in most functions of the contract.
Implications: DoS and unexpected reverts in most functions for a node operator whose penalty has been cleared.
Lido's response: ***Fixed*** in commit b6281e6.

M-03: Sanity check of the exited keys count update is missing

Severity: Medium
Category: Lack of Checks -> DoS
File(s): `NodeOperatorsRegistry.sol`, `StakingRouter.sol`
Property violated: `getMaxDepositsCountRevert`, `moduleActiveValidatorsDoesntUnderflow`
Bug description: In a situation where the oracle is mistaken, either when updating the exited keys amount (whether per module or per node operator), or by calling the 'unsafe' methods by the DAO, the update may result in a false state where the exited keys count surpasses the deposited keys count.
Exploit scenario #1: The report submitted by `AccountingOracle` updates the exited keys for some staking modules and some node operators registered in certain modules (depending on the secondary report type). No explicit check exists for the new value of the

exited keys against the registered deposit keys count for any staking module, so an erroneous report can potentially update the exited keys count to a larger value than the deposited, which is a bad state of the system.
Exploit scenario #2: By calling `unsafeSetExitedValidatorsCount()` in the `StakingRouter.sol` contract, the staking module exited keys count value is updated to the value provided by the DAO input (in the `ValidatorsCountsCorrection` struct). This update is not checked against the deposited keys summary value for that staking module, which can also lead to the same bad state of the system described above.
For both scenarios above, the active key count (deposited - exited) underflows in this case and therefore reverts in `StakingRouter`, including reverting the deposit flow.
Implications: DoS and reverts in most functions.
Lido's response: Exploit scenario #1 is ***fixed*** in commit 8a9bd67. Regarding exploit scenario #2: The `unsafeSetExitedValidatorsCount()` function was introduced to restore the staking module consistency in case of failures or inaccuracies in the Oracle-reported data. The function is prefixed with `unsafe_` and restricted by a role with `UNSAFE_` prefix to raise additional attention before ever being used. The Lido V2 deployment template doesn't assign this role. It's assumed that the `unsafeSetExitedValidatorsCount()` function can be used only on behalf of the `Lido DAO Agent` contract that would have a granted role assigned before. As a consequence, the change requires an on-chain Aragon vote to enact, and the Lido governance token holders accept the associated risks of changing data inconsistently if support the vote. Worth noting that the `deposited` keys must always be greater or equal to `exited`, and the `deposited` keys counter can only increase, while DAO ensures that before and during the possible vote.

M-04: Finalization `shareRate` recovery wouldn't fully recover

Severity: Medium
Category: Logic flaw
File(s): `Lido.sol`, `WithdrawalQueue.sol`, after the first big fix, before the mini-batches fix.
Bug description: The given share rate is compared against the pre-finalization state. In a case where there's a recovery due to users in the queue with a lower rate, the recovering user would get a rate that is a bit lower than if calculated in two different batches. This contradicts the documentation, stating that users in the queue fully recover with the protocol (in reality, only a partial recovery is possible).
Exploit scenario: see the numerical example below
Numerical example: (omitting 1E18 decimals for simplicity)
Let's assume:

a. user A requested 1 share (with a `shareRate` of 1.0) and later
b. user B requested 1 share (with a `shareRate` of 2.0)
    i. At the time of finalization, the total assets' value is 6 and the total number of shares is 4, so the effective `shareRate` is 6/4=1.5.

Under the *current* implementation, if we finalize A and B together:

a. A will receive 1=min(1,1.5) ETH, and
b. B will receive **1.5**=min(1.5,2) ETH.

However, had we first finalized A, and only then finalized B (on separate batches), then:
a. A would have still received 1 ETH, but
b. B would have received **1.666**…
c. since after A is finalized, the `shareRate` becomes (6-1)/(4-1)=5/3=1.666.

Therefore, currently, B gets a worse deal because she was finalized together with A.

Implications: Users in the queue would get less ETH than expected.
Lido's response: *Acknowledged*. The behavior is intended by design, and such peculiar edge cases are acceptable. The provided example operates with unrealistically huge rebases (actual values orders of magnitude less) in contrast to the real scenarios when deviations in share rates are tiny. Possible mitigation of the described problem would have significantly increased the finalization logic and exposed additional risks due to the complexity of the iterative approach for the Oracle report construction.

M-05: Wrong storage offset used in updating stuck validators count

Severity: Medium
Category: Coding error
File(s): `NodeOperatorsRegistry L#614`, `NodeOperatorsRegistry L#623`
Property violated: `operatorPenaltyStatusAfterKeysUpdate`
Bug description: The variables were read incorrectly, putting the stuck amount into the refunded variable and the refunded amount into the stuck. This could lead to a wrong state and allow a node operator to evade reward penalties.
Exploit scenario: The wrong update could manifest in 2 ways:
(1) An operator with stuck keys could evade the penalized state since the stuck keys are updated incorrectly, or
(2) An "honest" operator could get penalized when there would be an update to the refunded keys, even if it didn't have any stuck keys.
Implications: A malicious node operator could bypass stuck penalties. Evading said penalties removes the negative incentive for griefing the system.
Lido's response: ***Fixed*** in commit e470fca.

M-06: Staking modules can have the same address

Severity: Medium
Category: Possible configuration error
File(s): `StakingRouter.sol`
Property violated: `CannotAddStakingModuleIfAlreadyRegistered`, `StakingModuleAddressIsUnique`

Bug description: There's an assumption that each staking module is different and abides by a specific interface. If there is a duplicate, each state update to one module would cause the other to become inconsistent.
Exploit scenario: calling `addStakingModule()` with an address that is already registered will create an unintended duplicate of the staking module, which can break the logic of the deposit/oracle update flow.
Implications: This could put the system into an incorrect state, which would influence rewards and operator rewards calculations, possibly confuse the accounting oracles, or more.
Lido's response: ***Fixed*** in commit 61e1ce3.

M-07: `HashConsensus` does not revoke a previous hash

Severity: Medium
Category: Logic flaw
File(s): `HashConsensus.sol`
Bug description: The contract correctly updates the votes when there's a change to the consensus members or the quorum, but if the update causes the current hash to be below the quorum it does not revoke it from the `OracleContract`.
Exploit scenario: A bad/malformed/malicious report achieves consensus, then revoked on the *hashConsensus* level. One of the oracles can still submit the report to the `OracleContract` and it will be accepted.
Implications: A single oracle can still submit the revoked report; The impact depends on the error that caused the revoke in the first place (example worst case scenario - all balances are gone).
Lido's response: ***Fixed*** in commit e618c30.

