SMART CONTRACT AUDIT REPORT FOR LIDO V2

CONTENTS

CONTENTS

CONTENTS

ABOUT HEXENS

Hexens is a cybersecurity company that strives to elevate the standards of security in Web 3.0, create a safer environment for users, and ensure mass Web 3.0 adoption.

Hexens has multiple top-notch auditing teams specialized in different fields of information security, showing extreme performance in the most challenging and technically complex tasks, including but not limited to: Infrastructure Audits, Zero Knowledge Proofs / Novel Cryptography, DeFi and NFTs. Hexens not only uses widely known methodologies and flows, but focuses on discovering and introducing new ones on a day-to-day basis.

In 2022, our team announced the closure of a $4.2 million seed round led by IOSG Ventures, the leading web3 venture capital. Other investors include Delta Blockchain Fund, Chapter One, Hash Capital, ImToken Ventures, Tenzor Capital, and angels from Polygon and other blockchain projects.

Since Hexens was founded in 2021, it has had an impressive track record and recognition in the industry: Mudit Gupta - CISO of Polygon Technology - the biggest EVM Ecosystem, joined the company advisory board after completing just a single cooperation iteration. Polygon Technology, 1inch, Coinstats, Hats Finance, Quickswap, Layerswap, 4K, RociFi, as well as dozens of DeFi protocols and bridges, have already become our customers and taken proactive measures towards protecting their assets.

AUDIT LED BY

KASPER ZWIJSEN

Lead Smart Contract Auditor | Hexens

Audit Starting Date
06.02.2023

Audit Completion Date
14.04.2023

METHODOLOGY

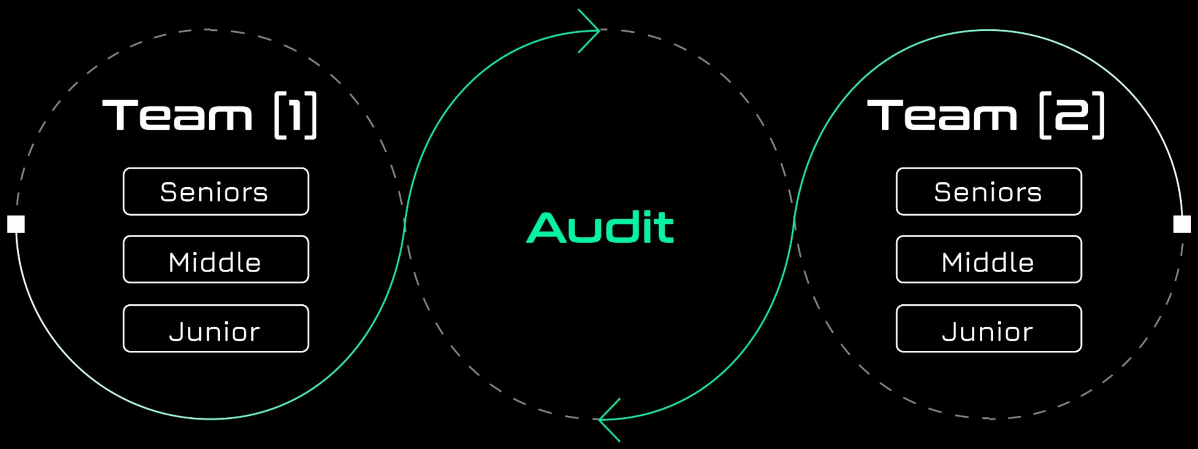
COMMON AUDIT PROCESS

Companies often assign just one engineer to one security assessment with no specified level. Despite the possible impeccable skills of the assigned engineer, it carries risks of the human factor that can affect the product's lifecycle.

Auditor*
Audit

HEXENS METHODOLOGY

Hexens methodology involves 2 teams, including multiple auditors of different seniority, with at least 5 security engineers. This unique cross-checking mechanism helps us provide the best quality in the market.



SEVERITY STRUCTURE

The vulnerability severity is calculated based on two components

- Impact of the vulnerability
- Probability of the vulnerability

IMPACT	PROBABILITY			
	Rare	Unlikely	Likely	Very Likely
Low / Info	Low / Info	Low / Info	Medium	Medium
Medium	Low / Info	Medium	Medium	High
High	Medium	Medium	High	Critical
Critical	Medium	High	Critical	Critical

SEVERITY CHARACTERISTICS

Smart contract vulnerabilities can range in severity and impact, and it's important to understand their level of severity in order to prioritize their resolution. Here are the different types of severity levels of smart contract vulnerabilities:

CRITICAL

Vulnerabilities with this level of severity can result in significant financial losses or reputational damage. They often allow an attacker to gain complete control of a contract, directly steal or freeze funds from the contract or users, or permanently block the functionality of a protocol. Examples include infinite mints and governance manipulation.

HIGH

Vulnerabilities with this level of severity can result in some financial losses or reputational damage. They often allow an attacker to directly steal yield from the contract or users, or temporarily freeze funds. Examples include inadequate access control integer overflow/underflow, or logic bugs.

MEDIUM

Vulnerabilities with this level of severity can result in some damage to the protocol or users, without profit for the attacker. They often allow an attacker to exploit a contract to cause harm, but the impact may be limited, such as temporarily blocking the functionality of the protocol. Examples include uninitialized storage pointers and failure to check external calls.

LOW

Vulnerabilities with this level of severity may not result in financial losses or significant harm. They may, however, impact the usability or reliability of a contract. Examples include slippage and front-running, or minor logic bugs.

INFORMATIONAL

Vulnerabilities with this level of severity are regarding gas optimizations and code style. They often involve issues with documentation, incorrect usage of EIP standards, best practices for saving gas, or the overall design of a contract. Examples include not conforming to ERC20, or disagreement between documentation and code.

It's important to consider all types of vulnerabilities, including informational ones, when assessing the security of smart contracts. A comprehensive security audit should consider all types of vulnerabilities to ensure the highest level of security and reliability.

EXECUTIVE SUMMARY

OVERVIEW

Lido on Ethereum is a liquid staking solution for Ethereum.

This audit covers the Lido V2 upgrade, which will enable withdrawals of staked Ether, as well as allow anyone to propose to the Lido DAO and build staking modules with different participation rules and requirements (e.g., permissionless or permissioned), storage flavor (e.g., off-chain key storage) and fee distribution.

Our security assessment was a full review of the Lido V2 protocol, which includes some legacy code, the developed smart contracts and the new off-chain oracle. We have thoroughly reviewed each contract individually, as well as the system as a whole.

During our audit, we identified 1 critical severity vulnerability. It was found in the Withdrawal Queue’s hint mechanism, where a user would be able to withdraw 100% of their Ether even if a discount was applied.

We have also identified 3 high severity vulnerabilities. In the first one, we showed that a malicious actor could always block Lido deposits; in the second one, we discovered flaws in the share-burning mechanism regarding withdrawal requests. The third one is the inter-contract compatibility that would result in the protocol being unable to finalize withdrawal requests.

Furthermore, our report also covers various minor vulnerabilities, optimizations, and design recommendations.

Finally, all our reported issues were fixed or acknowledged by the Lido development team and consequently validated by us.

We can confidently say that the overall security and code quality has increased significantly after the completion of our audit.

SCOPE

The analyzed resources are located on:
https://github.com/lidofinance/lido-dao/commit/e57517730c3e11a41e9cbc32ce018726722335b7
https://github.com/lidofinance/lido-oracle/commit/20bc575f4fae7b3139b210b92f6d05a28215a9fb

https://github.com/lidofinance/lido-dao/commit/2bce10d4f0cb10cde11bead4719a5bcde76b93f9
https://github.com/lidofinance/lido-oracle/commit/f3b314c31d9823f8c68b8ab3458f4c24a0eef004

https://github.com/lidofinance/lido-dao/commit/e45c4d6fb8120fd29426b8d969c19d8a798ca974

The issues described in this report were fixed. Corresponding commits are mentioned in the description.

