Protocol levers

The protocol provides a number of settings controllable by the DAO. Modifying each of them requires the caller to have a specific permission. After deploying the DAO, all permissions belong to either DAOVoting orAgent apps, which can also manage them. This means that, initially, levers can only be changed by the DAO voting, and other entities can be allowed to do the same only as a result of the voting.

All existing levers are listed below, grouped by the contract.

A note on upgradeability

The following contracts are upgradeable by the DAO voting:

- LidoLocator
- Lido
- StakingRouter
- NodeOperatorsRegistry
- AccountingOracle
- ValidatorsExitBusOracle
- WithdrawalVault
- WithdrawalQueueERC721
- LegacyOracle

Upgradeability is implemented either by the Aragon kernel and base contracts OR by theOssifiableProxy instances. To upgrade an Aragon app, one needs thedao.APP_MANAGER_ROLE permission provided by Aragon. To upgrade anOssifiableProxy implementation, one needs to be an owner of the proxy. As it was said previously, both belong either to the DAOVoting orAgent apps.

All upgradeable contracts use theUnstructured Storage pattern in order to provide stable storage structure across upgrades.

note Some of the contracts still contain structured storage data, hence the order of inheritance always matters.

Lido

Burning stETH tokens

There is a dedicated contract responsible forstETH tokens burning. The burning itself is a part of the core protocol procedures:

- deduct underlying finalized withdrawal requeststETH
- , seeLido.handleOracleReport
- penalize delinquent node operators by halving their rewards, seeValidator exits and penalties

These responsibilities are controlled by theREQUEST_BURN_SHARES_ROLE role which is assigned to bothLido andNodeOperatorsRegistry contracts. This role should not be ever permanently assigned to another entities.

Apart from this,stETH token burning can be applied to compensate for penalties/slashing losses by the DAO decision. It's possible via more restrictive roleREQUEST_BURN_MY_STETH_ROLE which is currently unassigned.

The key difference that despite of both roles rely on thestETH allowance provided to theBurner contract, the latter allows token burning only from the request originator balance.

Pausing

- Mutator:stop()
  - Permission required:PAUSE_ROLE
- Mutator:resume()
  - Permission required:RESUME_ROLE
- Accessor:isStopped() returns (bool)

When paused,Lido doesn't accept user submissions, doesn't allow user withdrawals and oracle report submissions. No token actions (burning, transferring, approving transfers and changing allowances) are allowed. The following transactions revert:

- plain ether transfers toLido
- ;
- calls tosubmit(address)
- ;
- calls todeposit(uint256, uint256, bytes)
- ;
- calls tohandleOracleReport(...)
- ;
- calls totransfer(address, uint256)
- ;
- calls totransferFrom(address, address, uint256)
- ;
- calls totransferShares(address, uint256)
- ;
- calls totransferSharesFrom(address, uint256)
- ;
- calls toapprove(address, uint256)
- ;
- calls toincreaseAllowance(address, uint256)
- ;
- calls todecreaseAllowance(address, uint256)
- .

As a consequence of the list above:

- calls toWithdrawalQueueERC721.requestWithdrawals(uint256[] calldata, address)
- , and its variants;
- calls towstETH.wrap(uint256)
- andwstETH.unwrap(uint256)
- ;
- calls toBurner.requestBurnShares
- ,Burner.requestBurnMyStETH
- , and its variants;

note External stETH/wstETH DeFi integrations are directly affected as well.

Override deposited validators counter

- Mutator:unsafeChangeDepositedValidators(uint256)
-
    - Permission required:UNSAFE_CHANGE_DEPOSITED_VALIDATORS_ROLE

The method unsafely changes deposited validator counter. Can be required when onboarding external validators to Lido (i.e., had deposited before and rotated their type-0x00 withdrawal credentials to Lido).

The incorrect values might disrupt protocol operation.

Oracle report

TODO: oracle reports are committee-driven

Deposit access control

TheLido.deposit method performs an actual deposit (stake) of buffered ether to Consensus Layer undergoing throughStakingRouter , its selected module, and the official Ethereum deposit contract in the end.

The method can be called only byDepositSecurityModule since access control is a part of the deposits frontrunning vulnerability mitigation.

Please seeLIP-5 for more details.

Deposit loop iteration limit

Controls how many Ethereum deposits can be made in a single transaction.

- The_maxDepositsCount
- parameter of thedeposit(uint256 _maxDepositsCount, uint256 _stakingModuleId, bytes _depositCalldata)
- function
- Default value:16

- Scenario test

When DSM callsdepositBufferedEther ,Lido tries to register as many Ethereum validators as it can given the buffered ether amount. The limit is passed as an argument to this function and is needed to prevent the transaction fromfailing due to the block gas limit , which is possible if the amount of the buffered ether becomes sufficiently large.

