Security Assessment & Formal Verification Draft Report

Lido Dual Governance

September 2024

Prepared for Lido

certora

LIDO

Table of content

Project Summary

Project Scope

Project Name	Repository (link)	Latest Commit Hash	Platform
Lido Dual Governance	https://github.com/lidofinance/dual-governance	Start: dfaf963 End : 071f033	EVM

Project Overview

This document describes the specification and verification of **Lido Dual Governance** using the Certora Prover and manual code review findings. The work was undertaken from **August 8 2024** to **September 5 2024**

The following contract list is included in our scope:

- contracts/Escrow.sol
- contracts/libraries/AssetsAccounting.sol
- contracts/DualGovernance.sol
- contracts/libraries/DualGovernanceStateMachine.sol
- contracts/EmergencyProtectedTimelock.sol
- contracts/libraries/WithdrawalBatchesQueue.sol
- contracts/committees/HashConsensus.sol
- contracts/libraries/ExecutableProposals.sol
- contracts/libraries/Tiebreaker.sol
- contracts/libraries/EmergencyProtection.sol
- contracts/libraries/DualGovernanceConfig.sol
- contracts/libraries/EscrowState.sol
- contracts/libraries/Proposers.sol
- contracts/types/Duration.sol
- contracts/committees/TiebreakerCore.sol
- contracts/committees/TiebreakerSubCommittee.sol
- contracts/committees/EmergencyExecutionCommittee.sol
- contracts/libraries/EnumerableProposals.sol
- contracts/DualGovernanceConfigProvider.sol
- contracts/types/Timestamp.sol
- contracts/libraries/TimelockState.sol

- contracts/committees/ResealCommittee.sol
- contracts/types/ETHValue.sol
- contracts/libraries/SealableCalls.sol
- contracts/ResealManager.sol
- contracts/types/SharesValue.sol
- contracts/committees/EmergencyActivationCommittee.sol
- contracts/TimelockedGovernance.sol
- contracts/types/PercentD16.sol
- contracts/types/IndexOneBased.sol
- contracts/committees/ProposalsList.sol
- contracts/libraries/ExternalCalls.sol
- contracts/Executor.sol
- contracts/utils/arrays.sol

The Certora Prover demonstrated that 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 all the Solidity contracts. During the verification process and the manual audit, the Certora team discovered bugs in the Solidity contracts code, as listed on the following page.

Please note that a few more formal rules are not included in this report, as they were proven with an unreleased version of the Certora Prover. Once those rules are proven on a released version of the Certora Prover, we will add them to the next version of this document.

Protocol Overview

Currently, the Lido protocol governance consists of the Lido DAO that uses LDO voting to approve DAO proposals, along with an optimistic voting subsystem called Easy Tracks that is used for routine changes of low-impact parameters and falls back to LDO voting given any objection from LDO holders.

Additionally, there is a Gate Seal emergency committee that allows pausing certain protocol functionality (e.g. withdrawals) for a pre-configured amount of time sufficient for the DAO to vote on and execute a proposal. The Gate Seal committee can only enact a pause once before losing its power (so it has to be re-elected by the DAO after that).

The Dual governance mechanism (DG) is an iteration on the protocol governance that gives stakers a say by allowing them to block DAO decisions and providing a negotiation device between stakers and the DAO.

Another way of looking at dual governance is that it implements 1) a dynamic user-extensible timelock on DAO decisions and 2) a rage quit mechanism for stakers taking into account the specifics of how Ethereum withdrawals work.

Findings Summary

The table below summarizes the findings of the review, including type and severity details.

Severity	Discovered	Confirmed	Fixed
Critical	2	2	2
High	6	6	6
Medium	11	11	10
Low	4	4	4
Total	23	23	22

Severity Matrix

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

certora

LIDO

Detailed Findings

ID	Title	Severity	Status
C-01	DOS on Proposers.sol::unregister due to the last proposer's ExecutorData not getting updated	Critical	Fixed
C-02	Evading the RageQuit's second seal	Critical	Fixed
H-01	getVetoerState() doesn't return the right unstETHLockedShares value	High	Fixed
H-02	Griefing _batchesQueue.close()	High	Fixed
H-03	cancelAllPendingProposals() is callable in any state	High	Fixed
H-04	tieBreakerResumeSealable has no call to activateNextState before checking for a tie	High	Fixed
H-05	RageQuit is DOS-d if the minimum amount of stEth to withdraw is 0	High	Fixed
H-06	RageQuit can fail to start even after the threshold have passed by frontrun	High	Fixed

M-01	HashConsensus.scheduledAt is not updated when the user changes their vote from true to false to true	Medium	Fixed
M-02	uint256 rageQuitRound = Math.min(self.rageQuitRound + 1, type(uint8).max); will revert when self.rageQuitRound == type(uint8).max	Medium	Fixed
M-03	Lack of access control for TiebreakerSubCommittee.sealableResume	Medium	Fixed
M-04	tieBreakerScheduleProposal does not trigger a state transition which will lead to wrong state post execution	Medium	Fixed
M-05	cancelAllPendingProposals does not trigger a state transition	Medium	Fixed
M-06	Front loading non-existing proposals	Medium	Fixed
M-07	Quorum change with pending votes	Medium	Fixed
M-08	Override proposal status	Medium	Fixed
M-09	Users may not get ProposalTimelock days to veto Proposals	Medium	Acknowledged

M-10	State transition to RageQuit isn't persisted when it should be	Medium	Fixed
M-11	requestWithdrawals() can be called whenRageQuit should have started	Medium	Fixed
L-01	removeSealableWithdrawalBlocker does not return a boolean or revert when failing to remove	Low	Fixed
L-02	lastAssetsLockTimestamp is updated even though unstEthIds = [ ]	Low	Fixed
L-03	withdrawETH is callable with unstEthIds = [ ]	Low	Fixed
L-04	SealableCalls.sol.callResume() isPaused flag is wrong	Low	Fixed

Critical Severity Issues

C-01 - DOS on `Proposers.sol::unregister` due to the last proposer's `ExecutorData` not getting updated

Severity: **Critical**	Impact: **High**	Likelihood: **High**
Files: DualGovernance.sol#L210-L218 Proposers.sol#L93-L109	Status: Certora awaiting response on Fix Review.	Violated Property: P-01. Proposer indexes match their index in the array and are always < the array length

Description:

In `Proposers.sol::unregister`, there's a swap and pop mechanism to replace in the `proposers` array the address that needs to be unregistered.
However, the swap is incomplete as the swapped proposer's `proposerIndex` field is not updated in the `ExecutorData`

Scenario

1. Initial setup:

```
proposer = [Alice, Bob, Celine, Dravee]
ExecutorData for Bob = {IndexOneBased: 2, executor: address(B)}
ExecutorData for Dravee = {IndexOneBased: 4, executor: address(D)}
```