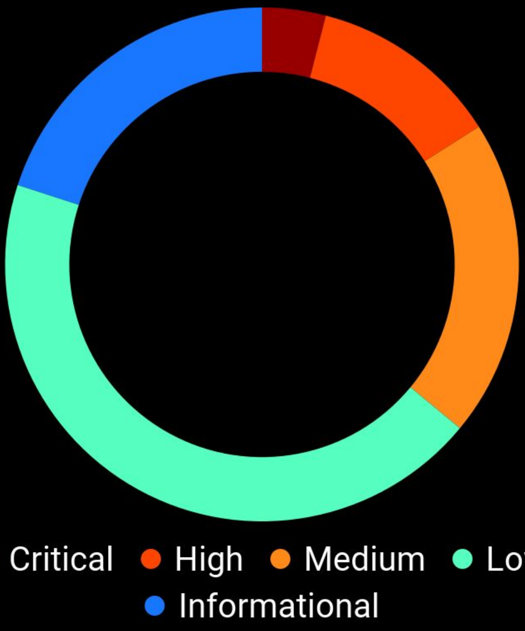
The final validated version of the protocol:
https://github.com/lidofinance/lido-dao/commit/e45c4d6fb8120fd29426b8d969c19d8a798ca974

SUMMARY

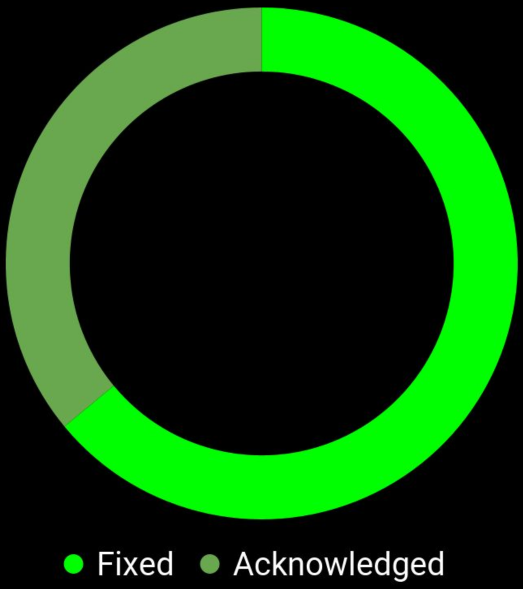
SEVERITY	NUMBER OF FINDINGS
CRITICAL	1
HIGH	3
MEDIUM	5
LOW	11
INFORMATIONAL	5

TOTAL: 25

SEVERITY



STATUS



WEAKNESSES

This section contains the list of discovered weaknesses.

LID-1. STEALING ETH USING DISCOUNT FACTOR BY PASS

SEVERITY: Critical

PATH: WithdrawalQueueBase.sol:_claimWithdrawalTo:L428-462

REMEDIATION: replace _hint with _requestId on line 449 in WithdrawalQueueBase.sol

STATUS: fixed, commits: 1, 2

DESCRIPTION:

Whenever a batch of withdrawal requests is finalised, a discount factor is calculated and a checkpoint is created if the new factor differs. The discount factor is calculated as the actual amount of ETH divided by the total requested amount and only the actual amount of ETH is locked.

When the user claims their withdrawal request, they would only receive their portion of the actual amount of ETH by multiplying their request amount with the discount factor.

We found that there is an incorrect comparison in **_claimWithdrawalTo** on line 449: The next checkpoint's **fromId** is compared with **_hint** instead of **_requestId**. As a result, this check can be entirely bypassed and allows a user to choose any checkpoint from the past with a better (or no) discount factor.

As a result, any user would be able to always claim the full amount, even though that ETH is not actually there in the contract. Too much ETH would be subtracted from the locked ETH counter and other withdrawal requests will not be claimable anymore.

The vulnerability can therefore be used to steal ETH from other users in the withdrawal queue.

```
if (_hint + 1 <= lastCheckpointIndex) {
   if (_getCheckpoints()[_hint + 1].fromId <= _hint) {
       revert InvalidHint(_hint);
   }
}
```

LID-17. WITHDRAWAL REQUEST FINALISATION WILL ALWAYS REVERT

SEVERITY: High

PATH:
OracleReportSanityChecker.sol:_checkRequestIdToFinalizeUpTo: L571-581

REMEDIATION: see description

STATUS: fixed

DESCRIPTION:

A withdrawal request gets finalised through an oracle report from the AccountingOracle to the Lido contract. Lido in turn checks the report's data using the OracleReportSanityChecker.

With regard to the withdrawal requests data, Lido calls **checkWithdrawalQueueOracleReport** on the OracleReportSanityChecker, which in turn calls **getWithdrawalRequestStatus** with the request ID on the WithdrawalQueue contract.

However, the function **getWithdrawalRequestStatus** does not exist in the WithdrawalQueue contract.

As a result, withdrawal request finalisation will always revert.

We suspect that this vulnerability never surfaced due to the use mock contracts. For example, the tests for the OracleReportSanityChecker use a WithdrawalQueueStub contract, which has the missing function but returns mock values.

```
function _checkRequestIdToFinalizeUpTo(
  LimitsList memory _limitsList,
  address _withdrawalQueue,
  uint256 _requestIdToFinalizeUpTo,
  uint256 _reportTimestamp
) internal view {
  (, , , uint256 requestTimestampToFinalizeUpTo, , ) = IWithdrawalQueue(_withdrawalQueue)
     .getWithdrawalRequestStatus(_requestIdToFinalizeUpTo);
  if (_reportTimestamp < requestTimestampToFinalizeUpTo + _limitsList.requestTimestampMargin)
     revert IncorrectRequestFinalization(requestTimestampToFinalizeUpTo);
}
```

Add the function **getWithdrawalRequestStatus** to the WithdrawalQueue contract.

For example:

```
function getWithdrawalRequestStatus(uint256 _requestId)
  external
  view
  returns (WithdrawalRequestStatus memory)
{
  return _getStatus(_requestId);
}
```

LID-2. BAD ACTOR CAN BLOCK LIDO DEPOSIT FUNCTION

SEVERITY: High

PATH: Lido.sol:deposit:L696-730, StakingRouter.sol:receiv:L153-155

REMEDIATION: move the calculation of variable unaccountedEth after calling the function IStakingRouter(locator.stakingRouter()).deposit.value(depositableEth). After that, the assert on the line 727 can be removed, as it will become redundant

STATUS: fixed

DESCRIPTION:

Using the **Lido:deposit** (L696-730), the Lido protocol makes deposits to the Ethereum **DepositContract** using the **StakingRouter** contract, which is inherited from **BeaconChainDepositor** contract.

The **deposit** function creates a variable **unaccountedEth** (L709) using the function **Lido.sol:_getUnaccountedEther** (L1118-1120) before calling to the StakingRouter. This function does not take the Staking Router's ETH balance into account, while the Staking Router does send any dust back to Lido after calling **deposit**.

The bad actor can send a certain amount of ETH to the **StakingRouter** contract using the **selfdestruct** function on the attacker contract (also, after EIP-4758, it will be possible to use the sendall, instead of selfdestruct).