M-08: `AccountingOracle`'s "`contractVersion`" mechanism doesn't defend against intended case

Severity: Medium
Category: Logic flaw
File(s): `AccountingOracle.sol`
Bug description: When submitting the report, an oracle also needs to send a *contractVersion* which is compared to the contract's version (from `Versioned.sol`). This check is intended to defend against a report (that somehow passed consensus), but would be incorrect/inconsistent with this version of the `AccountingOracle` (e.g. only the `AccountingOracle` was updated, and some oracles didn't update yet). It indeed may help against errors and help debugging which oracles are lagging in updates, but a single malicious oracle can still send the report with the correct new version (as this constant is not part of the data that had a consensus).
Exploit scenario: Only `AccountingOracle` is upgraded, such that the encoding of the report is changed as well as some of the field's meanings; the hash, however, is still calculated like today on the raw encoding of all the data (meaning, there should be a new

hash for the same reported state). Immediately following the upgrade, most oracles are not yet updated and still vote for the hash of the old-version report. Upon reaching consensus, a single malicious oracle can craft the data that would result in the same hash that reached quorum (if the new encoding is similar enough, it's easy. If not, there's no need for the `contractVersion` here). Since the `contractVersion` is checked against a parameter, the malicious oracle can simply give the right version and pass this check.
This way a single oracle can submit a report which by its meaning did not pass the quorum. The possible damage is dependent on how and what meaning was changed via the introduced patch (it could be very minor, but also could be extreme).
Implications: Depends on what was changed in the upgrade; A report that shouldn't have been processed (its meaning didn't pass quorum) would be. This also locks the *`refSlot`* of the report.
Lido's response: *Acknowledged*. It's assumed that contract code and the underlying meaning of the data are the different things that can be changed independently (some illustrative examples: can preserve Oracle contract code but change the address of the Execution Layer vault to collect the reported data by the off-chain oracle daemon, whereas can update the contract's code for gas optimization purposes or logic flaws still having the same data and pre-processing algorithm by the off-chain oracle daemons). To handle this, there is the `consensusVersion` which is part of the data and which **has to** be changed any time the meaning of the data changes, including (but not limited to) the changes caused by the contract code upgrades. The upgrade playbook checked by the Lido DAO members upon approving an on-chain oracle code upgrade includes the step enforcing the consensus version bump upon any change in the data interpretation.
For all that, the `contractVersion` param is not designed to add any security guarantees on top of `consensusVersion`. It exists merely to improve the debuggability and traceability of upgrades. The contract version is something that has to be updated manually (the same as the consensus version), so including it in the hashed data, in addition to the consensus version, would not automatically guarantee that any data meaning change caused by a code upgrade would prevent non-upgraded or malicious oracles from submitting the wrong data, and it would require unnecessary operations for upgrading off-chain oracle setup in the cases where on-chain code was upgraded without changing the data meaning, e.g. to fix a contract weakness or optimize for gas spending.

M-09: Wrong node operator penalty extension

Severity: Medium
Category: Logic flaw
File(s): `NodeOperatorRegistry.sol`
Property violated: `operatorPenaltyStatusAfterKeysUpdate`
Bug description: Regardless of the previous state, if a node operator was in a penalized state, it will remain so and its duration will be prolonged again. For example, this is true even if there were no stuck validators and its refunded keys count was updated.
Exploit scenario: Any update to the refunded or stuck keys of a given node operator will extend its penalty time even if the stuck keys amount turns out to be smaller than the refunded keys amount. A penalty directly affects the rewards distribution given to that node operator by cutting it in half.

Implications: Unjustifiably extended penalty which reduces potential validators rewards for the penalized node operator.
Lido's response: The issue was ***fixed*** independently in commit 99ec8c0.

M-10: Incorrect tracking of summary of total keys count

Severity: Medium
Category: Coding error
File(s): `NodeOperatorRegistry.sol`
Property violated: `SumOfTotalKeysEqualsSummary`
Bug description: Inside the function *`_invalidateReadyToDepositsKeysRange()`* the total number of keys for each node operator in the loop is updated but eventually the total keys summary is not.
Exploit scenario: Any time this inner function is called, (for example when *`onWithdrawalCredentialsChanged()`* is invoked from the `StakingRouter`), the total keys summary of that staking module won't be correctly updated and won't match the sum of total keys over all node operators.
Implications: The current version of the code makes no special use of this value and doesn't export it to other contracts, so the only expected implication is inconsistency between the actual sum and the stored summary. Had this value been used in the future, it would have been erroneous.
Lido's response: ***Fixed*** in commit fbb3195.

L-01: Request ids may be removed from a set and not added to a new set

Severity: Low
Category: Lack of checks
File(s): `WithdrawalQueueERC721.sol`
Bug description: In the *`_transfer()`* function, the owner will succeed if he tries to transfer ownership of his request to himself, an action that should properly revert. Also in this case the `requestId` is removed from the owner's `requestIds uintSet` since there is no check on the return values from the `uintSet`.
Exploit scenario: transfer request to the request owner.
Implications: The owner set will not have the ID and some function returns the wrong output because of it.
Lido's response: ***Fixed*** in commit 8073c3d.

L-02: *`canDeposit()`* returning true instead of false in some cases

Severity: Low
Category: Wrong return value

File(s): `DepositSecurityModule.sol`
Bug description: *`canDeposit()`* false result:
`canDeposit()` checks whether it is possible to deposit to some staking module by checking its active status and its last deposit block. For an unregistered module (ID > count) both conditions are met by storage default (active=0 and also the last block=0).
Implications: While such a procedure is highly unlikely, If one tries to deposit into an unregistered staking module of that ID, the deposit function inside `Lido.sol` will revert because the function *`getStakingModuleMaxDepositsCount()`* will revert.
Lido's response: ***Fixed*** in commit b944985.

L-03: Over-stringent sanity check for staking router total fee

Severity: Low
Category: Exclusion of possible outcomes
File(s): `StakingRouter.sol`
Property violated: `feeDistributionDoesntRevertAfterAddingModule`
Bug description: The total fee sanity check inside *`getStakingRewardsDistribution()`* in `StakingRouter.sol` is over-stringent as it excludes cases where the fee is exactly 100%.
Exploit scenario: In the case where there is only a single staking module with active validators whose treasury fee is zero but its module fee is 100%, the calculated total fee in *`getStakingRewardsDistribution()`* will be 100% and be equal to the fee precision points value. This case will revert because of the sanity check which doesn't allow 100% fees.
Implications: The rewards distribution mechanism will revert whenever it is called for the case mentioned above.
Lido's response: ***Fixed*** in commit 1789516.