2. After call to `Proposers.sol::unregister` on `Bob`, the final state is:

```
proposer = [Alice, Dravee, Celine]
ExecutorData for Bob = deleted
ExecutorData for Dravee = {IndexOneBased: 4, executor: address(D)}
```

3. Therefore, with a subsequent call to `unregister` for `Dravee`, there will be a revert due to an out-of-bound access on the array:

```
        if (executorData.proposerIndex != lastProposerIndex) { // <--------
true
            self.proposers[executorData.proposerIndex.toZeroBasedValue()] =
                self.proposers[lastProposerIndex.toZeroBasedValue()];
//<------ will revert due to out of bound on the array: `self.proposers[3] =
self.proposers[2]`
        }
```

Coded POC

The following test can be added in `Proposers.t.sol` and run with `forge test --mt test_DOSUnregister`

```
function test_DOSUnregister() external {
    _proposers.register(_ADMIN_PROPOSER, _ADMIN_EXECUTOR);

    address dravee = makeAddr("Dravee");
    address draveeExecutor = makeAddr("draveeExecutor");
    address alice = makeAddr("Alice");
    address aliceExecutor = makeAddr("aliceExecutor");
    address celine = makeAddr("Celine");
    address celineExecutor = makeAddr("celineExecutor");
    address bob = makeAddr("Bob");
    address bobExecutor = makeAddr("bobExecutor");

    _proposers.register(alice, aliceExecutor);
    _proposers.register(bob, bobExecutor);
    _proposers.register(celine, celineExecutor);
    _proposers.register(dravee, draveeExecutor);

    _proposers.unregister(bob);
    _proposers.unregister(dravee); // reverts with Reason: panic: array
out-of-bounds access
}
```

The test will fail with the message `[FAIL. Reason: panic: array out-of-bounds access (0x32)]`

Lido's response: Fixed in PR #104 with additional optimizations in PR #124

Certora's Fix Review: The fix in PR #104 can be gas optimized (saving a storage reading operation):

```
File: Proposers.sol
101:         if (executorData.proposerIndex != lastProposerIndex) {
+ 102:             address lastProposer =
self.proposers[lastProposerIndex.toZeroBasedValue()];
- 102:             self.proposers[proposerIndex.toZeroBasedValue()] =
self.proposers[lastProposerIndex.toZeroBasedValue()];
+ 103:             self.proposers[proposerIndex.toZeroBasedValue()] =
lastProposer;
- 103:
self.executors[self.proposers[proposerIndex.toZeroBasedValue()]].proposerIndex
= proposerIndex;
+ 104:             self.executors[lastProposer].proposerIndex = proposerIndex;
104:         }
```

C-02 - Evading the RageQuit's second seal

Severity: **Critical**	Impact: **High**	Likelihood: **High**
Files: Escrow.sol	Status: Fixed	

Description:

The following 2 lines check that the current global state is "Veto Signalling" and trigger a state transition if it's possible:

```
    _escrowState.checkSignallingEscrow();
    DUAL_GOVERNANCE.activateNextState();
```

However, in `unlockWstETH` (Escrow.sol#L175-L176), `unlockUnstETH` (Escrow.sol#L203-L204) and `unlockUnstETH` (Escrow.sol#L203-L204): the call to `DUAL_GOVERNANCE.activateNextState()` after passing the `_escrowState.checkSignallingEscrow()` check can start a Rage Quit state and unlock the funds that were supposed to be locked in the `Escrow` according to the Rage Quit's second seal.
Effectively, we'd be in a state of Rage Quit without the required locked fund in it.

Recommendation:

Be it for locking or unlocking, it'd be advisable to first call `DUAL_GOVERNANCE.activateNextState();` before calling `_escrowState.checkSignallingEscrow();`:

```
-     _escrowState.checkSignallingEscrow();
-     DUAL_GOVERNANCE.activateNextState();
+     DUAL_GOVERNANCE.activateNextState();
```

```
+    _escrowState.checkSignallingEscrow();
```

While we can see that the Check-Effect-Interaction pattern is as respected as possible: first calling `DUAL_GOVERNANCE.activateNextState()` would act as a status refresh, much needed before the initial checks.

Additionally, to avoid any mistakes, it'd be great to refactor the repeated code into a modifier.

As an example for a modifier:

```
    modifier onlySignallingEscrow() {
        DUAL_GOVERNANCE.activateNextState();
        _escrowState.checkSignallingEscrow();
        _;
        DUAL_GOVERNANCE.activateNextState();
    }
```

Example usage:

```
File: Escrow.sol
- 144:     function unlockStETH() external returns (uint256
unlockedStETHShares) {
+ 144:     function unlockStETH() external onlySignallingEscrow returns
(uint256 unlockedStETHShares) {
- 145:         _escrowState.checkSignallingEscrow();
- 146:
- 147:         DUAL_GOVERNANCE.activateNextState();
148:         _accounting.checkMinAssetsLockDurationPassed(msg.sender,
_escrowState.minAssetsLockDuration);
149:         unlockedStETHShares =
_accounting.accountStETHSharesUnlock(msg.sender).toUint256();
150:         ST_ETH.transferShares(msg.sender, unlockedStETHShares);
151:
- 152:         DUAL_GOVERNANCE.activateNextState();
153:     }
```

Lido's response: Fixed in PR 95

High Severity Issues

H-01 - `getVetoerState()` doesn't return the right `unstETHLockedShares` value

Severity: **High**	Impact: **Medium**	Likelihood: **High**
Files: Escrow.sol	Status: Fixed	

Description:

```
File: Escrow.sol
    function getVetoerState(address vetoer) external view returns (VetoerState
memory state) {
        HolderAssets storage assets = _accounting.assets[vetoer];

        state.unstETHIdsCount = assets.unstETHIds.length;
        state.stETHLockedShares = assets.stETHLockedShares.toUint256();
-        state.unstETHLockedShares = assets.stETHLockedShares.toUint256();
+        state.unstETHLockedShares = assets.unstETHLockedShares.toUint256();
        state.lastAssetsLockTimestamp =
assets.lastAssetsLockTimestamp.toSeconds();
    }
```

Lido's response: Fixed in PR 108

H-02 - Griefing `_batchesQueue.close()`

Severity: **High**	Impact: **High**	Likelihood: **Medium**
Files: Escrow.sol#L259, Escrow.sol#L264	Status: Fixed	Violated Property: P-25. Batches Queue Close Causes No Changes

Description:

Every time that the `ST_ETH.balanceOf(address(this));` will be less than `WITHDRAWAL_QUEUE.MIN_STETH_WITHDRAWAL_AMOUNT()`, an attacker can donate an amount above said `MIN_STETH_WITHDRAWAL_AMOUNT` (currently equal to 100 wei) via a frontrunning call to `requestNextWithdrawalsBatch()`.
The impact is that `_batchesQueue` will not be closed. Instead, funds will be sent to the withdrawal queue. Due to the use of `balanceOf` instead of internal accounting: this can be repeated.