When the dust is returned to Lido, the subsequent call to **_getUnaccountedEther** will return a higher amount that previously saved. The assertion will fail and as a result, the **deposit** function because it will revert.

```
function deposit(uint256 _maxDepositsCount, uint256 _stakingModuleId, bytes _depositCalldata) external {
  ILidoLocator locator = getLidoLocator();

  require(msg.sender == locator.depositSecurityModule(), "APP_AUTH_DSM_FAILED");
  require(_stakingModuleId <= uint24(-1), "STAKING_MODULE_ID_TOO_LARGE");
  require(canDeposit(), "CAN_NOT_DEPOSIT");

  uint256 depositableEth = getDepositableEther();

  if (depositableEth > 0) {
    /// available ether amount for deposits (multiple of 32eth)
    depositableEth = Math256.min(depositableEth, _maxDepositsCount.mul(DEPOSIT_SIZE));

    uint256 unaccountedEth = _getUnaccountedEther();
    /// @dev transfer ether to SR and make deposit at the same time
    /// @notice allow zero value of depositableEth, in this case SR will simply transfer the unaccounted ether to Lido contract
    uint256 depositsCount = IStakingRouter(locator.stakingRouter()).deposit.value(depositableEth)(
      _maxDepositsCount,
      _stakingModuleId,
      _depositCalldata
    );

    uint256 depositedAmount = depositsCount.mul(DEPOSIT_SIZE);
    assert(depositedAmount <= depositableEth);

    if (depositsCount > 0) {
      uint256 newDepositedValidators = DEPOSITED_VALIDATORS_POSITION.getStorageUint256().add(depositsCount);
      DEPOSITED_VALIDATORS_POSITION.setStorageUint256(newDepositedValidators);
      emit DepositedValidatorsChanged(newDepositedValidators);

      _markAsUnbuffered(depositedAmount);
      assert(_getUnaccountedEther() == unaccountedEth);
    }
  }
}
```

LID-20. SHARE RATE CAN BE NEGATIVELY INFLUENCED DUE TO SHARES BURN LIMIT

SEVERITY: High

PATH: PositiveTokenRebaseLimiter.sol:getSharesToBurnLimit: L119-136

REMEDIATION: remove the remainingRebase from the denominator in PositiveTokenRebaseLimiter.sol:getSharesToBurnLimit (L119-136)

STATUS: fixed

DESCRIPTION:

The PositiveTokenRebaseLimiter is responsible for limiting the increase of the stETH share rate in the positive direction. For example, if a lot of rewards come in from one oracle report, then the limiter would return how much ETH can be taken from the WithdrawalVault such that the share rate does not increase above the configured threshold.

In the testing environment, the default positive token rebase limit is set at 0.05%.

The PositiveTokenRebaseLimiter is also responsible for calculating the shares burn limit. For example, when a batch of withdrawal requests get finalised, the locked stETH shares have to be burned such that the withdrawer's rewards get distributed among the stakers.

However, the calculation of the shares burn limit is skewed and will result in a conservative limit amount.

First, Lido calls **smoothTokenRebase** on the OracleReportSanityChecker to calculate limits considering rewards and withdrawals.

This function first calculates the rewards limit, then it calls **raiseLimit** on the PositiveTokenRebaseLimiter with the total amount of ETH for finalisation of the withdrawal requests. This raises the limit for the full ETH amount to be locked for withdrawal claiming.

But instead of returning the corresponding share amount, **getSharesToBurnLimit** will use the following **x / (1 + x)** formula:

```
rebaseLimit = requestedETH / totalETH
sharesBurnLimit = totalShares * (rebaseLimit / (1 + rebaseLimit))
```

Less shares are burned, while the full ETH amount is locked for withdrawal claiming. As a result, the share rate decreases.

Taking the 0.05% standard rebase limit into account, the effects start to show at a batch of withdrawal requests containing ETH of 2% of the total pooled ETH. However, if there are also consensus/execution layer rewards claimed in the same oracle report (this consume some/all of the 0.05% standard limit), then the effects could start to show at any withdrawal amount.

The share rate would lower, resulting in discounted stETH. The burning remaining shares are postponed to be burned at a later oracle report, resulting in a temporary higher APR.

Due to this bug, it becomes profitable to perform large withdrawals, buy shares at a discounted rate and earn more APR.

```
function getSharesToBurnLimit(TokenRebaseLimiterData memory _limiterState)
  internal
  pure
  returns (uint256 maxSharesToBurn)
{
  if (_limiterState.rebaseLimit == UNLIMITED_REBASE) {
     return _limiterState.totalShares;
  }

  uint256 remainingRebase = _limiterState.rebaseLimit - _limiterState.accumulatedRebase;
  maxSharesToBurn = (
     _limiterState.totalShares * remainingRebase
  ) / (LIMITER_PRECISION_BASE + remainingRebase);
}
```

LID-21. WITHDRAWALS ACCRUED REWARDS ARE NOT DISTRIBUTED AMONG STAKERS

SEVERITY: Medium

PATH: PositiveTokenRebaseLimiter.sol

REMEDIATION: if the distribution of the accrued rewards of withdrawal requests among stakers after finalisation of a withdrawal requests is not desired, then we would recommend to change this logic

One possible solution could be to immediately remove the ETH from the total pooled ETH and the shares from the total shares. This would not impact the share rate and any subsequent rewards will be divided only among active shares.

In the less likely scenario where the share rate drops after a withdrawal request, the withdrawer would only be able to claim the lower ETH amount after finalisation, just like now. The remaining ETH could be (slowly) added back to the total pooled ETH (or potentially used for finalisation of other withdrawal requests).

This solution is just to give an idea, any implementation would have to be reviewed thoroughly again

STATUS: fixed

DESCRIPTION:

The PositiveTokenRebaseLimiter calculates the rewards to be taken from the vaults and the amount of shares to be burned for withdrawal requests.

OracleReportSanityChecker initialises the PositiveTokenRebaseLimiter with the total pooled ETH and total shares at the moment of the oracle report. The shares to be burned are then calculated with a ratio of the total ETH in the withdrawal requests against the total pooled ETH.

For example:

```
sharesBurnLimit = (totalWithdrawalRequestETH / totalPooledETH) * totalShares
```

However, this total pooled ETH includes rewards that have been accrued during the processing of the withdrawal request, also the ETH in the withdrawal requests cannot be larger than the ETH value from when the requests were created which therefore will not include any rewards that would have been accrued.

As a result, the ratio that is calculated will never be higher than the current share rate and the shares that will be burned will always be less than the shares that have been locked during withdrawal request creation.

This means that any rewards accrued by the withdrawers will not be divided among the stakers that were active during the processing of the withdrawal request. Instead, the burning of the shares is postponed and the rewards will be divided among stakers that are active later.

This could give rise to profitable strategies were depositing after the finalisation of a large withdrawal gives higher APR.

```
TokenRebaseLimiterData memory tokenRebaseLimiter = PositiveTokenRebaseLimiter.initLimiterState(
  getMaxPositiveTokenRebase(),
  _preTotalPooledEther,
  _preTotalShares
);
```

LID-15. INACTIVE NODE OPERATOR CAN ALWAYS BLOCK DEPOSITS

SEVERITY: Medium

PATH: NodeOperatorRegistry.sol:_increaseValidatorsKeysNonce: L1315-1321

REMEDIATION: see description

STATUS: fixed

DESCRIPTION:

The function to deposit ETH to the DepositContract requires a set of signatures of the deposit data. One element is the staking module nonce, which has to be equal to the current on-chain nonce of the staking module.

This staking module nonce increases upon any deposit data change in the staking module. For example, in the NodeOperatorRegistry this happens whenever there is a data change for a node operator. Node operators can invoke these changes themselves by adding or removing their deposit keys, which then increases the nonce.

If a node operator would front-run or spam adding and removing keys, then the nonce would increase and the signatures of the deposit data would no longer be valid, causing a block of deposits.

Furthermore, a node operator that has been set inactive due to leaving or becoming malicious can still invoke these functions to add or remove keys, because it only checks whether the node operator ID exists, not whether it is still active.