L-04: Misuse of privileged functions

Severity: Low
Category: Loss of 3rd party protocols funds
File(s): `Lido.sol`
Bug description: The privileged function *`unsafeChangeDepositedValidators`* allows a specific role to change the internal accounting of the deposited validators' amount, and thus change the ratio between the number of shares and the amount of ETH that is deposited in the protocol.
Exploit scenario: If a malicious user can call this function, he can increase the amount of reported ETH in the protocol by a large factor. This would in turn increase the *`stETH.balanceOf`* for all users by the same factor as well. Immediately afterward, the malicious user would be able to manipulate lending protocols that rely on off-chain price feeds and use stETH as a viable collateral, getting a large stETH-backed loan which is not truly covered by the supplied collateral.
Implications: Third-party protocols that use stETH can suffer major loss of funds..

Lido's response: *Acknowledged*. The function was introduced to support the onboarding of the already deposited validators with 0x00 credentials to the Lido protocol by rotating their withdrawal credentials to the type-0x01 ones used by Lido. The function is prefixed with `unsafe_` and restricted by a role with `UNSAFELY_` prefix to raise additional attention before ever being used. The Lido V2 deployment template doesn't assign this role. An explicit separate on-chain voting would have been required in a possible application case.

L-05: Anyone can clear penalty of a node operator, increase the nonce and potentially cause a revert of `deposit()` via frontrun

Severity: Low-
Category: Frontrun -> DoS
File(s): `NodeOperatorRegistry.sol`
Bug description: Similarly to issue H-01, it is possible to DoS new deposits via frontrun. If there is a node operator that completed its penalty but wasn't cleared yet, anyone could call `clearNodeOperatorPenalty()` to clear the penalty but it will also cause an increment to the validators keys nonce. Doing that as a frontrun to the Guardians' `deposit()` would cause a revert of the deposit. Unlike the mitigation mentioned in the response to H-01, here anyone could do the call to `clearNodeOperatorPenalty()`.
Exploit scenario: In a state where there are node operators that completed their penalty, and no one clears them until the deposit request, anyone who knows that the Guardians got a sign request for a deposit could call `clearNodeOperatorPenalty()` to DOS that deposit.
Implications: Although very unlikely, if possible to frontrun continuously, it could prevent new validator participation, which would decrease rewards (and thus reduce capital efficiency).
Lido's response: *Acknowledged***.** While the provided exploit scenario is possible, it's hard for a possible attacker to maintain the DoS opportunity open for a long time, as this could be easily mitigated by the monitoring tools and special bots resetting the penalties as soon as possible, mainly if malicious behavior was observed before. Moreover, Lido deposits are made using the private mempool by default which dramatically minimizes front-running risks.

Info-01: Possible to submit a report with zero hash

Severity: Informational
Category: Lack of checks
File(s): `BaseOracle.sol`
Property violated: `reportHashCannotBeZero`
Bug description: While the `HashConsensus.sol` contract considers the state of report's hash == bytes32(0) to be illegal, `BaseOracle.sol` will accept it
Exploit scenario: -
Implications: Discrepancy between contracts
Lido's response: ***Fixed*** in commit e618c30.

Assumptions and Simplifications Made During Verification

We made the following general assumptions during our verification:

- The following libraries: "`contracts/common/lib/MemUtils.sol`" and "`contracts/0.4.24/lib/SigningKeys.sol`" were omitted during formal verification due to tool compatibility issues. Nevertheless, those libraries were manually audited and are covered in the report.

Additional assumptions unique to the verification of specific contracts are listed explicitly under those contract sections.

Notations

✅Indicates the rule is formally verified.
❌Indicates the rule is violated.
⏳Indicates the rule is timing out.

Formal Verification Properties

Since the protocol consists of different contracts, we will present the relative properties for each of the main contracts in separate sections.
The rule's name will be shown as (`ruleName`)
Function names (and signatures) shall be written in Source code Pro font size 11 italic e.g.: *`foo(uint256)`*

Lido.sol

Assumptions

- Loop unrolling: we assume any loop can have at most 3 iterations.
- The ETH balance summary of every contract and user addresses never exceeds MAX_UINT128 (to avoid overflow).
- The shares balance summary of every contract and user addresses never exceeds MAX_UINT128 (to avoid overflow).

Properties

1. ✅There must be no fee on *`transferFrom`*. (`noFeeOnTransferFrom`)
2. ✅ There must be no fee on *`transferSharesFrom`*.

   (`noFeeOnTransferSharesFrom`)
3. ✅ There must be no fee on *`transfer`*. (`noFeeOnTransfer`)
4. ✅ There must be no fee on *`transferShares`*. (`noFeeOnTransferShares`)

5. ✅Token *transfer* must work correctly. Balances must be updated if not reverted. If reverted then the transfer amount must have been too high, or the recipient either 0, the same as the sender, or the *currentContract*. (`transferCorrect`)
6. ✅ *transferFrom* must work correctly. Balances must be updated if not reverted. (`transferFromCorrect`)
7. ✅ *transferSharesFrom* must work correctly. Balances must be updated if not reverted. (`transferSharesFromCorrect`)
8. ✅ *transferFrom* should revert if and only if the amount is too high or the recipient is 0 or the contract itself. (`transferFromReverts`)
9. ✅ *transferFrom* should revert if and only if the amount is too high or the recipient is 0 or the contract itself. (`transferSharesFromReverts`)
10. ✅ Allowance must change correctly as a result of calls to *approve*, *transferFrom*, *transferSharesFrom*, *increaseAllowance*, or *decreaseAllowance*. (`ChangingAllowance`)
11. ✅ A transfer from msg.sender to recipient must not change the balance of other addresses. (`TransferDoesntChangeOtherBalance`)
12. ✅ A transfer from sender to recipient using *transferFrom* must not change the balance of other addresses. (`TransferFromDoesntChangeOtherBalance`)
13. ✅ A transfer of shares from sender to recipient using *transferFrom* must not change the balance of other addresses. (`TransferSharesFromDoesntChangeOtherBalance`)
14. ✅ After calling *submit*:
    - If there is a stake limit then it must decrease by the submited eth amount.
    - The user gets the expected amount of shares.
    - Total shares count is increased as expected. (`integrityOfSubmit`)
15. ✅ After a successful call to *deposit*:
    - Bunker mode is inactive and the protocol is not stopped
    - If any of max deposits is greater than zero then the buffered ETH must decrease.
    - The buffered ETH must not increase. (`integrityOfDeposit`)
16. ✅ The buffered ETH must not exceed the ETH balance of the contract. (`BufferedEthIsAtMostLidoBalance`)

17. ✅ After a successful call to *`collectRewardsAndProcessWithdrawals`*:
    - Total EL rewards collected must increase by elRewardsToWithdraw
    - Contract's ETH balance must increase by elRewardsToWithdraw + withdrawalsToWithdraw - etherToLockOnWithdrawalQueue
    - The buffered ETH must increase by elRewardsToWithdraw + withdrawalsToWithdraw - etherToLockOnWithdrawalQueue (`integrityOfCollectRewardsAndProcessWithdrawals`)

Withdrawal Queue ERC721

Assumptions

- Loop unrolling: we assume any loop can have at most 3 iterations.

