STATE MIND

# Lido Dual Governance

12-09-2024 - 25-10-2024

# Table of contents

# 1. Project brief

| Title | Description |
|---|---|
| Client | Lido |
| Project name | Lido Dual Governance |
| Timeline | 12-09-2024 - 25-10-2024 |

## Project Log

| Date | Commit Hash | Note |
|---|---|---|
| 20-09-2024 | 8296824213195dd5421222602cbeb3f5a25017b2 | Initial commit |
| 05-02-2025 | 3e0f1ae5740ef8410e928f6cc106e3a5f45a5a75 | Reaudit |

## Short Overview

Lido Dual Governance is a new architecture for managing the entire protocol, now allowing users to express their disagreement with the DAO. In traditional DAO structures, token holders make decisions that determine the protocol's development and updates, while regular users have no way to influence or halt an on-chain proposal if they disagree.

The new governance update in Lido enables users to block a proposal, and if they fail to reach an agreement with token holders, they can exit the system. The traditional DAO and other proposal mechanisms remain the same, but now their proposals are not executed directly—they must first pass through Dual Governance system of contracts.

During a certain period, users can express their disagreement by sending stETH, wstETH, or ERC-721 from the Withdrawal Queue into a special contract called Escrow. Thus, Escrow can also be seen as an oracle reflecting the level of user disagreement with DAO decisions.

Depending on the percentage of funds deposited into Escrow and the time elapsed, Dual Governance changes its state. In a positive scenario, users and token holders reach an agreement, either canceling the proposals or withdrawing funds from Escrow, returning Dual Governance to its original state. In a negative scenario, where no agreement is reached, a RageQuit state is activated that grants users the ability and time to withdraw their ETH. During this period, no proposals are executed.

# Project Scope

The audit covered the following files:

- Escrow.sol
- AssetsAccounting.sol
- DualGovernance.sol
- EmergencyProtectedTimelock.sol
- HashConsensus.sol
- ExecutableProposals.sol
- DualGovernanceStateMachine.sol
- Tiebreaker.sol
- EmergencyProtection.sol
- DualGovernanceConfig.sol
- EscrowState.sol
- Proposers.sol
- TiebreakerCoreCommittee.sol
- TiebreakerSubCommittee.sol
- Duration.sol
- TimelockState.sol
- EnumerableProposals.sol
- PercentD16.sol
- ETHValue.sol
- ImmutableDualGovernanceConfigProvider.sol
- Timestamp.sol
- TimelockedGovernance.sol
- SharesValue.sol
- ResealManager.sol
- IndexOneBased.sol
- ProposalsList.sol
- Executor.sol
- SealableCalls.sol
- ExternalCalls.sol
- Resealer.sol
- DualGovernanceStateTransitions.sol
- WithdrawalsBatchesQueue.sol

# 2. Finding severity breakdown

All vulnerabilities discovered during the audit are classified based on their potential severity and have the following classification:

| Severity | Description |
| --- | --- |
| Critical | Bugs leading to assets theft, fund access locking, or any other loss of funds to be transferred to any party. |
| High | Bugs that can trigger a contract failure. Further recovery is possible only by manual modification of the contract state or replacement. |
| Medium | Bugs that can break the intended contract logic or expose it to DoS attacks, but do not cause direct loss of funds. |
| Informational | Bugs that do not have a significant immediate impact and could be easily fixed. |

Based on the feedback received from the Client regarding the list of findings discovered by the Contractor, they are assigned the following statuses:

| Status | Description |
| --- | --- |
| Fixed | Recommended fixes have been made to the project code and no longer affect its security. |
| Acknowledged | The Client is aware of the finding. Recommendations for the finding are planned to be resolved in the future. |

# 3. Summary of findings

| Severity | # of Findings |
| --- | --- |
| Critical | 0 (0 fixed, 0 acknowledged) |
| High | 0 (0 fixed, 0 acknowledged) |
| Medium | 4 (3 fixed, 1 acknowledged) |
| Informational | 42 (29 fixed, 13 acknowledged) |
| Total | 46 (32 fixed, 14 acknowledged) |

# 4. Conclusion

During the audit of the codebase, 46 issues were found in total:

- 4 medium severity issues (3 fixed, 1 acknowledged)
- 42 informational severity issues (29 fixed, 13 acknowledged)

The final reviewed commit is 3e0f1ae5740ef8410e928f6cc106e3a5f45a5a75

# 5. Findings report

| MEDIUM-01 | Infinity approval to **Withdrawal Queue** may present problems for Escrow and DG system | Acknowledged |
|---|---|---|