The impact is critical because this DOS-es the RageQuit state at a minimal cost for the attacker.

A workaround would be to deploy a contract that will call this function twice in the same transaction, which will close the `_batchesQueue` and allow the RageQuit to continue.

Recommendation

Use internal accounting or add a check to close the `_batchesQueue` at the end of the function. This way, if the condition holds after the batch request (and just not before which currently forces the user to call this twice), this can close the `_batchesQueue` in just one call without leaving this attack vector open.

Lido's response: Fixed in commits:

- 44ce534597755dc52c09e43ed2a0e22741fc0c0d

Certora's Fix Review:

The fixes actually introduce a bug.
At Escrow.sol#L269, there should be a return that was removed because it prevents a revert in the case of which the stEth balance is 0 or less than the minimal amount for withdraw (all the locked stEth is in NFTs). In that case, there is a DOS.
The workaround would be to donate more than the minimum withdrawable amount of stEth to the Escrow so that the call can continue.
This DOS has 3 reverts than will happen by order:

1. WithdrawalBatchesQueue.sol#L111
2. WithdrawalBatchesQueue.sol#L115
3. Escrow.sol#L283 (there is a check inside that it is open and it's not)

Lido's response: Fixed in PR #103 and PR #105

H-03 - `cancelAllPendingProposals()` is callable in any state

Severity: **High**	Impact: **Medium**	Likelihood: **High**
Files: DualGovernance.sol#L140-L146	Status: Fixed	

Description:

The specification explicitly mentions that when calling cancelAllPendingProposals the current governance state MUST NOT equal Normal, VetoCooldown, or RageQuit.

However, when we look at the contracts, there are no checks ensuring this:

```
    function cancelAllPendingProposals() external {
        Proposers.Proposer memory proposer =
_proposers.getProposer(msg.sender);
        if (proposer.executor != TIMELOCK.getAdminExecutor()) {
            revert NotAdminProposer();
        }
        TIMELOCK.cancelAllNonExecutedProposals();
    }
```

Lido's Response: Fixed in PR 100

H-04 - `tieBreakerResumeSealable` has no call to `activateNextState` before checking for a tie

Severity: **High**	Impact: **Medium**	Likelihood: **High**
Files: DualGovernance.sol#L280-L284	Status: Fixed	

Description:

As a consequence, the tiebreaker committee could call the function even though the state doesn't point to a tie anymore.

```
    function tiebreakerResumeSealable(address sealable) external {
        _tiebreaker.checkCallerIsTiebreakerCommittee();
        _tiebreaker.checkTie(_stateMachine.getCurrentState(),
_stateMachine.getNormalOrVetoCooldownStateExitedAt());
        RESEAL_MANAGER.resume(sealable);
    }
```

Lido's response: Fixed in PR 113

H-05 - RageQuit is DOS-d if the minimum amount of stEth to withdraw is 0

Severity: **High**	Impact: **High**	Likelihood: **Medium**
Files: Escrow.sol#L265-L266	Status: Fixed	

Description:

If the `WITHDRAWAL_QUEUE.MIN_STETH_WITHDRAWAL_AMOUNT()` is 0: then the `requestNextWithdrawalsBatch()` function can never close the `_batchesQueue`. The consequence is that the RageQuit will never finish.

This will cause a DOS to the RageQuit and will prevent users from RageQuiting properly.

As this is a mutable parameter and not a constant, it is therefore possible for the decision to one day set this minimum amount to 0.

Lido's response: Fixed in PR 121

H-06 - RageQuit can fail to start even after the threshold have passed by frontrun

Severity: **High**	Impact: **High**	Likelihood: **Medium**
Files: Escrow.sol	Status: Fixed	

Description:

The markUnstETHFinalized() function may be able to frontrun a call to start a RageQuit but this may cause it to fail by decreasing the support by updating the value of the locked NFTs. When the conditions for the RageQuit have been met, it shouldn't be reversible. However this function breaks this rule. Anyone can call this function so the scope is broad.

Lido's response: Pending fix in PR #127

Medium Severity Issues

M-01 - `HashConsensus.scheduledAt` is not updated when the user changes their vote from `true` to `false` to `true`

Severity: **Medium**	Impact: **High**	Likelihood: **Low**
Files: HashConsensus.sol#L53-L70	Status: Fixed	

Description:

Note: This can only happen with the ResealCommitee as this is the only place where members can vote true/false.

Assume the following scenario:

- 3 committee members for ResealManager
- 3 of them vote yes, so now the proposal is scheduled. So far so good
- 1 of them changes their vote to no, because he changes his mind cause he doesn't understand the proposal. Now, the proposal cannot be executed cause it doesn't meet the quorum
- Now, he understands it better and wants to vote yes. - He votes again, but the scheduledAt doesn't get updated

```
    function _vote(bytes32 hash, bool support) internal {
        if (_hashStates[hash].usedAt > Timestamps.from(0)) {
            revert HashAlreadyUsed(hash);
        }

        if (approves[msg.sender][hash] == support) {
            return;
        }

        uint256 heads = _getSupport(hash);
```

```
        // heads compares to quorum - 1 because the current vote is not
counted yet
        if (heads >= quorum - 1 && support == true &&
_hashStates[hash].scheduledAt == Timestamps.from(0)) {
            _hashStates[hash].scheduledAt = Timestamps.from(block.timestamp);
        }

        approves[msg.sender][hash] = support;
        emit Voted(msg.sender, hash, support);
    }
```

Lido's response: Fixed in PR 116

M-02 - `uint256 rageQuitRound = Math.min(self.rageQuitRound + 1, type(uint8).max);` will revert when `self.rageQuitRound == type(uint8).max`

Severity: **Medium**	Impact: **High**	Likelihood: **Low**
Files: DualGovernanceStateMachine.sol	Status: Fixed	

Description:

`Math.min` will revert due to an overflow at `self.rageQuitRound == type(uint8).max` as `rageQuitRound` is of type `uint8` and `rageQuitRound + 1` will automatically revert in Solidity versions 0.8+:

```
            IEscrow signallingEscrow = self.signallingEscrow;
            uint256 rageQuitRound = Math.min(self.rageQuitRound + 1,
type(uint8).max);
            self.rageQuitRound = uint8(rageQuitRound);
```

Lido's response: Fixed in PR 107

M-03 - Lack of access control for `TiebreakerSubCommittee.sealableResume`

Severity: **Medium**	Impact: **Low**	Likelihood: **High**
Files: TiebreakerSubCommittee	Status: Fixed	

Description:

There should be a check to _checkCallerIsMember

```
    function sealableResume(address sealable) public {
        (bytes memory proposalData, bytes32 key,) =
_encodeSealableResume(sealable);
        _vote(key, true);
        _pushProposal(key, uint256(ProposalType.ResumeSelable), proposalData);
    }
```

While this enables non-members to vote: their votes won't be taken into account due to how the members list is browsed when counting the votes. Hence the impact is low. Also, while the Proposal list can be polluted, it's unlikely to be DOS-ed thanks to the `offset` and `limit` parameters when fetching the list.