Properties

1. ✅ After calling *`requestWithdrawal`*:
    - The stETH.shares of the user must decrease.
    - The contract's stETH.shares must increase by the same amount.
    - The contract must generate the desired withdrawal request. (`integrityOfRequestWithdrawal`)
2. ✅ After calling *`requestWithdrawalsWstEth`*:
    - The WSTETH.balanceOf of the user must decrease by amount.
    - The contract's WSTETH.balanceOf must increase by the same amount.
    - The contract must generate the desired withdrawal request. (`integrityOfRequestWithdrawalsWstEth`)
3. ✅ After calling *`claimWithdrawal`*, if the user's ETH balance was increased then:
    - The locked ETH amount must be decreased.
    - The request's claimed and finalized flags must be on.
    - The request must be finalized. (`integrityOfClaimWithdrawal`)
4. ✅ After calling *`finalize`*, the locked ETH amount must have increased and the last finalized request-ID must be updated accordingly. (`integrityOfFinalize`)
5. ✅ If there is a new checkpoint index, then the last finalized request ID must have increased. (`priceIndexFinalizedRequestsCounterCorelation`)
6. ✅ Checkpoint history must be preserved. (`preserveCheckpointHistory`)

7. ✅ Claiming the same withdrawal request twice without asserting that there are no changes after the second claim must cause the code to revert. (`claimSameWithdrawalRequestTwice`)
8. ✅ A claimed withdrawal request must not later be unclaimed. (`onceClaimedAlwaysClaimed`)
9. ✅ The last finalized request-ID must always be less than the last request-ID. (`finalizedRequestsCounterisValid`)
10. ✅ It must not be possible to withdraw less than the minimum amount or more than the maximum amount. (`cantWithdrawLessThanMinWithdrawal`)
11. ✅ Each request's cumulative ETH must be greater than the minimum withdrawal amount. (`cumulativeEtherGreaterThamMinWithdrawal`)
12. ✅ Cumulative ETH and cumulative shares must increase monotonically. (`cumulativeEthMonotonocInc`)
13. ✅ If the request-ID is greater than the last finalized request-ID then the request's claimed and finalized flags must be off. (`finalizedCounterFinalizedFlagCorrelation`)
14. ✅ If a request is not finalized then it is not claimed. (`claimedFinalizedFlagsCorrelation`)
15. ✅ If there is a new checkpoint index then the Checkpoint's *`fromRequestId`* index must increase. (`newCheckpoint`)
16. ✅ Finalization FIFO order must be preserved. (`finalizationFifoOrder`)
17. ✅ Checkpoint's *`FromRequestId`* must always be less than the last finalized request-ID. (`CheckpointFromRequestIdisValid`)
18. ✅ Checkpoint's *`FromRequestId`* must increase monotonically. (`CheckpointFromRequestIdMonotonic`)

Staking Router

Definitions

- A registered module is a module whose ID is less than or equal to the module count
- An unregistered module is a module whose ID is greater than the module count.

Assumptions

- The key count summary for any staking module never exceeds MAX_UINT32 (< 2^{32} ~ 4.3 billion)
- Loop unrolling: we assume any loop can have at most 3 iterations

- *_makeBeaconChainDeposits32ETH()* was substituted with a simple optimistic ETH transfer to some external address.

Properties

1. ✅ The staking modules count must equal the last staking module ID. (`modulesCountIsLastIndex`)
2. ✅ Any module ID must be less than or equal to the last staking module ID. (`StakingModuleIdLELast`)
3. ✅
    a. Every registered staking module's ID must equal its one-based index+1.
    b. Every unregistered staking module's one-based index must be equal to zero. (`StakingModuleIndexIsIdMinus1`)
4. ✅ The staking module ID of an unregistered module must be equal to zero. (`StakingModuleId`)
5. ✅ The address of every registered staking module must never be zero. (`StakingModuleAddressIsNeverZero`)
6. ✅ The address of every unregistered staking module must always be zero. (`ZeroAddressForUnRegisteredModule`)
7. ✅ The target share of every staking module must always be less than or equal to 100% (i.e. <= TOTAL_BASIS_POINTS). (`stakingModuleTargetShareLEMAX`)
8. ✅ The total fee of every staking module (module fee + treasury fee) must always be less than or equal to 100% (i.e. <= TOTAL_BASIS_POINTS). (`stakingModuleTotalFeeLEMAX`)
9. ✅ The exited validators count of every unregistered module must be zero. (`ZeroExitedValidatorsForUnRegisteredModule`)
10. ❌ Staking modules addresses must be unique (there must be no two staking modules with the same address). (`StakingModuleAddressIsUnique`) (Issue M-06)
11. ✅ The staking module count must only be increased by one after calling *addStakingModule()*, and otherwise must not change. (`stakingModulesCountIncrement`)
12. ✅ Only the functions *setStakingModuleStatus()*, *addStakingModule()* and *resumeStakingModule()* must be able to turn a module to `active`. (`statusChangedToActive`)

13. ✅ Only the functions `setStakingModuleStatus` and `pauseStakingModule` must be able to turn a module to `paused`. (`statusChangedToPaused`)
14. ✅ Only the function `setStakingModuleStatus()` must be able to turn a module to `stopped`. (`statusChangedToStopped`)
15. ✅ Any method that can change a module status must do so for only one module at a time. (`oneStatusChangeAtATime`)
16. ✅ The exited validators count of every module must not decrease unless intended by calling `unsafeSetExitedValidatorsCount()`. (`ExitedValidatorsCountCannotDecrease`)
17. ❌ It must be impossible to add a new staking module with an address that is already registered. (`CannotAddStakingModuleIfAlreadyRegistered`) (Issue M-06)
18. ✅ The value returned by `getStakingModuleMaxDepositsCount()` must always be less than or equal to that staking module's depositable keys summary. (`validMaxDepositCountBound`)
19. ✅ It must be impossible to call `initialize()` twice. (`cannotInitializeTwice`)
20. ❌ For every staking module, the exited keys count must not surpass the staking module deposited keys count summary. (`moduleActiveValidatorsDoesntUnderflow`) (Issue M-03)
21. ✅ After setting a staking module to either 'paused' or 'stopped', the only functions that can revert, if they didn't do so before, must be `setStakingModuleStatus()`, `resumeStakingModule()` and `pauseStakingModule()`. (`whichFunctionsRevertIfStatusIsNotActive`)
22. ✅ After adding a staking module, it must always be possible to fetch that module later (`getStakingModule()` must never revert). (`canAlwaysGetAddedStakingModule`)
23. ❌ `getStakingModuleMaxDepositsCount()` must never revert for a registered staking module. (`getMaxDepositsCountRevert`) (Issue M-03)
24. ✅After calling `deposit(depositCount, moduleId, calldata)`, that same staking module's deposited keys summary must increase by `depositCount`. (`afterDepositSummaryIsUpdatedCorrectly`)
25. ✅Integrity of `getStakingFeeAggregationDistribution`:
    a. The sum of module fees must be less or equal to 100% (i.e. <= FEE_PRECISION_POINTS).