As a result, any node operator that ever existed, including kicked ones, can still always block deposits of ETH to the deposit contract.

```
function _increaseValidatorsKeysNonce() internal {
  uint256 keysOpIndex = KEYS_OP_INDEX_POSITION.getStorageUint256() + 1;
  KEYS_OP_INDEX_POSITION.setStorageUint256(keysOpIndex);
  /// @dev [DEPRECATED] event preserved for tooling compatibility
  emit KeysOpIndexSet(keysOpIndex);
  emit NonceChanged(keysOpIndex);
}
```

Modify **_onlyNodeOperatorManager** such that the node operator manager can only call these kind of functions if the node operator is also active.

For example:

```
function _onlyNodeOperatorManager(address _sender, uint256 _nodeOperatorId) internal view {
  bool isRewardAddress = _sender == _nodeOperators[_nodeOperatorId].rewardAddress;
  bool isActive = _nodeOperators[_nodeOperatorId].active;
  _requireAuth((isRewardAddress && isActive) || canPerform(_sender, MANAGE_SIGNING_KEYS,
arr(_nodeOperatorId)));
}
```

LID-18. FINALISATION SHARE RATE SANITY CHECK DOES NOT TAKE POSTPONED SHARE BURNING INTO ACCOUNT

SEVERITY: Medium

PATH: Lido.sol:_handleOracleReport:L1187-1298

REMEDIATION: see description

STATUS: fixed

DESCRIPTION:

When an oracle report with a withdrawal request finalisation comes in, the corresponding ETH is sent to the Withdrawal Queue contract and the shares are burned through the Burner contract. The Burner contract uses a burn limit calculated from the ETH amount. The Burner contract can therefore have a backlog of shares to be burned in the future.

If there is a backlog of shares, then these will be burned first upon a withdrawal request, instead of the requested shares. Therefore **Lido.sol:_burnSharesLimited** (L1366-1390) could return 0 if the backlog is as large as the to-be-burned shares for the withdrawal requests, because it does not count the previously postponed shares.

Later in the execution of **_handleOracleReport** on line 1290, the Oracle Report Sanity Checker is called to check the finalisation share rate with the total requested ETH amount and burned shares amount as deltas.

However, the rate will be miscalculated if there was a backlog of shares, because the burned shares delta will be lower and could be 0.

As a result, the check and the entire oracle report will fail the check in **OracleReportSanityChecker.sol:_checkSimulatedShareRate** if the shares backlog and withdrawal request are large enough to cause a deviation in the miscalculation.

```
function _checkSimulatedShareRate(
    LimitsList memory _limitsList,
    uint256 _noWithdrawalsPostTotalPooledEther,
    uint256 _noWithdrawalsPostTotalShares,
    uint256 _simulatedShareRate
  ) internal pure {
    uint256 actualShareRate = (
      _noWithdrawalsPostTotalPooledEther * SHARE_RATE_PRECISION_E27
    ) / _noWithdrawalsPostTotalShares;

    if (actualShareRate == 0) {
      // can't finalize anything if the actual share rate is zero
      revert IncorrectSimulatedShareRate(MAX_BASIS_POINTS);
    }

    uint256 simulatedShareDiff = Math256.abs(
      SafeCast.toInt256(_simulatedShareRate) - SafeCast.toInt256(actualShareRate)
    );
    uint256 simulatedShareDeviation = (MAX_BASIS_POINTS * simulatedShareDiff) / actualShareRate;

    if (simulatedShareDeviation > _limitsList.simulatedShareRateDeviationBPLimit) {
      revert IncorrectSimulatedShareRate(simulatedShareDeviation);
    }
  }
```

Change the calculation for **burntCurrentWithdrawalShares** in **Lido.sol:_burnSharesLimited** such that it takes postponed shares into account.

For example:

```
burntCurrentWithdrawalShares = Math.min(_sharesToBurnFromWithdrawalQueue, sharesCommittedToBurnNow);
```

LID-5. DEPOSIT CALL DATA NOT INCLUDED IN GUARDIAN SIGNATURE

SEVERITY: Medium

PATH: DepositSecurityModule.sol:depositBufferedEther:L413-439

REMEDIATION: make the hash of the parameter (keccak256(depositCalldata)) part of the signature so it cannot be tampered with

STATUS: acknowledged, see commentary

DESCRIPTION:

The function **depositBufferedEther** is used to make the Lido contract deposit ETH to the deposit contract and create new nodes. For this a set of valid guardian signatures is required.

However, the guardian signature only contains the block information, staking module ID, root and nonce. It does not include the parameter **depositCalldata**.

This parameter gets passed to Lido and there it gets passed to the Staking Router, which in turns passes it to the right staking module with **obtainDepositData(maxDepositsCount, _depositCalldata)** to obtain the public keys for the deposit contract.

Because the parameter is not part of the signature, it becomes possible for a malicious user to front-run the transaction and submit the signatures with arbitrary **depositCalldata**.

This can become a problem if the staking module uses this data to derive the public keys. However, **NodeOperatorRegistry.sol** currently ignores this parameter, only **ModuleSolo.sol** (mock contract) directly decodes the data into public keys.

```
function depositBufferedEther(
  uint256 blockNumber,
  bytes32 blockHash,
  bytes32 depositRoot,
  uint256 stakingModuleId,
  uint256 nonce,
  bytes calldata depositCalldata,
  Signature[] calldata sortedGuardianSignatures
) external validStakingModuleId(stakingModuleId) {
  if (quorum == 0 || sortedGuardianSignatures.length < quorum) revert DepositNoQuorum();

  bytes32 onchainDepositRoot = IDepositContract(DEPOSIT_CONTRACT).get_deposit_root();
  if (depositRoot != onchainDepositRoot) revert DepositRootChanged();

  if (!STAKING_ROUTER.getStakingModuleIsActive(stakingModuleId)) revert DepositInactiveModule();

  uint256 lastDepositBlock = STAKING_ROUTER.getStakingModuleLastDepositBlock(stakingModuleId);
  if (block.number - lastDepositBlock < minDepositBlockDistance) revert DepositTooFrequent();
  if (blockHash == bytes32(0) || blockhash(blockNumber) != blockHash) revert DepositUnexpectedBlockHash();

  uint256 onchainNonce = STAKING_ROUTER.getStakingModuleNonce(stakingModuleId);
  if (nonce != onchainNonce) revert DepositNonceChanged();

  _verifySignatures(depositRoot, blockNumber, blockHash, stakingModuleId, nonce, sortedGuardianSignatures);

  LIDO.deposit(maxDepositsPerBlock, stakingModuleId, depositCalldata);
}

function _verifySignatures(
  bytes32 depositRoot,
  uint256 blockNumber,
  bytes32 blockHash,
  uint256 stakingModuleId,
  uint256 nonce,
  Signature[] memory sigs
) internal view {
  bytes32 msgHash = keccak256(
     abi.encodePacked(ATTEST_MESSAGE_PREFIX, blockNumber, blockHash, depositRoot, stakingModuleId, nonce)
  );

  address prevSignerAddr = address(0);

  for (uint256 i = 0; i < sigs.length; ++i) {
     address signerAddr = ECDSA.recover(msgHash, sigs[i].r, sigs[i].vs);
     if (!_isGuardian(signerAddr)) revert InvalidSignature();
     if (signerAddr <= prevSignerAddr) revert SignatureNotSorted();
     prevSignerAddr = signerAddr;
  }
}
```

Commentary from client:

"In the current design, the ***depositCalldata*** *argument is the data might be necessary to prepare the deposit data by the staking module. Depending on the staking module implementation, it might be empty (****NodeOperatorsRegistry****) or contain complete deposit data (hypothetical staking module implementation that uses offchain storage for the deposit data). But for any staking module, the following is* ***ALWAYS*** *true: passed* ***depositCalldata MUST NOT*** *affect the deposit data set of the staking module (This requirement was explicitly added in the comments of* ***IStakingModule.obtainDepositData()*** *method in the* commit *). In other words, call of* ***StakingModule.obtainDepositData()*** *with different valid* ***depositCalldata*** *for the same* ***StakingModule.nonce*** *leads to the same unambiguous set of deposited validators. For example, the offchain staking module might store a hash of the expected* ***depositCalldata*** *onchain to validate it and reject any call with unexpected* ***depositCalldata****."*

LID-12. LIDO DOES NOT CHECK ACTUAL AMOUNT FROM LIDOEXECUTIONLAYERREWARDSVAULT REWARDS WITHDRAWAL

SEVERITY: Medium

PATH: Lido.sol:_collectRewardsAndProcessWithdrawals:L827-860

REMEDIATION: see description

STATUS: fixed

DESCRIPTION:

Part of the Oracle report processing, is the withdrawal of rewards from the LidoExecutionLayerRewardsVault, which happens in **_collectRewardsAndProcessWithdrawals** on line 835-837.