### Description

Line: Escrow.sol#L130

Infinite approval is given from the Escrow to Withdrawal Queue to request withdrawals for locked **stETH** when the rage quit state is activated. It is given at the moment of the initialization, not at the moment when requests are performed. This may pose problems and may allow DAO to violate the DG rules. Consider a scenario:

1. Withdrawal Queue has infinite approval to transfer funds from Escrow
2. Malicious DAO makes a proposal to upgrade WQ implementation to be able to arbitrarily transfer these funds.
3. To gain more support from **stETH** holders DAO may propose to burn shares from escrow to increase the value of other holders
4. This may lead to a situation when some of the holders will be afraid to lock funds in Escrow due to the risk that upgraded WQ may transfer their funds. Other holders won't be against DAO because the value of their shares will be increased if the proposal passes.

### Recommendation

We recommend removing infinite approval at the initialization step and only giving approvals to WQ for a certain amount when withdrawals are requested and zeroing it in the same call.

### Client's comments

> The described potential issue remains even if infinite approval is not granted. For instance, a malicious DAO could propose an upgrade that burns all stETH locked in the **Escrow** contract. The risk arises because the upgradability of core contracts creates opportunities for a malicious DAO to manipulate stETH balances. The system is designed to protect against such proposals by enabling users to utilize the Veto Signalling Escrow to initiate a Rage Quit before any malicious proposal is executed.

| MEDIUM-02 | Possible DG state change due to **unstETH** finalization | Fixed at: 1ffa251 |
|---|---|---|

### Description

Line: Escrow.sol#L221

In function **Escrow.markUnstETHFinalized()** value recalculation of locked **unstETH** happens when they are finalized in Withdrawal Queue. After these actions rage support is also changed, which may trigger a change of Dual Governance state.

### Recommendation

We recommend adding an external call – **DUAL_GOVERNANCE.activateNextState()** – before and after main operations in the **Escrow.markUnstETHFinalized()** function.

| MEDIUM-03 | Incorrect configuration of **MAX_EMERGENCY_PROTECTION_DURATION** | Fixed at: 071f033 |
|---|---|---|

### Description

Line: EmergencyProtectedTimelock.sol#L63.
The immutable variable **MAX_EMERGENCY_PROTECTION_DURATION** is incorrectly set to **sanityCheckParams.maxEmergencyModeDuration**, when it should be set to **sanityCheckParams.maxEmergencyProtectionDuration**.
If the values differ by a lot, **MAX_EMERGENCY_PROTECTION_DURATION** can be bigger or smaller than expected.

### Recommendation

We recommend to set **MAX_EMERGENCY_PROTECTION_DURATION = sanityCheckParams.maxEmergencyProtectionDuration**.

| MEDIUM-04 | Incorrect deployment can break **TiebreakerCoreCommittee** | Fixed at: 40ca9e6 |
|---|---|---|

### Description

Lines: TiebreakerCoreCommittee.sol#L33-L35
The **TiebreakerCoreCommittee**'s constructor does not call the internal **HashConsensus._addMembers()**. If contract deployment and a public function **HashConsensus.addMembers()** call happen in different transactions, then the **TiebreakerCoreCommittee**'s logic can be broken.
After the deployment, the **HashConsensus.quorum** variable will be zero, which allows scheduling any possible **hash** via the **HashConsensus.schedule()** function bypassing a voting process for any future proposal.

```
...
if (_getSupport(hash) < quorum) { // 0 < 0 -> false
      revert QuorumIsNotReached();
}
...
```

To execute these proposals **DualGovernance** has to meet **Tiebreaker.isTie()** requirements, thus this issue does not pose a threat to the governance system unless the **TiebreakerCoreCommittee** is not deployed during the deadlock state.

### Recommendation

We recommend adding the **HashConsensus._addMembers()** call to the constructor. As an additional precaution, the **quorum** variable can be set to **1** to block any possibility of unexpected execution of the **HashConsensus.schedule()** function.

| INFORMATIONAL-01 | Inconsistent result of the DualGovernance state view functions | Fixed at: 1ffa251 |
|---|---|---|

### Description

The return values of the **DualGovernance.canSubmitProposal()**, **DualGovernance.canScheduleProposal()**, **DualGovernance.getState()**, and **DualGovernance.getStateDetails()** will not reflect the actual state of the **DualGovernance** as they do not invoke the state transition function **DualGovernance.activateNextState()**.
Additionally, the state details are queried in the functions **DualGovernance.getTiebreakerDetails()** and **DualGovernance.resealSealable()** without considering possible state transitions.