   b. The total fee must be less or equal to 100% (i.e. <= FEE_PRECISION_POINTS). (`aggregatedFeeLT100Percent`)

26. ❌ Adding another staking module must not result in *`getStakingFeeAggregateDistribution()`* reverting if it didn't before. (`feeDistributionDoesntRevertAfterAddingModule`) (Issue L-03)
27. ✅Excluding *`initialize()`*, only the functions *`grantRole()`*, *`renounceRole()`*, and *`revokeRole()`* must be able to change any role status (*`hasRole()`*) for any account. (`rolesChange`)

Node Operators Registry

Definitions

- A registered node operator is an operator whose ID is less than the node operator count.
- An unregistered operator is an operator whose ID is greater than or equal to the node operator count.

Assumptions

- The key count of any type, including summaries, for any node operator never exceeds MAX_UINT32 (< 2^{32} ~ 4.3 billion).
- Loop unrolling: we assume any loop can have at most 3 iterations.
- The precondition of the method *`finalizeUpgrade_v2()`* is that all key summaries (*`summarySigningKeysStats`*) are equal to zero.
- We assume no duplicates of node operator IDs in functions that receive batches of update data for several node operators (e.g. *`updateExitedValidatorsCount(bytes,bytes)`*).
- For the function *`obtainDepositData()`*, some of the rules below were verified assuming only one node operator in *`_loadAllocatedSigningKeys()`*.
- We haven't verified the validity of the internal functions of the *`SigningKeys`* library. This includes the *`getSigningKeys()`* function.

Properties

1. ✅ The node operator count must never surpass the max amount (MAX_NODE_OPERATORS_COUNT). (`NodeOperatorCountLEMAX`)
2. ✅ The active node operator count must always be less than or equal to the total node operator count. (`ActiveOperatorsLECount`)
3. ✅ The sum of all active node operators must equal the active node operator count. (`SumOfActiveOperatorsEqualsActiveCount`)

4. ✅ The sum of exited keys over all node operators must equal the exited keys summary. (`SumOfExitedKeysEqualsSummary`)
5. ✅ The sum of deposited keys over all node operators must equal the deposited keys summary. (`SumOfDepositedKeysEqualsSummary`)
6. ✅ The sum of max validators over all node operators must equal the max validators summary. (`SumOfMaxKeysEqualsSummary`)
7. ❌ The sum of total keys over all node operators must equal the total keys summary. (`SumOfTotalKeysEqualsSummary`) (Issue M-10)
8. ✅ For every node operator, the exited keys count must be less than or equal to the deposited keys count. (`ExitedKeysLEDepositedKeys`)
9. ✅ For every node operator, the deposited keys count must be less than or equal to the vetted keys count. (`DepositedKeysLEVettedKeys`)
10. ✅ For every node operator, the vetted keys count must be less than or equal to the total keys count. (`VettedKeysLETotalKeys`)
11. ✅ For every node operator, the max validators count must be less than or equal to the vetted keys count. (`VettedKeysGEMaxValidators`)
12. ✅ For every node operator, the sum of the stuck validators count and the exited keys count must be less than or equal to the deposited keys count. (`StuckPlusExitedLEDeposited`)
13. ❌ For every node operator, the sum of target validators and exited keys must not overflow (i.e. must be <= MAX_UINT64) (`TargetPlusExitedDoesntOverflow`) (Issue M-02)
14. ✅ The key count of every kind of every unregistered node operator must be zero. (`KeysOfUnregisteredNodeAreZero`)
15. ✅ An unregistered node operator must not be penalized. (`UnregisteredOperatorIsNotPenalized`)
16. ✅ An inactive node operator must have no depositable keys, i.e. its deposited keys count must equal its vetted keys count and its max validators count. (`NoDepositableKeysForInactiveModule`)
17. ✅ If a node operator was added, it must always be possible to deactivate it in the future (without reverting). (`canAlwaysDeactivateAddedNodeOperator`)
18. ✅ If a node operator was activated, it must always be possible to deactivate it in the future (without reverting). (`canDeactivateAfterActivate`)
19. ✅ If a node operator was deactivated, it must always be possible to activate it in the future (without reverting). (`canActivateAfterDeactivate`)

20. ✅ It must be impossible to call *finalizeUpgrade_v2()* twice. (`cannotFinalizeUpgradeTwice`)
21. ✅ It must be impossible to call *initialize()* twice. (`cannotInitializeTwice`)
22. ✅ Only *deactivateNodeOperator* must be able to deactivate a node operator. (`whoDeactivatesNodeOperators`)
23. ✅ The sum of reward shares over all node operators given by *getRewardsDistribution()* must always be less than or equal to the number of distributed reward shares. (`sumOfRewardsSharesLETotalShares`)
24. ✅ *getRewardsDistribution()* monotonicity: if the total distributed reward shares increases, the distributed rewards for any node operator must not decrease. (`rewardSharesAreMonotonicWithTotalShares`)
25. ✅ The number of exited keys must not decrease unless intended by calling *unsafeUpdateValidatorsCount()* for every node operator. (`exitedKeysDontDecrease`)
26. ✅ No function (except *updateExitedValidatorsCount*()) must be able to change the exited keys count of more than one node operator. (`exitedKeysChangeForOnlyOneNodeOperator`)
27. ✅ For every node operator the number of deposited keys must not decrease. (`depositedKeysDontDecrease`)
28. ✅ Only *obtainDepositData*() must be able to change a node operator deposited keys count. (`depositedKeysDontChangeByOtherFunctions`)
29. ✅ Total keys transition characteristics for every node operator:
    a. *addSigningKeys()* : Must be increased by the key count.
    b. *removeSigningKeys()* : Must be decreased by the key count.
    c. *invalidateReadyToDepositKeysRange()* : Must be equal to the deposited key count after the call.
    d. *onWithdrawalCredentialsChanged()* : Must be equal to the deposited key count after the call.
    e. Any other function must not be able to change the total keys count for any node operator. (`totalKeysChangeIntegrity`)
30. ✅ No function (except *updateExitedValidatorsCount*() and *updateStuckValidatorsCount*()) must be able to change the total keys count of more than one node operator. (`totalKeysChangeForOnlyOneNodeOperator`)