The LidoExecutionLayerRewardsVault's withdrawal function **withdrawRewards** takes a maximum amount parameter, against which a saturated subtraction is done with the vault's ETH balance. The actual ETH amount is sent back to Lido and returned as function return value.

However, Lido does not check the return value and uses the max amount as received value, even though this value may be higher than the actual received value.

Furthermore, the buffered ETH counter gets decreased with this max amount, which could potentially cause it to go out-of-sync and ETH to get lost.

```
if (_elRewardsToWithdraw > 0) {

ILidoExecutionLayerRewardsVault(_contracts.elRewardsVault).withdrawRewards(_elRewardsToWithdraw);
}
```

```
function withdrawRewards(uint256 _maxAmount) external returns (uint256 amount) {
  require(msg.sender == LIDO, "ONLY_LIDO_CAN_WITHDRAW");

  uint256 balance = address(this).balance;
  amount = (balance > _maxAmount) ? _maxAmount : balance;
  if (amount > 0) {
     ILido(LIDO).receiveELRewards{value: amount}();
  }
  return amount;
}
```

Replace line 841 with:

```
_elRewardsToWithdraw =
ILidoExecutionLayerRewardsVault(_contracts.elRewardsVault).withdrawRewards(_elRewardsToWithdraw);
```

LID-3. GAS OPTIMISATION OF CANDEPOSIT

SEVERITY: Low

PATH: DepositSecurityModule.sol:canDeposit, Lido.sol:canDeposit (L385-395, L672-674)

REMEDIATION: see description

STATUS: acknowledged, see commentary

DESCRIPTION:

The function **canDeposit** in both contracts checks whether it is currently allowed to deposit to the deposit contract. Both checks contain conditionals over multiple variables, including external calls. However, the conditional elements are not ordered by cost or likelihood, so the cost of the functions increases as an early-exit does not improve the cost.

For example, if the first element in a conditional with multiple AND's, then Solidity will early-exit by returning **false** and the other elements won't be evaluated, saving a lot of gas if those are external calls. The same applies to multiple OR's with **true**.

```
function canDeposit(uint256 stakingModuleId) external view validStakingModuleId(stakingModuleId) returns (bool) {
  bool isModuleActive = STAKING_ROUTER.getStakingModuleIsActive(stakingModuleId);
  uint256 lastDepositBlock = STAKING_ROUTER.getStakingModuleLastDepositBlock(stakingModuleId);
  bool isLidoCanDeposit = LIDO.canDeposit();
  return (
     isModuleActive
     && quorum > 0
     && block.number - lastDepositBlock >= minDepositBlockDistance
     && isLidoCanDeposit
  );
}
```

```
function canDeposit() public view returns (bool) {
    return !IWithdrawalQueue(getLidoLocator().withdrawalQueue()).isBunkerModeActive() &&
!isStopped();
}
```

For **DepositSecurityModule.sol:canDeposit** move the external calls into the conditional of the return statement, instead of first saving it in variables.

Then these conditionals should be ordered, such that cheaper elements or elements that are more likely to return false are put first, i.e. external calls should always be last.

For example:

```
function canDeposit(uint256 stakingModuleId) external view validStakingModuleId(stakingModuleId) returns (bool) {
  bool isModuleActive = ;
  uint256 lastDepositBlock = ;
  bool isLidoCanDeposit = ;
  return (
     quorum > 0
     && STAKING_ROUTER.getStakingModuleIsActive(stakingModuleId)
     && block.number - STAKING_ROUTER.getStakingModuleLastDepositBlock(stakingModuleId) >= minDepositBlockDistance
     && LIDO.canDeposit()
  );
}
```

```
function canDeposit() public view returns (bool) {
    return !isStopped() &&
!IWithdrawalQueue(getLidoLocator().withdrawalQueue()).isBunkerModeActive();
}
```

Commentary from client:

"We assume that total costs of deposit an order of magnitude higher than this gas optimisation."

LID-4. DEPOSITSECURITYMODULE REDUNDANT STAKING MODULE ID CHECKS

SEVERITY: Low

PATH: DepositSecurityModule.sol:pauseDeposits, unpauseDeposits, canDeposit, depositBufferedEther (L336-365, L372-378, L385-395, L413-439)

REMEDIATION: the check only covers the edge case where the module staking ID is much too large. During normal use of the protocol this edge case won't occur so the extra check for an early-exit is not beneficial

therefore we would recommend to remove the modifier validStakingModuleId in favour of contract size and gas savings

STATUS: fixed

DESCRIPTION:

The modifier **validStakingModuleId** in **DepositSecurityModule.sol** is redundant. Each of the functions that make use of this modifier all directly call into the Staking Router with the same provided **moduleStakingId**. The Staking Router has the same modifier and therefore already performs the check.

```
modifier validStakingModuleId(uint256 _stakingModuleId) {
    if (_stakingModuleId > type(uint24).max) revert StakingModuleIdTooLarge();
    _;
}
```

LID-8. GAS OPTIMISATION OF WITHDRAWAL QUEUE INITIALISATION

SEVERITY: Low

PATH: WithdrawalQueueBase.sol:_initializeQueue:L470-476

REMEDIATION: see description

STATUS: acknowledged, see commentary

DESCRIPTION:

The function **_initializeQueue** is called upon initialisation of the Withdrawal Queue contract. We found that it could be further gas optimised.

1. A zero struct WithdrawalRequest is created and written to the 0th key in the queue, however the first 2 elements that occupy the first storage slot are default values (**0**). Only the timestamp and claim marker are set, which occupy the second storage slot. It is therefore cheaper to directly write to these values:

```
_getQueue()[0].timestamp = uint64(block.timestamp);
_getQueue()[0].claimed = true;
```

2. A zero struct **DiscountCheckpoint** with default values (**0**) for all elements is written to the 0th key in the checkpoints. Because it is a mapping, the 0th index would already return a **DiscountCheckpoint** with default values upon reading. Therefore, the assignment can be completely removed.

```
function _initializeQueue() internal {
  // setting dummy zero structs in checkpoints and queue beginning
  // to avoid uint underflows and related if-branches
  // 0-index is reserved as 'not_found' response in the interface everywhere
  _getQueue()[0] = WithdrawalRequest(0, 0, payable(0), uint64(block.number), true);
  _getCheckpoints()[getLastCheckpointIndex()] = DiscountCheckpoint(0, 0);
}
```

Commentary from client:

“The suggested gas optimization has a little impact on costs and impairs code readability. We decided to prioritize the code clarity, but appreciate your input.”