Execution layer rewards

Lido implements an architecture design which was proposed in the Lido Improvement Proposa#12 to collect the execution level rewards (starting from the Merge hardfork) and distribute them as part of the Lido Oracle report.

These execution layer rewards are initially accumulated on the dedicatedLidoExecutionLayerRewardsVault contract and consists of priority fees and MEV.

There is an additional limit to prevent drastic token rebase events. See the following issue#405

- Mutator:setELRewardsVault()
-
  - Permission required:SET_EL_REWARDS_VAULT_ROLE
- Mutator:setELRewardsWithdrawalLimit()
-
  - Permission required:SET_EL_REWARDS_WITHDRAWAL_LIMIT_ROLE
- Accessors:
-
  - getELRewardsVault()
-
  - ;
-
  - getELRewardsWithdrawalLimit()
-
  - .

Staking rate limiting

Lido features a safeguard mechanism to prevent huge APR losses facing thepost-merge entry queue demand .

New staking requests could be rate-limited with a soft moving cap for the stake amount per desired period.

Limit explanation scheme:

- ▲ Stake limit
- | .... Stake limit = max
- | . . . . . .
- | . . . . . .
- | . . . . .
- | ——>
  Time
- | ^ ^ ^ ^^^ ^ ^ ^ ^^^ ^ Stake events
- Mutators:resumeStaking()
- ,setStakingLimit(uint256, uint256)
- ,removeStakingLimit()
-
  - Permission required:STAKING_CONTROL_ROLE
- Mutator:pauseStaking()
-
  - Permission required:STAKING_PAUSE_ROLE
- Accessors:
-
  - isStakingPaused()
-
  - getCurrentStakeLimit()
-
  - getStakeLimitFullInfo()

When staking is paused,Lido doesn't accept user submissions. The following transactions revert:

- Plain ether transfers;
- calls tosubmit(address)
- .

For details, see the Lido Improvement Proposal#14 .

StakingRouter

Fee

The total fee, in basis points (10000 corresponding to100%).

- Mutator:setFee(uint16)
-
    - Permission required:MANAGE_FEE
- Accessor:getFee() returns (uint16)

The fee is taken on staking rewards and distributed between the treasury, the insurance fund, and node operators.

Fee distribution

Controls how the fee is distributed between the treasury, the insurance fund, and node operators. Each fee component is in basis points; the sum of all components must add up to 1 (10000 basis points).

- Mutator:setFeeDistribution(uint16 treasury, uint16 insurance, uint16 operators)
-
    - Permission required:MANAGE_FEE
- Accessor:getFeeDistribution() returns (uint16 treasury, uint16 insurance, uint16 operators)

Ethereum withdrawal Credentials

Credentials to withdraw ETH on the Execution Layer side

- Mutator:setWithdrawalCredentials(bytes)
-
    - Permission required:MANAGE_WITHDRAWAL_KEY
- Accessor:getWithdrawalCredentials() returns (bytes)

The protocol uses these credentials to register new Ethereum validators.

NodeOperatorsRegistry

Node Operators list

- Mutator:addNodeOperator(string _name, address _rewardAddress, uint64 _stakingLimit)
-
    - Permission required:ADD_NODE_OPERATOR_ROLE
- Mutator:setNodeOperatorName(uint256 _id, string _name)
-
    - Permission required:SET_NODE_OPERATOR_NAME_ROLE
- Mutator:setNodeOperatorRewardAddress(uint256 _id, address _rewardAddress)
-
    - Permission required:SET_NODE_OPERATOR_ADDRESS_ROLE
- Mutator:setNodeOperatorStakingLimit(uint256 _id, uint64 _stakingLimit)
-
    - Permission required:SET_NODE_OPERATOR_LIMIT_ROLE

Node Operators act as validators on the Beacon chain for the benefit of the protocol. Each node operator submits no more than_stakingLimit signing keys that will be used later by the protocol for registering the corresponding Ethereum validators. As oracle committee reports rewards on the Ethereum side, the fee is taken on these rewards, and part of that fee is sent to node operators' reward addresses (_rewardAddress).

Deactivating a node operator

- Mutator:setNodeOperatorActive(uint256 _id, bool _active)
-
    - Permission required:SET_NODE_OPERATOR_ACTIVE_ROLE

Misbehaving node operators can be deactivated by calling this function. The protocol skips deactivated operators during validator registration; also, deactivated operators don’t take part in fee distribution.