31. ✅ The function *obtainDepositData(depositsCount,data)* must not revert if the depositsCount is less than or equal to the depositable keys summary of the module. (obtainDepositDataDoesntRevert)
32. ✅ After deactivating a node operator, its depositable keys count must be set to zero (vetted = deposited = max validators). (afterDeactivateNoDepositableKeys)
33. ✅ Integrity of *_invalidateReadyToDepositKeysRange(from,to)* : the function must not change the keys count of any kind for a node operator whose ID is outside the range [from,to). (invalidateReadytIndexIntegrity)
34. ✅ After calling *clearNodeOperatorPenalty()* for some node operator:
    a. The function *isOperatorPenaltyCleared()* must return true for the same node operator.
    b. The function *isOperatorPenalized()* must return false for the same node operator.
    c. Any other node operator's penalty status must not be affected. (afterClearPenaltyOperatorIsNotPenalized)
35. ❌ Integrity of updating the stuck/refunded keys: by calling either *_updateStuckValidatorsCount()* or *_updateRefundedValidatorsCount()* for a specific node operator with some *validators_count* the following must hold:
    a. After updating the stuck keys, the stuck keys count of that node operator must be equal to *validators_count*.
    b. After updating the refunded keys, the refunded keys count of that node operator must be equal to *validators_count*.
    c. If an operator was **penalized** before and the stuck keys count is **larger** than the refunded keys count after the call, it must remain **penalized**.
    d. If an operator was **not penalized** before and the stuck keys count is **less than or equal** to the refunded keys count after the call, it must remain **not penalized**.
    e. If an operator was **not penalized** before and the stuck keys count is **larger** than the refunded keys count after the call, it must become **penalized**. (operatorPenaltyStatusAfterKeysUpdate) (Issues M-09, M-05)
36. ✅ Cannot clear penalty for unregistered node operators: The function *clearNodeOperatorPenalty(nodeOperatorId)* for an unregistered node operator must always revert. (cannotClearPenaltyForUnRegisteredOperators)

Deposit Security Module

Assumptions

- *_makeBeaconChainDeposits32ETH()* was substituted with a simple ETH transfer for simplification purposes.
- Maximum *guardians* array size was limited to (`max_uint128 - 1`)
- Loop unrolling: we assume any loop can have at most 3 iterations

Properties

1. ✅ Only *owner* must be able to change *owner* and only via *setOwner()*. (`onlyOwnerCanChangeOwner`)
2. ✅ Only *owner* must be able to change *pauseIntentValidityPeriodBlocks* and only via *setPauseIntentValidityPeriodBlocks()*. (`onlyOwnerCanChangePauseIntentValidityPeriodBlocks`)
3. ✅ Only *owner* must be able to change *maxDepositsPerBlock* and only via *setMaxDeposits()*. (`onlyOwnerCanChangeMaxDepositsPerBlock`)
4. ✅ Only *owner* must be able to change *minDepositBlockDistance* and only via *setMinDepositBlockDistance()*. (`onlyOwnerCanChangeMinDepositBlockDistance`)
5. ✅ Only *owner* must be able to change *quorum* and only via *setGuardianQuorum()*, *addGuardian()*, *addGuardians()*, and *removeGuardian()*. (`onlyOwnerCanChangeQuorum`)
6. ✅ `quorum` was set correctly to `newQuorum` value. (`correctQuorumUpdate`)
7. ✅ Only *owner* must be able to add/remove *guardians* and only via *addGuardian()*, *addGuardians()*, and *removeGuardian()*. (`onlyOwnerCanChangeGuardians`)
8. ✅ Only *owner* must be able to unpause deposits and only via *unpauseDeposits()*. (`onlyOwnerCanChangeUnpause`)
9. ✅ If *stakingModuleId* was paused, it must be possible to unpause it. (`canUnpause`)
10. ✅ It must be impossible to stop deposits from `DepositSecurityModule.sol`. (`cantStop`)

11. ✅ If *stakingModuleId* is paused, *canDeposit()* must return false. (cannotDepositDuringPauseBool)
12. ✅ If *stakingModuleId* is paused, *depositBufferedEther()* must revert. (cannotDepositDuringPauseRevert)
13. ✅ The *guardians* array must not contain the same *guardian* twice. The *guardians* array and *guardianIndicesOneBased* mapping must be correlated. (unique)
14. ✅ The zero address must not be a *guardian*. (zeroIsNotGuardian)
15. ✅ Checking signatures: It must not be possible to pass the same *guardian* signature twice in *_verifySignaturesCall()*. (youShallNotPassTwice)
16. ✅Checking signatures: It must not be possible to pass a signature of non-guardian. (nonGuardianCantSign)
17. ✅If *canDeposit()* returns false/reverts, *depositBufferedEther()* must revert. (agreedRevertsSimple)
18. ✅Only a *guardian* must be able to pause deposits. (onlyGuardianCanPause)

Base Oracle

Assumptions

- Loop unrolling: we assume any loop can have at most two iterations

Properties

1. ✅ For *setConsensusContract()* to be called, *msg.sender* must have the appropriate role. (onlyManagerCanSetConsensusContract)
2. ✅ For *setConsensusVersion()* to be called, *msg.sender* must have the appropriate role (onlyManagerCanSetConsensusVersion)
3. ✅ Only the predefined Consensus contract must be able to submit a report, i.e., call *submitConsensusReport()*.(onlyConsensusContractCanSubmitConsensusReport)
4. ✅ It must not be possible to *submitConsensusReport()* if its *refSlot* is less than *prevSubmittedRefSlot* (refSlotCannotDecrease)
5. ✅ It must not be possible to *submitConsensusReport()* if its *refSlot* is less than or equal to *prevProcessingRefSlot*. (refSlotMustBeGreaterThanProcessingOne)

6. ❌ It must not be possible to *submitConsensusReport()* if its *reportHash* is equal to *0*. (reportHashCannotBeZero) (Issue Info-01)

Validators ExitBus Oracle

Assumptions

- Loop unrolling: we assume any loop can have at most three iterations
- One check was commented out from the code because of the tool limitations in _handleConsensusReportData(): ValidatorsExitBusOracle.sol#L360