Lido's response: Fixed in PR 93

M-04 - `tieBreakerScheduleProposal` does not trigger a state transition which will lead to wrong state post execution

Severity: **Medium**	Impact: **Low**	Likelihood: **High**
Files: DualGovernance	Status: Fixed	

Description:

`tieBreakerScheduleProposal`'s specification says it should trigger a state transition before checking the preconditions

```
    function tiebreakerScheduleProposal(uint256 proposalId) external {
>>          _tiebreaker.checkCallerIsTiebreakerCommittee();
>>          _tiebreaker.checkTie(_stateMachine.getCurrentState(),
_stateMachine.getNormalOrVetoCooldownStateExitedAt());

        TIMELOCK.schedule(proposalId);
    }
```

However, in the code there are no calls to `_stateMachine.activateNextState(...);`

Lido's reponse: Fixed in PR 97

M-05 - `cancelAllPendingProposals` does not trigger a state transition

Severity: **Medium**	Impact: **Low**	Likelihood: **High**
Files: DualGovernance	Status: Fixed	

Description:

The specification says that the function **Triggers a transition of the current governance state, if one is possible.**

However, in the code there are no calls to `_stateMachine.activateNextState(...);`

```
    function cancelAllPendingProposals() external {
        Proposers.Proposer memory proposer =
_proposers.getProposer(msg.sender);
        if (proposer.executor != TIMELOCK.getAdminExecutor()) {
            revert NotAdminProposer();
        }
        TIMELOCK.cancelAllNonExecutedProposals();
    }
```

Lido's Response: Fixed in PR 100

M-06 - Front loading non-existing proposals

Severity: **Medium**	Impact: **High**	Likelihood: **Low**
Files: EmergencyExecutionCommittee.sol#L37-L42 HashConsensus.sol#L60-L86	Status: Fixed	

Description:

The emergency execution committee can vote on a proposal that has to be executed "emergently"

- EmergencyExecutionCommittee.sol#L37-L42

```
    function voteEmergencyExecute(uint256 proposalId, bool _supports) public {
        _checkCallerIsMember();
        (bytes memory proposalData, bytes32 key) =
_encodeEmergencyExecute(proposalId);
        _vote(key, _supports);
        _pushProposal(key, uint256(ProposalType.EmergencyExecute),
proposalData);
    }
```

When you vote and the quorum is met, the timelock kicks off:

- HashConsensus.sol#L60-L86

```
    function _vote(bytes32 hash, bool support) internal {

        // @note this is updated when executing the scheduled proposal
        // @note as in, you can't vote for a proposal that has been executed
        if (_hashStates[hash].usedAt > 0) {
            revert HashAlreadyUsed(hash);
        }

        // @note this prevents "true" support double counting
```

```
        if (approves[msg.sender][hash] == support) {
            return;
        }

        // @note just iterate through all members and check whether they
support it
        uint256 heads = _getSupport(hash);


        // @note mark quorum as reached if its reached
        // @note quorum can't be reached if its "achieved" by changing the
quorum
        if (heads == quorum - 1 && support == true) {
>>>>>           _hashStates[hash].quorumAt = uint40(block.timestamp);
        }

        approves[msg.sender][hash] = support;
        emit Voted(msg.sender, hash, support);
    }
```

Once a quorum is reached, anyone can make the call to execute the emergency proposal
However, if the committee votes for non-existing proposal ID, it won't be executed (it will revert), but the timelock duration will be under progress
so, once there is a real to be executed, it can be executed without timelock as it would have expired already and the proposal has been approved regardless

Lido's Response: Fixed in PR 117

M-07 - Quorum change with pending votes

Severity: **Medium**	Impact: **High**	Likelihood: **Low**
Files: HashConsensus.sol	Status: Fixed	

Description:

All the committee contracts use the HashConsensus contract and when you vote, and the quorum is just-to-be-reached, then .quorumAt is updated

HashConsensus.sol#L60-L85

```
    function _vote(bytes32 hash, bool support) internal {
        if (_hashStates[hash].usedAt > 0) {
            revert HashAlreadyUsed(hash);
        }

        if (approves[msg.sender][hash] == support) {
            return;
        }

        uint256 heads = _getSupport(hash);

        if (heads == quorum - 1 && support == true) {
>>>>            _hashStates[hash].quorumAt = uint40(block.timestamp);
        }

        approves[msg.sender][hash] = support;
        emit Voted(msg.sender, hash, support);
    }
```

For a proposal to be executed, we call _markUsed in each of the execute functions (in any committee)

HashConsensus.sol#L90-L104

```
    function _markUsed(bytes32 hash) internal {
        if (_hashStates[hash].usedAt > 0) {
            revert HashAlreadyUsed(hash);
        }
        if (_getSupport(hash) < quorum) {
            revert QuorumIsNotReached();
        }
        if (block.timestamp < _hashStates[hash].quorumAt + timelockDuration) {
            revert TimelockNotPassed();
        }

        _hashStates[hash].usedAt = uint40(block.timestamp);

        emit HashUsed(hash);
    }
```

- The first check is whether the proposal has been executed
- The second check is whether the quorum was reached
- And the third is whether the timelock duration for the quorum has expired

Assume the following scenario:

- You have 5 members, 3 of them voted, quorum is 5
- remember, once you voted, you cannot vote again:

```
    function _vote(bytes32 hash, bool support) internal {
        // @note this prevents "true" support double counting
>>>>        if (approves[msg.sender][hash] == support) {
            return;
        }
```

- You call removeMembers, remove 2 members and update the quorum to 3
- Now, because they all voted already, the execution path of _vote can't get to here, because they voted already:

```
        if (heads == quorum - 1 && support == true) {
>>>             _hashStates[hash].quorumAt = uint40(block.timestamp);
        }
```

This is easily mitigatable if the check is `heads >= quorum - 1` instead.

Lido's response: Fixed in PR #93

M-08 - Override proposal status

Severity: **Medium**	Impact: **Low**	Likelihood: **High**
Files: ExecutableProposals.sol	Status: Fixed	Violated Property: P-14. Executed is a terminal state for a proposal

Description:

Cancelling all proposals means that the last currently existing proposal is set to `self.lastCancelledProposalId`

```
    function cancelAll(Context storage self) internal {
        uint64 lastCancelledProposalId = self.proposalsCount;
        self.lastCancelledProposalId = lastCancelledProposalId;
        emit ProposalsCancelledTill(lastCancelledProposalId);
    }
```

Then, for a proposal to be marked as cancelled all we need to consider is that its proposalId is less than the last cancelled proposal id.

```
    function _isProposalMarkedCancelled(
        Context storage self,
        uint256 proposalId,
        ProposalData memory proposalData
    ) private view returns (bool) {
        // @note the || can be removed
        return proposalId <= self.lastCancelledProposalId ||
proposalData.status == Status.Cancelled;
    }
```

However, this means that all previously executed proposals will be marked as cancelled too.