LID-9. WITHDRAWALREQUESTERC721 BALANCE DOES NOT DECREASE AFTER CLAIMING

SEVERITY: Low

PATH: WithdrawalRequestERC721.sol:balanceOf:L112-115

REMEDIATION: remove the withdrawal request from the requestsByOwner mapping upon claiming. The request and its data will still available in the queue mapping

STATUS: fixed, commits: 1, 2

DESCRIPTION:

A withdrawal request implements ERC721 and so acts as an NFT. Once a withdrawal request is claimed, it cannot be transferred anymore and NFT should have been burned.

This is also apparent from **WithdrawalQueue.sol:claimWithdrawal** (L214-238) where a Transfer event from the owner to **address(0)** for the request ID is emitted, which suggests a burn of the withdrawal request NFT.

However, we found that claimed withdrawals are still counted in the NFT balance of a user when calling **balanceOf**. This is due to the withdrawal request not being removed from the **requestsByOwner** mapping.

```
function balanceOf(address _owner) external view override returns (uint256) {
  if (_owner == address(0)) revert InvalidOwnerAddress(_owner);
  return _getRequestsByOwner()[_owner].length();
}
```

LID-16. ADDING A NODE OPERATOR DOES NOT INCREASE THE NONCE

SEVERITY: Low

PATH: NodeOperatorRegistry.sol:addNodeOperator:L302-322

REMEDIATION: add a call to _increaseValidatorsKeysNonce() at the end of addNodeOperator

STATUS: acknowledged, see commentary

DESCRIPTION:

The function to add a new node operator to the node operator registry does not increase the staking module nonce, even though this nonce should increase upon the activation of a node operator according to the spec.

```
function addNodeOperator(string _name, address _rewardAddress) external returns (uint256 id) {
  _onlyValidNodeOperatorName(_name);
  _onlyNonZeroAddress(_rewardAddress);
  _auth(MANAGE_NODE_OPERATOR_ROLE);

  id = getNodeOperatorsCount();
  require(id < MAX_NODE_OPERATORS_COUNT, "MAX_OPERATORS_COUNT_EXCEEDED");

  TOTAL_OPERATORS_COUNT_POSITION.setStorageUint256(id + 1);

  NodeOperator storage operator = _nodeOperators[id];

  uint256 activeOperatorsCount = getActiveNodeOperatorsCount();
  ACTIVE_OPERATORS_COUNT_POSITION.setStorageUint256(activeOperatorsCount + 1);

  operator.active = true;
  operator.name = _name;
  operator.rewardAddress = _rewardAddress;

  emit NodeOperatorAdded(id, _name, _rewardAddress, 0);
}
```

Commentary from client:

"It's assumed that a nonce can only be changed when changes applied in the operator validators subset, therefore the addNodeOperator method does not modify the operator validators subset and, as a result, it does not affect the nonce."

Useful links: LIP-5 and Code-snippet

LID-10. STAKING ROUTER REDUNDANT MODULE ID CHECKS

SEVERITY: Low

PATH:
StakingRouter.sol:getStakingModuleIsStopped,getStakingModuleIsDepositsPaused,getStakingModuleIsActive (L781-786, L788-793, L795-800).

REMEDIATION: remove the validStakingModuleId modifier from getStakingModuleIsStopped, getStakingModuleIsDepositsPaused and getStakingModuleIsActive

STATUS: fixed

DESCRIPTION:

The modifier **validStakingModuleId** checks whether the **_stakingModuleId** is a valid ID. However, the modifier is repeated in the function chain for the status functions.

getStakingModuleIsStopped, **getStakingModuleIsDepositsPaused** and **getStakingModuleIsActive** have the **validStakingModuleId** modifier and they all directly call **getStakingModuleStatus** which also has the modifier. As a result, the check is performed twice.

```
function getStakingModuleIsStopped(uint256 _stakingModuleId) external view
  validStakingModuleId(_stakingModuleId)
  returns (bool)
{
  return getStakingModuleStatus(_stakingModuleId) == StakingModuleStatus.Stopped;
}

function getStakingModuleIsDepositsPaused(uint256 _stakingModuleId) external view
  validStakingModuleId(_stakingModuleId)
  returns (bool)
{
  return getStakingModuleStatus(_stakingModuleId) == StakingModuleStatus.DepositsPaused;
}

function getStakingModuleIsActive(uint256 _stakingModuleId) external view
  validStakingModuleId(_stakingModuleId)
  returns (bool)
{
  return getStakingModuleStatus(_stakingModuleId) == StakingModuleStatus.Active;
}
```

LID-14. REDUNDANT STORAGE WRITE

SEVERITY: Low

PATH:
StakingRouter.sol:updateExitedValidatorsCountByStakingModule:
L271-303

REMEDIATION: wrap the storage write in a branch statement to check whether they are not the same

STATUS: acknowledged, see commentary

DESCRIPTION:

The function updates the exited validator count for a staking module. The function only checks whether the reported value is less than the previous value and if so reverts.

However, if the newly reported value and the previous value are the same, then the function writes the same value back to **stakingModule.exitedValidatorsCount** resulting in the waste of a storage write.

```
function updateExitedValidatorsCountByStakingModule(
    uint256[] calldata _stakingModuleIds,
    uint256[] calldata _exitedValidatorsCounts
  )
    external
    onlyRole(REPORT_EXITED_VALIDATORS_ROLE)
  {
    for (uint256 i = 0; i < _stakingModuleIds.length; ) {
      StakingModule storage stakingModule = _getStakingModuleById(_stakingModuleIds[i]);
      uint256 prevReportedExitedValidatorsCount = stakingModule.exitedValidatorsCount;
      if (_exitedValidatorsCounts[i] < prevReportedExitedValidatorsCount) {
        revert ErrorExitedValidatorsCountCannotDecrease();
      }

      (
        uint256 totalExitedValidatorsCount,
        /* uint256 totalDepositedValidators */,
        /* uint256 depositableValidatorsCount */
      ) = IStakingModule(stakingModule.stakingModuleAddress).getStakingModuleSummary();

      if (totalExitedValidatorsCount < prevReportedExitedValidatorsCount) {
        // not all of the exited validators were async reported to the module
        emit StakingModuleExitedValidatorsIncompleteReporting(
          stakingModule.id,
          prevReportedExitedValidatorsCount - totalExitedValidatorsCount
        );
      }
      stakingModule.exitedValidatorsCount = _exitedValidatorsCounts[i];
      unchecked { ++i; }
    }
  }
```

Commentary from client:

“We would like to emphasize that Accounting Oracle passes only the changed data for this part of the report. As such, we do not believe that this issue or gas optimisation requires fixing.”

LID-22. REBASE LIMITER ISLIMITREACHED WILL ALMOST ALWAYS RETURN FALSE DUE TO ROUNDING ERROR

SEVERITY: Low

PATH: PositiveTokenRebaseLimiter.sol:isLimitReached:L103-115

REMEDIATION: either remove the check, as the limit is already enforced in `increaseEther` or to round the calculations up

STATUS: fixed

DESCRIPTION:

The function **isLimitReached** is used in **getSharesToBurnLimit** to provide an early return if the rebase limit is already reached and it can just return 0. However, we found that this function will almost always return **false**, due to rounding errors.

In **isLimitReached**, the difference between the current ETH and pre-total ETH is taken as the accumulated ETH. This accumulated ETH is then divided over the pre-total ETH to get the new rebase. The result of the function is whether this new rebase is greater than the rebase limit.

But this will almost never be the case, because **increaseEther** is the only function that increases the current ETH and already takes the maximum against the rebase limit. This calculation is rounded down and as a result will always be smaller than the actual maximum if it's not a perfect division without remainder.

Secondly, the calculation for the new rebase in **isLimitReached** is also rounded down and will also almost always be smaller than the rebase limit.