Properties

1. ✅ It must not be possible to initialize twice. (cantInitializeTwice)
2. ✅ Only a user with the necessary role must be able to resume within the pause time frame. (canResume)
3. ✅ It must not be possible to submit the same report twice. (cantSubmitReportTwice)
4. ✅ It must not be possible for *getTotalRequestsProcessed* to be decreased. (totalRequestsProcessedMonotonicity)
    a. only *submitReportData()* can change *getTotalRequestsProcessed*.
5. ✅ if the report data wasn't submitted and deadline (ConsensusReport.processingDeadlineTime) is passed, then it must be impossible for data to be submitted (cannotSubmitAnymore)
6. ✅ Integrity properties of *submitReportData()* must hold (submitReportDataIntegrity):
    a. can't exit more validators than a limit;
    b. if requestsCount was 0, TOTAL_REQUESTS_PROCESSED_POSITION remains unchanged;
    c. the caller is not a member of the oracle committee and doesn't possess the SUBMIT_DATA_ROLE;
    d. the provided contract version is different from the current one;
    e. the provided consensus version is different from the expected one;
    f. the provided reference slot differs from the current consensus frame's one;
    g. the processing deadline for the current consensus frame is missed.
7. ✅no function, except *submitReportData()*, must be able to change the value in LAST_PROCESSING_REF_SLOT_POSITION (whoCanChangeLastProcessingEtc)

8. ✅*getLastRequestedValidatorIndices()* must not return the same index for the same `moduleId` and `nodeOpIds` (`noSameIndex`)
9. ✅After successfully processing a consensus report, the `lastProcessingRefSlot` must be updated correctly (`correctUpdateOfLastProcessingRefSlot`)
10. ✅The index from *getLastRequestedValidatorIndices()* must not decrease (`indexIncrease`)

Accounting Oracle

Assumptions

- Loop unrolling: we assume any loop can have at most two iterations.

Properties

1. ✅ It must not be possible to initialize the contract twice (`cannotInitializeTwice`)
2. ✅ It must not be possible to initialize the contract without admin or consensus contract (`cannotInitializeWithEmptyAddresses`)
3. ✅ The following scenarios must revert when calling *submitReportData()*:
    a. Both the caller is not a member of the oracle committee and doesn't possess the *SUBMIT_DATA_ROLE*.
    b. The provided contract version is different from the current one.
    c. The provided consensus version is different from the expected one.
    d. The provided reference slot differs from the current consensus frame's slot.
    e. The processing deadline for the current consensus frame is missed.
    f. The keccak256 hash of the ABI-encoded data differs from the last hash provided by the hash consensus contract.
    (`correctRevertsOfSubmitReportData`)
4. ✅ *ReportData.extraDataFormat* must not be anything other than *EXTRA_DATA_FORMAT_EMPTY=0* or *EXTRA_DATA_FORMAT_LIST=1* (`correctRevertsOfSubmitReportData`) note: verified by the same rule as in property 3 above
5. ✅ If the oracle report contains no extra data, then *ReportData.extraDataHash* must equal *0* (`correctRevertsOfSubmitReportData`) note: verified by the same rule as in property 3 above

6. ✅ If the oracle report contains extra data, then *ReportData.extraDataHash* must not equal *0* (`correctRevertsOfSubmitReportData`) note: verified by the same rule as in property 3 above
7. ✅ If the oracle report contains no extra data, then *ReportData.extraDataItemsCount* must equal *0* (`correctRevertsOfSubmitReportData`) note: verified by the same rule as in property 3 above
8. ✅ If the oracle report contains extra data, then *ReportData.extraDataItemsCount* must not equal *0* (`correctRevertsOfSubmitReportData`) note: verified by the same rule as in property 3 above
9. ✅ For *submitReportData()*, *submitReportExtraDataList()*, or *submitReportExtraDataEmpty()* to be called, *msg.sender* must have the appropriate role *SUBMIT_DATA_ROLE* (same as 3a) or the caller must be a member of the oracle committee. (`callerMustHaveSubmitDataRoleOrBeAConsensusMember`)
10. ✅ It must not be possible to call *submitReportData()*, *submitReportExtraDataList()*, or *submitReportExtraDataEmpty()* twice at the same *block.timestamp* with different arguments (`cannotSubmitReportDataTwiceAtSameTimestamp`)
11. ✅ It must not be possible to call *submitReportData()*, *submitReportExtraDataList()*, or *submitReportExtraDataEmpty()* twice with the same arguments but at a different *block.timestamp*. (`cannotSubmitTheSameReportDataTwice`)
12. ✅ It must not be possible to call *submitReportExtraDataEmpty()* if the report submitted with *submitReportData()* had *report.extraDataFormat != EXTRA_DATA_FORMAT_EMPTY* (`cannotSubmitReportExtraDataEmptyWhenExtraDataIsNotEmpty`)
13. ✅ It must not be possible to call *submitReportExtraDataList()* if the report submitted with *submitReportData()* had *report.extraDataFormat != EXTRA_DATA_FORMAT_LIST* (`cannotSubmitReportExtraDataListWhenExtraDataIsEmpty`)

14. ✅ No function except *submitReportData()* must be able to change the value stored in *LAST_PROCESSING_REF_SLOT_POSITION* (`nobodyCanChangeLastProcessingRefSlotExceptSubmitReportData`)
15. ✅ The processed *refSlot* value stored in *LAST_PROCESSING_REF_SLOT_POSITION* must only increase (`refSlotIsMonotonicallyIncreasing`)
16. ✅ After successfully processing a consensus report, the *lastProcessingRefSlot* must be updated correctly (`correctUpdateOfLastProcessingRefSlot`)
17. ✅ To be submitted, a report must be newer and point to a higher *refSlot* than previous submissions. (`correctUpdateOfLastProcessingRefSlot`) note: the same rule as in 16 above
18. ❌ It must not be possible to submit a new report without first calling *submitReportExtraDataList()* or *submitReportExtraDataEmpty()* (`cannotSubmitNewReportIfOldWasNotProcessedFirst`)
    The failure suggests that it is possible to submit a new report without first providing the expected extra data of the previous report. (Issue M-01)

Hash Consensus

Assumptions

- Loop unrolling: we assume any loop can have at most two iterations.
- For the sake of simplification, to prevent timeouts, in some of the rules we assume explicit and sane values for the parameters representing the chain configuration (`slotsPerEpoch`, `secondsPerSlot`, `genesisTime`), `epochsPerFrame` and `initialEpoch`
- We summarize the external functions of the `IReportAsyncProcessor` to return constant value, i.e., we assumed that the `consensusVersion` and the `lastProcessingRefSlot` are constant

Properties

1. ✅ Calling the external view function *getCurrentFrame()* must not revert (`getCurrentFrameDoesNotRevert`)
2. ✅ It must not be possible to call the external non-view functions without having the appropriate ACL role (`onlyAllowedRoleCanCallMethod`)
3. ✅ Setting the fast lane length to zero must disable the fast lane subset (`setFastLaneLengthSlotsCorrectness`)