Consider the following scenario, there are proposals with id's = `[1,2,3,4,5]` where for each of them the status = `Executed`.

Now, consider there's a new proposal with `id = 6`. But then it gets cancelled. So now `self.lastCancelledProposalId = 6`.

This means, when calling getProposalInfo(4), it will return `status = Cancelled` even though it has been already executed.

```
    function getProposalInfo(
        Context storage self,
        uint256 proposalId
    ) internal view returns (Status status, address executor, Timestamp
submittedAt, Timestamp scheduledAt) {
        ProposalData memory proposalData = self.proposals[proposalId].data;
        _checkProposalExists(proposalId, proposalData);

>>>         status = _isProposalMarkedCancelled(self, proposalId, proposalData)
? Status.Cancelled : proposalData.status;
        executor = address(proposalData.executor);
        submittedAt = proposalData.submittedAt;
        scheduledAt = proposalData.scheduledAt;
    }
```

Lido's response: Fixed in PR #94

M-09 - Users may not get ProposalTimelock days to veto Proposals

Severity: **Medium**	Impact: **Medium**	Likelihood: **Medium**
Files: DualGovernanceStateMachine.sol	Status: Acknowledged	

Description:

Scenario:

- An attacker submits a proposal at time 0

- Then they lock the first seal amount of stEth in the Escrow

- Then they wait for ((ProposalTimelock - (deactivationMaximalPeriod + CooldownPeriodLength)) +1) to unlock all the stEth

- After the deactivation ends in Time (ProposalTimelock - CooldownPeriodLength - 1) : the cooldown starts for CooldownPeriodLength. Therefore, at the ProposalTimelock mark, the attacker can execute the proposal. However, the users could only veto in the first (ProposalTimelock - CooldownPeriodLength - 1) due to the cooldown preventing them to veto before the execution. This is a general issue that restricts the choice of these parameters.

Lido's response: The final configuration of the Dual Governance parameters will consider the described scenario to ensure users have adequate time to veto submitted proposals. For instance, a possible mitigation could involve setting the ProposalTimelock duration equal to the DeactivationMaximalPeriod duration. In this setup, proposers would have the entire ProposalTimelock timeframe to veto a proposal.

M-10 - State transition to RageQuit isn't persisted when it should be

Severity: **Medium**	Impact: **Low**	Likelihood: **High**
Files: Escrow.sol	Status: Fixed	

Description:

If the RageQuit is supposed to start due to time having passed: then a call to DUAL_GOVERNANCE.activateNextState() is supposed to trigger the transition to the RageQuit state. However, going through any of the lock/unlock functions will revert due not being in the VetoSignalling state. This means that the state transition to RageQuit isn't persisted while an error message with a failed transaction stems from the fact that the current state is RageQuit. The state transition can't happen unless someone specifically calls `activateNextState()` directly on the `DualGovernance`.

This is a degraded flow: we'd expect the final state to be "RageQuit".

Lido's response: While the likelihood of these scenarios is low, they should still be handled on the UI side by clearly informing users of the reason for the transaction failure. Additionally, the specification has been updated to outline the expected sequence of user actions, further minimizing the chances of this issue occurring. The specification updates were introduced in the PR: #127

M-11 - requestWithdrawals() can be called whenRageQuit should have started

Severity: **Medium**	Impact: **Low**	Likelihood: **High**
Files: Escrow.sol	Status: Fixed	

Description:

The requestWithdrawals() function can be called successfully even if the RageQuit should have started (but hasn't started yet). This is unlike the other lock and unlock functions.

Lido's response: This method was not used in the DualGovernance contract and has been removed in PR: #135

Low Severity Issues

L-01 - `removeSealableWithdrawalBlocker` does not return a boolean or revert when failing to remove

Severity: **Low**	Impact: **Low**	Likelihood: **Low**
Files: Tiebreaker.sol	Status: Fixed	

Lido's response: Fixed in PR 112

L-02 - `lastAssetsLockTimestamp` is updated even though `unstEthIds = [ ]`

Severity: **Low**	Impact: **Low**	Likelihood: **Low**
Files: Escrow.sol	Status: Fixed	

Description:

When calling `lockUnstETH` the user has to pass an amount of ID's to be locked.

```
 function lockUnstETH(uint256[] memory unstETHIds) external {
        _escrowState.checkSignallingEscrow();
        DUAL_GOVERNANCE.activateNextState();

        WithdrawalRequestStatus[] memory statuses =
WITHDRAWAL_QUEUE.getWithdrawalStatus(unstETHIds);

        _accounting.accountUnstETHLock(msg.sender, unstETHIds, statuses);
        uint256 unstETHIdsCount = unstETHIds.length;
        for (uint256 i = 0; i < unstETHIdsCount; ++i) {
            WITHDRAWAL_QUEUE.transferFrom(msg.sender, address(this),
unstETHIds[i]);
        }

        DUAL_GOVERNANCE.activateNextState();
    }
```

Then, when these are processed in the accounting library, the lastAssetLockTimestamp is updated to indicate that they just recently had another deposit.

```
    function accountUnstETHLock(
        Context storage self,
        address holder,
```

```
        uint256[] memory unstETHIds,
        WithdrawalRequestStatus[] memory statuses
    ) internal {
        assert(unstETHIds.length == statuses.length);

        SharesValue totalUnstETHLocked;
        uint256 unstETHcount = unstETHIds.length;
        for (uint256 i = 0; i < unstETHcount; ++i) {
            totalUnstETHLocked = totalUnstETHLocked + _addUnstETHRecord(self,
holder, unstETHIds[i], statuses[i]);
        }
        // @audit timestamp gets updated even if there's nothing locked
>>        self.assets[holder].lastAssetsLockTimestamp = Timestamps.now();
```

However, right you you can pass `unstETHIds = [ ]` and the execution flow would still go to `lastAssetsLockTimestamp = Timestamps.now();`

The mitigation is as simple as just revering if `unstEthIds.length == 0`;

Lido's Response: Fixed in PR #103

L-03 - `withdrawETH` is callable with `unstEthIds = [ ]`

Severity: **Low**	Impact: **Low**	Likelihood: **Low**
Files: Escrow.sol	Status: Fixed	

Lido's response: Fixed in PR #103

L-04 - SealableCalls.sol.callResume() `isPaused` flag is wrong

Severity: **Low**	Impact: **Low**	Likelihood: **Low**
Files: SealableCalls.sol	Status: Fixed	

Description:

Given that this will flag the call as unsuccessful, this will prevent resuming the contract's functionality, forever leaving it paused.

```
File: SealableCalls.sol
63:     function callResume(ISealable sealable) internal returns (bool
success, bytes memory lowLevelError) {
64:         try sealable.resume() {
65:             (bool isPausedCallSuccess, bytes memory isPausedLowLevelError,
bool isPaused) = callIsPaused(sealable);
- 66:              success = isPausedCallSuccess && isPaused;
+ 66:              success = isPausedCallSuccess && !isPaused;
```

Lido's response: Fixed in PR 102

Informational Severity Issues

I-01. `Duration.sol:MIN` is never used

Recommendation: Consider deleting it: Duration.sol#L111.