As a result, the check becomes redundant in its current form.

```
function isLimitReached(TokenRebaseLimiterData memory _limiterState) internal pure returns (bool) {
   if (_limiterState.positiveRebaseLimit == UNLIMITED_REBASE) return false;
   if (_limiterState.currentTotalPooledEther < _limiterState.preTotalPooledEther) return false;

   uint256 accumulatedEther = _limiterState.currentTotalPooledEther - _limiterState.preTotalPooledEther;
   uint256 accumulatedRebase;

   if (_limiterState.preTotalPooledEther > 0) {
       accumulatedRebase = accumulatedEther * LIMITER_PRECISION_BASE /
_limiterState.preTotalPooledEther;
   }

   return accumulatedRebase >= _limiterState.positiveRebaseLimit;
}
```

LID-23. REBASE LIMITER MAXTOTALPOOLEDETHER OPTIMISATION

SEVERITY: Low

PATH: PositiveTokenRebaseLimier.sol:increaseEther:L139-160

REMEDIATION: perform the change as described in the description in favour of gas optimisation

STATUS: fixed

DESCRIPTION:

The calculation for **maxTotalPooledEther** in **increaseEther** depends fully on two variables that are only set during initialisation and won't change any further. The calculation can therefore be moved to initialisation as a memory variable in **TokenRebaseLimiterData**. This is because the result will always be the same in every subsequent call to **increaseEther**.

```
function increaseEther(
  TokenRebaseLimiterData memory _limiterState, uint256 _etherAmount
)
  internal
  pure
  returns (uint256 consumedEther)
{
  if (_limiterState.positiveRebaseLimit == UNLIMITED_REBASE) return _etherAmount;

  uint256 prevPooledEther = _limiterState.currentTotalPooledEther;
  _limiterState.currentTotalPooledEther += _etherAmount;

  uint256 maxTotalPooledEther = _limiterState.preTotalPooledEther +
     (_limiterState.positiveRebaseLimit * _limiterState.preTotalPooledEther) / LIMITER_PRECISION_BASE;

  _limiterState.currentTotalPooledEther
     = Math256.min(_limiterState.currentTotalPooledEther, maxTotalPooledEther);

  assert(_limiterState.currentTotalPooledEther >= prevPooledEther);

  return _limiterState.currentTotalPooledEther - prevPooledEther;
}
```

LID-24. WITHDRAWAL QUEUE ERC721 OWNEROF GAS OPTIMISATION

SEVERITY: Low

PATH: WithdrawalQueueERC721.sol:ownerOf:L156-163

REMEDIATION: apply the optimisation as shown in the description, in favour of gas optimisation

STATUS: fixed

DESCRIPTION:

In the **ownerOf** function, the **WithdrawalRequest** of the corresponding **requestId** is retrieved. The request is loaded fully into memory and both storage slots will be loaded.

However, only the values **claimed** and **owner** are used, which are both in the second slot only. Therefore, by replacing **WithdrawalRequest** memory with **WithdrawalRequest storage** on line 159, one **SLOAD** can be saved on each call.

```
function ownerOf(uint256 _requestId) public view override returns (address) {
   if (_requestId == 0 || _requestId > getLastRequestId()) revert InvalidRequestId(_requestId);

   WithdrawalRequest memory request = _getQueue()[_requestId];
   if (request.claimed) revert RequestAlreadyClaimed(_requestId);

   return request.owner;
}
```

LID-25. ANYONE CAN ALWAYS TRANSFER 1 WEI STETH

SEVERITY: Low

PATH: StETH.sol:getSharesByPooledEth:L311-315

REMEDIATION: see description

STATUS: acknowledged, see commentary

DESCRIPTION:

The functions **getSharesByPooledEth** and **getPooledEthByShares** are used to convert from stETH to shares and vice versa. Normally, the pooled ETH amount is greater than the number of shares.

Both functions round down and as a result one would get 0 shares for 1 wei stETH. This leads to some undesired behaviour in the **ERC20.transfer** functions of stETH.

For example, when calling **stETH.transfer(address(1337), 1)**, the transfer would succeed even though I would not have any stETH balance.

This is due to stETH using shares as underlying balance and the conversion leads to a transfer of 0. However, an event with the stETH amount is still emitted: **Transfer(address(this), address(1337), 1)**. This can potentially cause problems with front-ends or trackers.

```
function getSharesByPooledEth(uint256 _ethAmount) public view returns (uint256) {
  return _ethAmount
      .mul(_getTotalShares())
      .div(_getTotalPooledEther());
}
function _transfer(address _sender, address _recipient, uint256 _amount) internal {
  uint256 _sharesToTransfer = getSharesByPooledEth(_amount);
  _transferShares(_sender, _recipient, _sharesToTransfer);
  _emitTransferEvents(_sender, _recipient, _amount, _sharesToTransfer);
}
function _emitTransferEvents(address _from, address _to, uint _tokenAmount, uint256 _sharesAmount)
internal {
  emit Transfer(_from, _to, _tokenAmount);
  emit TransferShares(_from, _to, _sharesAmount);
}
```

We would recommend to consider adding an early exit if the shares amount is 0, instead of reverting.

For example:

```
function _transfer(address _sender, address _recipient, uint256 _amount) internal {
  uint256 _sharesToTransfer = getSharesByPooledEth(_amount);
  if (_sharesToTransfer == 0)
      return;
  _transferShares(_sender, _recipient, _sharesToTransfer);
  _emitTransferEvents(_sender, _recipient, _amount, _sharesToTransfer);
}
```

Commentary from client:

"The issue is related to the known rounding down integer math cases: https://github.com/lidofinance/lido-dao/issues/442

This peculiarity doesn't pose risks to the protocol or its users while having a tolerable negative impact on user experience."

LID-19. USERS WOULD LOSE THEIR FUNDS IF CLAIMING A WITHDRAWAL REQUEST DIRECTLY TO LIDO

SEVERITY: Informational

PATH: WithdrawalQueue.sol:claimWithdrawalsTo:L246-255

REMEDIATION: add an assertion to check if _recipient is not equal the Lido address

STATUS: acknowledged, see commentary

DESCRIPTION:

The function **claimWithdrawalsTo** allows a user to claim their withdrawal requests and send the ETH amount of another address.

The Lido contract allows users to mint shares by simply sending ETH due to the fallback function.

In case a user wants to claim their withdrawal and immediately stake it again, they might think that inputting the Lido address into **_recipient** would work. But it wouldn't work and as a result Lido will mint StETH not for user but for the WithdrawalQueue, and the user would lose their ETH.

```
function claimWithdrawalsTo(uint256[] calldata _requestIds, uint256[] calldata _hints, address _recipient)
  external
{
  if (_recipient == address(0)) revert ZeroRecipient();

  for (uint256 i = 0; i < _requestIds.length; ++i) {
     _claim(_requestIds[i], _hints[i], _recipient);
     _emitTransfer(msg.sender, address(0), _requestIds[i]);
  }
}
```

Commentary from client:

“It’s expected behaviour since the protocol can’t enforce the exhaustive set of addresses for this check”

LID-13. FRONTRUNNING INITIALISATION

SEVERITY: Informational

1. PATH: AccountingOracle.sol:initialize:L161-170
2. PATH: AccountingOracle.sol:initializeWithoutMigration: L172-181

REMEDIATION: use a factory contract for proxies that creates and initialises the contract in the same transaction and/or to verify the configured variables (such as owner) right after deployment and initialisation of the contract

STATUS: acknowledged, see commentary

DESCRIPTION:

Any attacker can frontrun the initialisation functions and take control of the contract right after creation.