4. ✅ It must not be possible to add an existing member (`addMemberRevertsCorrectly`)
5. ✅ It must not be possible to add and an empty address as a member (`addMemberRevertsCorrectly`) note: verified by the same rule as in property 4 above
6. ✅ Adding a new member must not add or remove any other member (`addMemberADoesNotModifyMemberB`)
7. ✅ Adding a new member must increase the total members only by 1 (`addMemberCorrectness`)
8. ✅ It must not be possible to remove a non-existing member (`removeMemberRevertsCorrectly`)
9. ✅ Removing a member must not add or remove any other member (`removeMemberADoesNotModifyMemberB`)
10. ✅ Removing a member must decrease the total members only by 1 (`removeMemberCorrectness`)
11. ✅ Calling *`setQuorum()`* must update the quorum correctly (`setQuorumCorrectness`)
12. ✅ Calling *`disableConsensus()`* must update the quorum to be unreachable (`disableConsensusCorrectness`)
13. ✅ It must not be possible to set the `ReportProcessor`'s address to be the same as its current address (`setReportProcessorCorrectness`)
14. ✅ It must not be possible to set the zero address as a report processor address (`setReportProcessorCorrectness`) note: verified by the same rule as in property 13 above
15. ✅ Calling *`setReportProcessor()`* must update the report processor correctly (`setReportProcessorCorrectness`) note: verified by the same rule as in property 13 above
16. ✅ It must not be possible to submit a report if the `slot` is above `maxUint64` (`cannotSubmitReportWhenSlotIsAboveMaxUint64`)
17. ✅ It must not be possible to submit a report if the `slot` does not match the current `refSlot` of the current frame (`cannotSubmitReportWhenSlotDoesNotMatchCurrentRefSlot`)
18. ✅ It must not be possible for the same member to submit two different reports that have the same `slot` (`sameMemberCannotSubmitDifferentReportForTheSameSlot`)

19. ✅ It must not be possible to submit a report if its hash is zero (`cannotSubmitReportWithEmptyHash`)
20. ✅ It must not be possible to submit a report if its `consensusVersion` does not match the return value of *`getConsensusVersion()`* (`submitReportMustHaveCorrectConsensusVersion`)
21. ✅ When submitting a report, if a new frame started, `_reportVariantsLength` must reset to 1 (`variantsResetUponNewFrameStart`)
22. ✅ It must not be possible to update the `initialEpoch` to be one that already arrived (`updateInitialEpochCorrectness`)
23. ✅ Calling *`updateInitialEpoch()`* must update the `initialEpoch` variable correctly (`updateInitialEpochCorrectness`) note: verified by the same rule as in property 22 above
24. ✅ If the system is initialized, the variable `initialEpoch` must always be sane (`initialEpochSanity`)
25. ✅ If the system is initialized, and the stored `initialEpoch` already passed, calling *`updateInitialEpoch()`* must always revert (`updateInitialEpochRevertsCorrectly`)

Legacy Oracle

Assumptions

- The product of the beacon chain 'slots per epoch' and 'seconds per slot' doesn't exceed MAX_UINT64 : `slotsPerEpoch * secondsPerSlot` $< 2^{64}$

Properties

1. ✅ It must not be possible to call *`finalizeUpgrade_v4()`* twice.

Min First Allocation Strategy

Assumptions

- The sum of buckets plus the allocation size must not overflow (i.e. must be <= MAX_UINT256).

Properties

1. ✅ *`allocate`* must never revert. (`allocateDoesntRevert`)

2. ✅ The sum of increments (differences in buckets after and before the allocation) must always be less than or equal to the allocation size and must be equal to the allocated amount. (`sumOfIncrementsEqualsAllocated`)
3. ✅ The capacity of any bucket must never be surpassed after the allocation. (`capacityIsNeverSurpassed`)
4. ✅ The least filled bucket must always be allocated with an amount if possible i.e. when the capacity > bucket and the allocation size > 0. (`minimumBucketIsAlwaysIncrementedWhenPossible`)

Access Control Enumerable

Properties

1. ✅ Only the admin of a role must be able to grant the role to any account; Granting roles permission must be possible for every role. (`onlyAdminCanGrantRole`)
2. ✅ Only the admin of a role or the role owner account must be able to revoke the role of an account; Revoking roles must be possible for every role. (`onlyAdminOrSelfCanRevokeRole`)
3. ✅ Granting or revoking a role to / from any account must not affect any role of any **other** account (`nonInterferenceOfRolesAndAccounts`)
4. ✅ Granting a role to a member must increase the count of *`getRoleMemberCount(role)`* by one. (`countIncreaseByOneWhenGrantRole`)
5. ✅ Revoking a role from a member must decrease the count of *`getRoleMemberCount(role)`* by one. (`countDecreaseByOneWhenRenounceRole`)
6. ✅ *`getRoleMemberCount(role)`* must not be affected by adding or removing roleR (roleR != roleX); If a member is added to /removed from the list of some role, the members count of any other role must not be changed. (`memberCountNonInterference`)
7. ✅ If a role was granted by an agent, it must always be possible for the same agent to revoke it afterwards. (`canRevokeAfterGrant`)
8. ✅ If a role was revoked, then the only case in which granting this role again could revert is when the role being revoked is the admin role. (`canGrantAfterRevoke`)
9. ✅ It must always be possible for an agent to renounce a role. (`canAlwaysRenounce`)

10. ✅ Any revoking or granting permission must change only one role status at a time and for only one address at a time. (`oneRoleAtATime`)
11. ✅ Granting a role to any account must add its address to the member list. (`roleSetAfterGrant`)
12. ✅ Integrity of *`hasRole(role, account)`:*
    a. After calling *`revokeRole(role, account)`*, the function must return false and the *`msg.sender`* must be the role admin.
    b. After calling *`grantRole(role, account)`*, the function must return true and the *`msg.sender`* must be the role admin.
    c. After calling *`renounceRole(role, account)`*, the function must return false, and the *`account`* must be the *`msg.sender`*. (`whoChangedRoles`)
13. ✅ *`hasRole(role, account)`* must be equivalent to *`contains(role, account)`* of the role's enumerable address set. (`containsRole`)
14. ✅ The role member list must have no duplicates. (`noDuplicates`)

Disclaimer

The Certora Prover takes a contract and a specification as input and formally proves that the contract satisfies the specification in all scenarios. Notably, the guarantees of the Certora Prover are scoped to the provided specification and the Certora Prover does not check any cases not covered by the specification.

Even though we hope this information is helpful, we provide no warranty of any kind, explicit or implied. The contents of this report should not be construed as a complete guarantee that the contract is secure in all dimensions. In no event shall Certora or any of its employees be liable for any claim, damages, or other liability, whether in an action of contract, tort, or otherwise, arising from, out of, or in connection with the results reported here.