I-02. Lack of CEIP => Bypassing `MAX_SEALABLE_WITHDRAWAL_BLOCKERS_COUNT`

Description:
Note: This is an out of scope centralized risk. But still worth mentioning as it's an open path that could easily be corrected.

If you look at this function: Tiebreaker.sol#L42-L61, there's a strict equality `sealableWithdrawalBlockersCount == maxSealableWithdrawalBlockersCount` before reverting.
There's a way to have `sealableWithdrawalBlockersCount > maxSealableWithdrawalBlockersCount` and make this check "not strong enough" (check should've been `sealableWithdrawalBlockersCount >= maxSealableWithdrawalBlockersCount` for extra safety).
At DualGovernance.sol#L240-L243, the admin needs to input a malicious `address sealableWithdrawalBlocker.` Then through re-entrancy, at this line: Tiebreaker.sol#L52, it could re-enter through the admin contract, pass the check, and reiterate the reentrancy several times. Then the flow would proceed to calling `self.sealableWithdrawalBlockers.add(sealableWithdrawalBlocker)` after all reentrancies, effectively getting `sealableWithdrawalBlockers` above the `maxSealableWithdrawalBlockersCount`.

The likelihood is extremely low and this is an admin protected function, but still, the code would probably be safer by respecting the CEIP like this:

```
File: Tiebreaker.sol
42:     function addSealableWithdrawalBlocker(
43:         Context storage self,
44:         address sealableWithdrawalBlocker,
45:         uint256 maxSealableWithdrawalBlockersCount
46:     ) internal {
```

```
47:         uint256 sealableWithdrawalBlockersCount =
self.sealableWithdrawalBlockers.length();
- 48:         if (sealableWithdrawalBlockersCount ==
maxSealableWithdrawalBlockersCount) {
+ 48:         if (sealableWithdrawalBlockersCount >=
maxSealableWithdrawalBlockersCount) { // <------ Stronger check
49:             revert SealableWithdrawalBlockersLimitReached();
50:         }
51:
+ 57:         bool isSuccessfullyAdded =
self.sealableWithdrawalBlockers.add(sealableWithdrawalBlocker); // <----
Respecting CEIP
+ 58:         if (isSuccessfullyAdded) {
+ 59:             emit
SealableWithdrawalBlockerAdded(sealableWithdrawalBlocker);
+ 60:         }
52:         (bool isCallSucceed, /* lowLevelError */, /* isPaused */ ) =
ISealable(sealableWithdrawalBlocker).callIsPaused();
53:         if (!isCallSucceed) {
54:             revert InvalidSealable(sealableWithdrawalBlocker);
55:         }
56:
- 57:         bool isSuccessfullyAdded =
self.sealableWithdrawalBlockers.add(sealableWithdrawalBlocker);
- 58:         if (isSuccessfullyAdded) {
- 59:             emit
SealableWithdrawalBlockerAdded(sealableWithdrawalBlocker);
- 60:         }
61:     }
```

I-03. Variable renaming

Description:

This input variable's name is "support", not "supports", in other inheritors of HashConsensus

```
File: EmergencyExecutionCommittee.sol
- 38:     /// @param _supports Indicates whether the member supports the
proposal execution
- 39:     function voteEmergencyExecute(uint256 proposalId, bool _supports)
public { //@audit-issue issue var name is "support", not "supports", in other
```

```
inheritors of HashConsensus
+ 38:     /// @param _support Indicates whether the member supports the
proposal execution
+ 39:     function voteEmergencyExecute(uint256 proposalId, bool _support)
public {
```

See:

```
contracts/committees/HashConsensus.sol:
  17:     event Voted(address indexed signer, bytes32 hash, bool support);
  49:     function _vote(bytes32 hash, bool support) internal {

contracts/committees/ResealCommittee.sol:
  34:     function voteReseal(address sealable, bool support) public {
```

I-04. Duplicate import statements

Description:

```
## File: contracts/DualGovernance.sol

DualGovernance.sol:7: import {IResealManager} from
"./interfaces/IResealManager.sol";

DualGovernance.sol:13: import {IResealManager} from
"./interfaces/IResealManager.sol";
```

I-05. Unused `error` definitions

Description:

- contracts/libraries/WithdrawalBatchesQueue.sol

```
## File: contracts/libraries/WithdrawalBatchesQueue.sol

WithdrawalBatchesQueue.sol:26:     error NotAllBatchesClaimed(uint256 total,
uint256 claimed);
```

```
WithdrawalBatchesQueue.sol:27:     error
InvalidWithdrawalsBatchesQueueState(State actual);
```

I-06. Event is never emitted

Description:
The following are defined but never emitted. They can either be removed or added where they're missing.

Affected code:

- contracts/Escrow.sol

```
## File: contracts/Escrow.sol

Escrow.sol:67:     event ConfigProviderSet(address newConfigProvider);
```

- contracts/libraries/Proposers.sol

```
## File: contracts/libraries/Proposers.sol

Proposers.sol:22:     event AdminExecutorSet(address indexed adminExecutor);
```

I-07. `setResealCommittee` should emit an event

Description:

```
File: DualGovernance.sol
302:     function setResealCommittee(address resealCommittee) external {
303:         _checkCallerIsAdminExecutor();
304:         _resealCommittee = resealCommittee;
305:     }
```

I-08. TODO Left in the code

Description:
Affected code:

- contracts/libraries/DualGovernanceConfig.sol

```
## File: contracts/libraries/DualGovernanceConfig.sol

DualGovernanceConfig.sol:114:             ); // TODO: rewrite in a prettier
way
```

I-09. `secondSealRageQuitSupport == firstSealRageQuitSupport` is theoretically possible, but shouldn't be

Lido's Response: Such a configuration is not considered valid and should never be used. To prevent this, additional sanity checks may be implemented in the `DualGovernanceConfigProvider` constructor or in the `DualGovernance.setConfigProvider()` method.

I-10. Formula simplification for power of 2

Description:
The expression here can be simplified to rageQuitRound ** 2

```
    function calcRageQuitWithdrawalsTimelock(
…
        return self.rageQuitEthWithdrawalsMinTimelock
            + Durations.from(
                (
-                   self.rageQuitEthWithdrawalsTimelockGrowthCoeffs[0] *
rageQuitRound * rageQuitRound
+                   self.rageQuitEthWithdrawalsTimelockGrowthCoeffs[0] *
rageQuitRound ** 2
…
```

certora LIDO

Formal Verification

Verification Notations

Formally Verified	The rule is verified for every state of the contract(s), under the assumptions of the scope/requirements in the rule.
Formally Verified After Fix	The rule was violated due to an issue in the code and was successfully verified after fixing the issue
Violated	A counter-example exists that violates one of the assertions of the rule.