Managing node operator’s signing keys

- Mutator:addSigningKeys(uint256 _operator_id, uint256 _quantity, bytes _pubkeys, bytes _signatures)
-
  - Permission required:MANAGE_SIGNING_KEYS
- Mutator:removeSigningKey(uint256 _operator_id, uint256 _index)
-
  - Permission required:MANAGE_SIGNING_KEYS

Allow to manage signing keys for the given node operator.

Signing keys can also be managed by the reward address of a signing provider by calling the equivalent functions with theOperatorBH suffix:addSigningKeysOperatorBH ,removeSigningKeyOperatorBH .

Reporting new stopped validators

- Mutator:reportStoppedValidators(uint256 _id, uint64 _stoppedIncrement)
-
  - Permission required:REPORT_STOPPED_VALIDATORS_ROLE

Allows to report that_stoppedIncrement more validators of a node operator have become stopped.

LegacyOracle

Lido

Address of the Lido contract.

- Accessor:getLido() returns (address)

Members list

The list of oracle committee members.

- Mutators:addOracleMember(address)
- ,removeOracleMember(address)
-
  - Permission required:MANAGE_MEMBERS
- Accessor:getOracleMembers() returns (address[])

The quorum

The number of exactly the same reports needed to finalize the epoch.

- Mutator:setQuorum(uint256)
-
  - Permission required:MANAGE_QUORUM
- Accessor:getQuorum() returns (uint256)

When thequorum number of the same reports is collected for the current epoch,

- the epoch is finalized (no more reports are accepted for it),
- the final report is pushed to the Lido,
- statistics collected and thesanity check
- is evaluated,

Sanity check

To make oracles less dangerous, we can limit rewards report by 0.1% increase in stake and 15% decrease in stake, with both values configurable by the governance in case of extremely unusual circumstances.

- Mutators:setAllowedBeaconBalanceAnnualRelativeIncrease(uint256)
- andsetAllowedBeaconBalanceRelativeDecrease(uint256)
-

    - Permission required:SET_REPORT_BOUNDARIES
- Accessors:getAllowedBeaconBalanceAnnualRelativeIncrease() returns (uint256)
- andgetAllowedBeaconBalanceRelativeDecrease() returns (uint256)

Current reporting status

For transparency we provide accessors to return status of the oracle daemons reporting for the current "expected epoch ".

- Accessors:* getCurrentOraclesReportStatus() returns (uint256)
-
    -
        - returns the current reporting bitmap,
-
    - representing oracles who have already pushed their version of report during the expected
-
    - epoch, every oracle bit corresponds to the index of the oracle in the current members list,
-
    - getCurrentReportVariantsSize() returns (uint256)
-
    -
        - returns the current reporting variants
-
    - array size,
-
    - getCurrentReportVariant(uint256 _index) returns (uint64 beaconBalance, uint32 beaconValidators, uint16 count)
-
    -
        - returns the current reporting array element with the given
-
    - index.

Expected epoch

The oracle daemons may provide their reports only for the one epoch in every frame: the first one. The following accessor can be used to look up the current epoch that this contract expects reports.

- Accessor:getExpectedEpochId() returns (uint256)
- .

Note that any later epoch, that has already comeand is also the first epoch of its frame, is also eligible for reporting. If some oracle daemon reports it, the contract discards any results of this epoch and advances to the just reported one.

Version of the contract

Returns the initialized version of this contract starting from 0.

- Accessor:getVersion() returns (uint256)
- .

Beacon specification

Sets and queries configurable beacon chain specification.

- Mutator:setBeaconSpec(uint64 _epochsPerFrame, uint64 _slotsPerEpoch, uint64 _secondsPerSlot, uint64 _genesisTime)
- ,* Permission required:SET_BEACON_SPEC
-
    - ,
- Accessor:getBeaconSpec() returns (uint64 epochsPerFrame, uint64 slotsPerEpoch, uint64 secondsPerSlot, uint64 genesisTime)
- .

Current epoch

Returns the epoch calculated from current timestamp.

- Accessor:getCurrentEpochId() returns (uint256)
- .

Supplemental epoch information

Returns currently reportable epoch (the first epoch of the current frame) as well as its start and end times in seconds.

- Accessor:getCurrentFrame() returns (uint256 frameEpochId, uint256 frameStartTime, uint256 frameEndTime)
- .

Last completed epoch

Return the last epoch that has been pushed to Lido.

- Accessor:getLastCompletedEpochId() returns (uint256)
- .

Supplemental rewards information

Reports beacon balance and its change during the last frame.

- Accessor:getLastCompletedReportDelta() returns (uint256 postTotalPooledEther, uint256 preTotalPooledEther, uint256 timeElapsed)
- . Edit this page Previous Multisig deployment Next DAO voting with Etherscan