AccountingOracle.sol:initialize

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

    uint256 lastProcessingRefSlot = _checkOracleMigration(LEGACY_ORACLE, consensusContract);
    _initialize(admin, consensusContract, consensusVersion, lastProcessingRefSlot);
  }
```

AccountingOracle.sol:initializeWithoutMigration

```
function initializeWithoutMigration(
    address admin,
    address consensusContract,
    uint256 consensusVersion,
    uint256 lastProcessingRefSlot
  ) external {
    if (admin == address(0)) revert AdminCannotBeZero();

    _initialize(admin, consensusContract, consensusVersion, lastProcessingRefSlot);
  }
```

Commentary from client:

“Thank you for your attention to this matter. We take great care in ensuring the safety and reliability of our upgrade process. To this end, we have developed an dedicated template contract that allows us performing all necessary operations - such as creation, initialization, and configuration - atomically. This approach mitigates potential risks and ensures the seamless and secure upgrade of the protocol.”

LID-6. CODE STYLE PROBLEMS

SEVERITY: Informational

1. PATH: Lido.sol:_getCurrentStakeLimit , _initialize_v2 (L1159-1164, L266-277). There are two methods of getting the maximum value of uint256 in one contract:
   a. uint256(-1)
   b. ~uint256(0)
2. PATH: NodeOperatorsRegistry.sol, LegacyOracle.sol, Versioned.sol, Lido.sol, StETHPermit.sol, AccountingOracle.sol, BaseOracle.sol, HashConsensus.sol, ValidatorsExitBusOracle.sol, OracleReportSanityChecker.sol, Burner.sol, StakingRouter.sol, WithdrawalQueue.sol, WithdrawalQueueBase.sol, WithdrawalRequestNFT.sol. There are two methods of setting the slot/role hash to a constant variable in various contracts.
   a. bytes32 internal constant SOME_VAR = keccak256("SOME_VAR_VALUE")
   b. bytes32 internal constant SOME_VAR = d9fe083e02cff7a67080d5a6363878406eed2fcb0c90803 30f30d216c0f1574d
3. PATH: WithdrawalQueueBase.sol:_initializeQueue() (L482-488). The timestamp argument is set incorrectly during initialisation. it must be block.timestamp, but it is set to block.number).

4. **PATH**: WithdrawalQueueERC721.sol:name (L98-100). WithdrawalQueueERC721 contract is inherited from the IERC721Metadata interface (all interface functions are implicitly virtual), so the function declaration should have the override keyword (symbol() and tokenURI() have that keyword).

REMEDIATION:

1. Use ~uint256(0) instead of uint256(-1)
2. Use only one method to set slot/role hash to variable.
3. To change block.number to block.timestamp
4. Add override keyword for name()

STATUS: fixed

DESCRIPTION:

We found that there are several places in the code where the code style differs.

1.

```
contract Lido is Versioned, StETHPermit, AragonApp {
  [...]
  function _initialize_v2(address _lidoLocator, address _eip712StETH) internal {
        _setContractVersion(2);

        LIDO_LOCATOR_POSITION.setStorageAddress(_lidoLocator);
        _initializeEIP712StETH(_eip712StETH);

        // set unlimited allowance for burner from withdrawal queue
        // to burn finalized requests' shares
        _approve(ILidoLocator(_lidoLocator).withdrawalQueue(), ILidoLocator(_lidoLocator).burner(), ~uint256(0));

        emit LidoLocatorSet(_lidoLocator);
    }
  [...]
  function _getCurrentStakeLimit(StakeLimitState.Data memory _stakeLimitData) internal view returns (uint256) {
        if (_stakeLimitData.isStakingPaused()) {
            return 0;
        }
        if (!_stakeLimitData.isStakingLimitSet()) {
            return uint256(-1);
        }

        return _stakeLimitData.calculateCurrentStakeLimit();
    }
  [...]
}
```

2.

```
contract Lido is Versioned, StETHPermit, AragonApp {
   [...]
   /// ACL
   bytes32 public constant PAUSE_ROLE = // @note hardcoded slot position hash
       0x139c2898040ef16910dc9f44dc697df79363da767d8bc92f2e310312b816e46d  ; //
keccak256("PAUSE_ROLE");
   [...]
}

contract StakingRouter is AccessControlEnumerable , BeaconChainDepositor , Versioned {
   [...]
   bytes32 public constant MANAGE_WITHDRAWAL_CREDENTIALS_ROLE =
keccak256("MANAGE_WITHDRAWAL_CREDENTIALS_ROLE" );
   [...]
```

3.

```
struct WithdrawalRequestStatus {
      /// @notice stETH token amount that was locked on withdrawal queue for this request
      uint256 amountOfStETH;
      /// @notice amount of stETH shares locked on withdrawal queue for this request
      uint256 amountOfShares;
      /// @notice address that can claim or transfer this request
      address owner;
      /// @notice timestamp of when the request was created, in seconds
      uint256 timestamp;
      /// @notice true, if request is finalized
      bool isFinalized;
      /// @notice true, if request is claimed. Request is claimable if (isFinalized && !isClaimed)
      bool isClaimed;
  }
```

```
function _initializeQueue() internal {
      // setting dummy zero structs in checkpoints and queue beginning
      // to avoid uint underflows and related if-branches
      // 0-index is reserved as 'not_found' response in the interface everywhere
      _getQueue()[0] = WithdrawalRequest(0, 0, payable(0), uint64(block.number), true);
      _getCheckpoints()[getLastCheckpointIndex()] = DiscountCheckpoint(0, 0);
  }
```

4.

```
function name() external view returns (string memory) {
      return _toString(NAME);
  }
```

LID-7. TYPOGRAPHICAL ERRORS

SEVERITY: Informational

PATH: see description

REMEDIATION:

1. OracleReportSanityChecker.sol
   a. Replace MANGER with MANAGER.
   b. Remove the extra _ROLE.

STATUS: fixed

DESCRIPTION:

There are typographical errors in:

1. OracleReportSanityChecker.sol
   a. The constant variable CHURN_VALIDATORS_PER_DAY_LIMIT_MANGER_ROLE and its value are misspelled on line 91 and 92.
   b. The value of the constant variable ONE_OFF_CL_BALANCE_DECREASE_LIMIT_MANAGER_ROLE is misspelled on line 94.

```
bytes32 public constant CHURN_VALIDATORS_PER_DAY_LIMIT_MANGER_ROLE =
  keccak256("CHURN_VALIDATORS_PER_DAY_LIMIT_MANGER_ROLE");
bytes32 public constant ONE_OFF_CL_BALANCE_DECREASE_LIMIT_MANAGER_ROLE =
  keccak256("ONE_OFF_CL_BALANCE_DECREASE_LIMIT_MANAGER_ROLE_ROLE");
```

LID-26. MISSING TWO-STEP OWNERSHIP TRANSFERRAL

SEVERITY: Informational

PATH: DepositSecurityModule.sol:setOwner:L140-142

REMEDIATION: implement a two-step owner transferral, where the current owner sets a pending owner and the pending owner would have to accept before becoming the current owner

STATUS: acknowledged, see commentary

DESCRIPTION:

The **setOwner** function is used to set a new owner of the contract.

If this function were to be called with an incorrect address, then the contract would be bricked. This is currently a single step/transaction.

```
function setOwner(address newValue) external onlyOwner {
  _setOwner(newValue);
}

function _setOwner(address _newOwner) internal {
  if (_newOwner == address(0)) revert ZeroAddress("_newOwner");
  owner = _newOwner;
  emit OwnerChanged(_newOwner);
}
```

Commentary from client:

*“The **setOwner** method can be used on behalf of the Lido DAO Agent contract that has a granted role which is a part of the whole protocol ACL setup. Therefore, the change requires an on-chain Aragon vote to enact and the Lido governance token holders accept associated risks of changing **owner** if support the vote.”*

hexens