General Assumptions and Simplifications

For each of the contracts under verification, we rely on "mock" contracts that give artificial and simplified implementations of a few related contracts we do not have implementations of. We designed these to avoid any simplifications that overly limit the scope of verification. These are as follows:

- IStETH – we model this `DummyStETH.sol` as having a fixed exchange rate of ETH * 5 / 3 = shares amount
- ERC20s – DummyERC20A / DummyERC20B implement relatively standard ERC20 contracts that are identical but allow the prover to choose different addresses for various ERC20 contracts
- DummyWstEth – implements a relatively standard ERC20 extended with wrap/unwrap functions
- IWithdrawalQueue – we implemented a simplified version of the real withdrawal queue that was designed to adequately capture the behavior of the real withdrawal queue

Formal Verification Properties

DualGovernance

Module General Assumptions

- We model the following functions as returning an arbitrary value on each invocation and assume they have no side-effects on the DualGovernance contract: Address.sendValue, Address.functionCallWithValue, ISealable.getResumeSinceTimestamp, IOwnable.transferOwnership, Executor.execute
- We assume the rage quit first seal threshold is greater than zero and the ragequit second seal is greater than the first seal

Module Properties

P-01. Proposer indexes match their index in the array and are always < the array length	
Status: Violated	*Assumption: we assume the proposer array is less than 5 to allow us to bound the iterations of loops.*

Rule Name	Status	Description	Link to rule report
w2_1a_indexes_match	Formally Verified after Fix	*for any registered proposer, his index should be ≤ the length of the array of proposers" and "for each entry in the struct in the array, show that the index inside is the same as the real array index* ***NOTE:*** *This originally caught a bug during which there was a counterexample. It now passes after Lido acknowledged and fixed the bug. Report with counterxample before bug fix.*	*Report*

Note: we ran this rule against both the code before the fix attempt was implemented and after the fix attempt was implemented (fix attempt commit link)

P-02. Dual Governance Key Property 1

Status: Verified

Rule Name	Status	Description	Link to rule report
dg_kp_1_proposal_execution	Verified	Proposals cannot be executed in the Veto Signaling (both parent state and Deactivation sub-state) and Rage Quit states.	*Report*

Note: this property is meant to verify a rule from Lido's Key Properties documentation

P-03. Dual Governance Key Property 2

Status: Verified

Rule Name	Status	Description	Link to rule report
dg_kp_2_proposal_submission	Verified	*Proposals cannot be submitted in the Veto Signaling Deactivation sub-state or in the Veto Cooldown state.*	*Report*

P-04. Dual Governance Key Property 3

Status: Verified

Rule Name	Status	Description	Link to rule report
dg_kp_3_cooldown_execution	Verified	*If a proposal was submitted after the last time the Veto Signaling state was activated, then it cannot be executed in the Veto Cooldown state.*	*Report*

P-05. Dual Governance Key Property 4

Status: Verified

Rule Name	Status	Description	Link to rule report
dg_kp_4_single_ragequit	Verified	*One rage quit cannot start until the previous rage quit has been finalized. In other words, there can only be at most one active rage quit escrow at a time.*	*Report*

P-06. Dual Governance Key Property 4 Addendum

Status: Verified	Note: this only checks the state of the Veto Signaling Escrow after functions have completed and it does not check temporary changes part-way through function execution.

Rule Name	Status	Description	Link to rule report
dg_kp_4_single_ragequit_addendum	Verified	*The vetoSignalling Escrow is never in the RageQuit state.*	*Report*

P-07. Protocol Key Property 1

Status: Verified	

Rule Name	Status	Description	Link to rule report
pp_kp_1_ragequit_extends	Verified	*Regardless of the state in which a proposal is submitted, if the stakers are able to amass and maintain a certain amount of rage quit support before the ProposalExecutionMinTimelock expires, they can extend the timelock for a proportional time, according to the dynamic timelock calculation*	*Report*

P-08. Protocol Key Property 2

Status: Verified

Rule Name	Status	Description	Link to rule report
pp_kp_2_rageq uit_trigger	Verified	*PP-2: It's not possible to prevent a proposal from being executed indefinitely without triggering a rage quit.*	*Report*

P-09. Protocol Key Property 3

Status: Verified

Rule Name	Status	Description	Link to rule report
pp_kp_3_no_in definite_propos al_submission_ block	Verified	*PP-3: It's not possible to block proposal submission indefinitely.*	*Report*

P-10. Protocol Key Property 4

Status: Verified

Rule Name	Status	Description	Link to rule report
pp_kp_4_veto_signalling_deactivation_cancellable	Verified	*PP-4: Until the Veto Signaling Deactivation sub-state transitions to Veto Cooldown, there is always a possibility (given enough rage quit support) of canceling Deactivation and returning to the parent state (possibly triggering a rage quit immediately afterwards).*	*Report*

P-11. Proposal Submission States

Status: Verified

Rule Name	Status	Description	Link to rule report
dg_states_1_proposal_submission_states	Verified	*If proposal submission succeeds, the system was in one of these states: Normal, Veto Signalling, Rage Quit*	*Report*

P-12. Proposal Scheduling States

Status: Verified

Rule Name	Status	Description	Link to rule report
dg_states_2_proposal_scheduling_states	Verified	*If proposal scheduling succeeds, the system was in one of these states: Normal, Veto Cooldown*	*Report*

P-13. Only legal transitions are possible

Status: Verified

Rule Name	Status	Description	Link to rule report
dg_transitions_1_only_legal_transitions	Verified	*If proposal scheduling succeeds, the system was in one of these states: Normal, Veto Cooldown*	*Report*

Emergency Protected Timelock

Module General Assumptions

- We assume that the calls executed through proposals do not have side-effects on the EmergencyProtectedTimelock and we model these as returning empty bytes

Module Properties

P-14. Executed is a terminal state for a proposal

Status: Verified

Rule Name	Status	Description	Link to rule report
W1_4_TerminalityOfExecuted	Verified	*Executed is a terminal state for a proposal, once executed it cannot transition to any other state* *NOTE: this was initially violated before a fix from Lido. Violated report prior to fix.* *Link to PR with fix*	*Report*

P-15. Nonzero Proposals are within bounds

Status: Verified

Rule Name	Status	Description	Link to rule report
outOfBoundsProposalDoesNotExist	Verified	*Proposals with nonzero Ids must either have an ID in the range (0,proposalsCount] or have the NotExist status*	*Report*

P-16. Emergency Protected Timelock Key Property 1

Status: Verified

Rule Name	Status	Description	Link to rule report
EPT_KP_1_SubmissionToSchedulingDelay	Verified	*A proposal cannot be scheduled for execution before at least ProposalExecutionMinTimelock has passed since its submission.*	*Report*

P-17. Emergency Protected Timelock Key Property 2

Status: Verified

Rule Name	Status	Description	Link to rule report
EPT_KP_2_SchedulingToExecutionDelay	Verified	*A proposal cannot be executed until the emergency protection timelock has passed since it was scheduled.*	*Report*

P-18. Emergency Protection Configuration Guarded

Status: Verified

Rule Name	Status	Description	Link to rule report
EPT_1_EmergencyProtectionConfigurationGuarded	Verified	*Emergency protection configuration changes are guarded by committees or admin executors. We check here that the part of the state that should only be alterable by the respective emergency committees or through an admin proposal is indeed not changed on any method call other than ones correctly authorized .*	*Report*

P-19. Only Governance Can Schedule

Status: Verified

Rule Name	Status	Description	Link to rule report
EPT_2a_SchedulingGovernanceOnly	Verified	*Only governance can schedule proposals.*	*Report*

P-20. Only Governance Can Submit Proposals

Status: Verified

Rule Name	Status	Description	Link to rule report

EPT_2b_SubmissionGovernanceOnly	Verified	*Only governance can submit proposals.*	*Report*

P-21. Emergency Mode Restriction

Status: Verified

Rule Name	Status	Description	Link to rule report
EPT_3_EmergencyModeExecutionRestriction	Verified	*If emergency mode is active, only emergency execution committee can execute proposals*	*Report*

P-22. Emergency Mode Liveness

Status: Verified

Rule Name	Status	Description	Link to rule report
EPT_9_EmergencyModeLiveness	Verified	*When emergency mode is active, the emergency execution committee can execute proposals successfully*	*Report*

P-23 .ProposalTimestampConsistency

Status: Verified

Rule Name	Status	Description	Link to rule report
EPT_10_ProposalTimestampConsistency	Verified	*Proposal timestamps reflect timelock actions*	*Report*

P-24. Terminality of Canceled

Status: Verified

Rule Name	Status	Description	Link to rule report
EPT_11_TerminalityOfCancelled	Verified	*Canceled is a terminal state for a proposal, once canceled it cannot transition to any other state*	*Report*

Escrow

Module General Assumptions

- We assume the following function calls have no side effects on the Escrow contract and model these as returning arbitrary numbers (with no side effects):
  - ResealManager: resume, reseal
  - Timelock: submit, schedule, execute, cancelAllNonExecutedProposals, canSchedule, canExecute, getProposalSubmissionTime

Module Properties

P-25. Batches Queue Close Front Running Resistance			
Status: Verified			
Rule Name	Status	Description	Link to rule report
W2_2_front_running	Verified	*In a situation where requestNextWithdrawalsBatch should close the queue, there is no way to prevent it from being closed by first calling another function. NOTE: This rule previously resulted in a counter-example when it was run against a bug in the Lido code: Counterexample Report*	*Report*

P-26. Batches Queue Close Final State

Status: Verified

Rule Name	Status	Description	Link to rule report
W2_2_batchesQueueCloseFinalState	Verified	*once requestNextWithdrawalsBatch results in batchesQueue.close() all additional calls result in close();*	*Report*

P-27. Escrow Key Property 1

Status: Verified

Rule Name	Status	Description	Link to rule report
E_KP_1_rageQuitSupportValue	Verified	*ignoring imprecisions due to fixed-point arithmetic, the rage quit support of an escrow is equal to the formula from the Lido Key Properties document*	*Report*

P-28. Escrow Key Property 3

Status: Verified

Rule Name	Status	Description	Link to rule report

E_KP_3_rageQuitNolockUnlock	Verified	*It's not possible to lock funds in or unlock funds from an escrow that is already in the rage quit state. locking/unlocking implies changing the stETHLockedShares or unstETHLockedShares of an account*	*Report*

P-29. Escrow Key Property 4

Status: Verified

Rule Name	Status	Description	Link to rule report
E_KP_4_unlockMinTime	Verified	*An agent cannot unlock their funds until SignallingEscrowMinLockTime has passed since this user last locked funds.*	*Report*

P-30. Escrow Key Property 5

Status: Verified

Rule Name	Status	Description	Link to rule report
E_KP_5_rageQuitStarter	Verified	*only dual governance can start a rage quit*	*Report*

P-31. Escrow Rage Quit State Final

Status: Verified

Rule Name	Status	Description	Link to rule report
E_State_1_rageQuitFinalState	Verified	*If the state of an escrow is RageQuitEscrow, we can execute any method and it will still be in the same state afterwards*	*Report*

P-32. Valid State Rules

Status: Verified

Rule Name	Status	Description	Link to rule report
validState_batchQueuesSum	Verified	*For the various data structures of the escrow, the structures stay within a safe subset of the statespace. For example, batch queue entry is monotonically increasing.*	*Report*
validState_batchesQueue_claimed_vs_actual_1	Verified		
validState_batchesQueue_distinct_unstETHRecords	Verified		
validState_batchesQueue_monotonicity	Verified		
validState_batchesQueue_ordering	Verified		

validState_batchesQueue_withdrawalQueue	Verified		
validState_claimedUnstEth	Verified		
validState_nonInitialized	Verified		
validState_partialSumMonotonicity_1	Verified		
validState_partialSumMonotonicity_2	Verified		
validState_partialSumOfClaimedUnstETH	Verified		
validState_ragequit	Verified		
validState_signalling validState_totalETHIds	Verified		
validState_totalLockedShares	Verified		
validState_withdrawalQueue	Verified		
validState_withdrawnEth	Verified		
valid_batchIndex	Verified		

P-33. Escrow Key Property 2: Solvency

Status: Verified

Rule Name	Status	Description	Link to rule report
solvency_ETH	Verified	*The amount of each token accounted for in the ragequit support calculation must be less than or equal to the balance of the escrow in the token.*	*Report*
solvency_ETH_before_ragequit	Verified	*Before rage quit eth value of escrow can not be reduced*	*Report*
solvency_stETH_before_ragequit	Verified	*Total holding of stEth before rageQuit start is at least the value of lockedShared*	*Report*
solvency_zeroWstEthBalance	Verified	*Total holding of wst_eth is zero as all wst_eth are converted to st_eth*	*Report*
solvency_batchesQueue_solvent_leftToClaim	Verified	*Those request id left to claim are indeed not claimed*	*Report*

solvency_batchesQueue_allClaimed	Verified	*When all nft are claimed (according to internal accounting), the last one has been claimed*	*Report*

Disclaimer

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.

About Certora

Certora is a Web3 security company that provides industry-leading formal verification tools and smart contract audits. Certora's flagship security product, Certora Prover, is a unique SaaS product that automatically locates even the most rare & hard-to-find bugs on your smart contracts or mathematically proves their absence. The Certora Prover plugs into your standard deployment pipeline. It is helpful for smart contract developers and security researchers during auditing and bug bounties.

Certora also provides services such as auditing, formal verification projects, and incident response.