### Recommendation

We recommend using the **DualGovernanceStateTransitions.getStateTransition()** to obtain the up-to-date governance state or always bundle these view function calls with the **DualGovernance.activateNextState()** invocation.

| INFORMATIONAL-02 | unregisterProposer gas optimization | Fixed at: 2d44940 |
|---|---|---|

### Description

Line: DualGovernance.sol#L232

The **DualGovernance.unregisterProposer()** function checks if a caller is the **TIMELOCK.getAdminExecutor()**, hence **TIMELOCK.getAdminExecutor()** is **msg.sender**.

Later it checks that **TIMELOCK.getAdminExecutor()** still belongs to the **_proposers**.

```
if (!_proposers.isExecutor(TIMELOCK.getAdminExecutor())) {
    revert UnownedAdminExecutor();
}
```

### Recommendation

We recommend replacing **TIMELOCK.getAdminExecutor()** with **msg.sender**.

```
- if (!_proposers.isExecutor(TIMELOCK.getAdminExecutor())) {
+ if (!_proposers.isExecutor(msg.sender)) {
    revert UnownedAdminExecutor();
}
```

| INFORMATIONAL-03 | Redundant governance config validity check | Acknowledged |
|---|---|---|

### Description

Lines:

- ImmutableDualGovernanceConfigProvider.sol#L32
- DualGovernance.sol#L329

The **DualGovernanceConfig** is validated twice:

1. During the **ImmutableDualGovernanceConfigProvider** deployment.
2. During the **DualGovernance** deployment or the **DualGovernance.setConfigProvider()** execution.

If **ImmutableDualGovernanceConfigProvider** is the only implementation planned to be used as a config provider, then one of the checks can be lifted.

### Recommendation

We recommend retaining validation in the **DualGovernance._setConfigProvider()** while removing it from the **ImmutableDualGovernanceConfigProvider**, reducing config deployment cost.

### Client's comments

The validation is preserved to ensure that proposals for setting a new Dual Governance config provider do not fail due to invalid values in the **ImmutableDualGovernanceConfigProvider**. This validation guarantees that a successfully deployed **ImmutableDualGovernanceConfig** can always be set as the config provider.

| INFORMATIONAL-04 | Missing constructor check for TiebreakerActivationTimeout | Fixed at: 70542a5 |
|---|---|---|

### Description

Lines: DualGovernance.sol#L101-L102
The contract's constructor has no validation check to ensure that **minTiebreakerActivationTimeout** is less than **maxTiebreakerActivationTimeout**. These values initialize immutable variables and cannot be changed later.

### Recommendation

We recommend introducing a sanity check in the constructor to enforce that **minTiebreakerActivationTimeout** is always less than **maxTiebreakerActivationTimeout**.

### Client's comments

Fixed in 70542a52b92167ca8123aed08a1ce64bed39b734 and 602235ff3374cb4521248de7abcfa39b4395dcf3

| INFORMATIONAL-05 | Re-writing address of the Rage Escrow | Acknowledged |
|---|---|---|

### Description

Line: DualGovernanceStateMachine.sol#L125
Each time the State-Machine enters the **RageQuit** state, it re-writes the **Context.rageQuitEscrow**'s value, which may lead to losing the address of the previous Escrow. Since then, there'll be no chance to obtain the address through the **DualGovernance** contract interface, thus it's becoming hard to withdraw ETH.
**DualGovernanceStateMachine.activateNextState**:

```
} else if (newState == State.RageQuit) {
    IEscrow signallingEscrow = self.signallingEscrow;

    uint256 currentRageQuitRound = self.rageQuitRound;

    /// @dev Limits the maximum value of the rage quit round to prevent failures due to arithmetic overflow
    /// if the number of consecutive rage quits reaches MAX_RAGE_QUIT_ROUND.
    uint256 newRageQuitRound = Math.min(currentRageQuitRound + 1, MAX_RAGE_QUIT_ROUND);
    self.rageQuitRound = uint8(newRageQuitRound);

    signallingEscrow.startRageQuit(
        config.rageQuitExtensionPeriodDuration, config.calcRageQuitWithdrawalsDelay(newRageQuitRound)
    );
    self.rageQuitEscrow = signallingEscrow; // << here
    _deployNewSignallingEscrow(self, escrowMasterCopy, config.minAssetsLockDuration);
}
```

### Recommendation

We recommend adding a container from which all previous **RageQuitEscrow** addresses can be obtained.

### Client's comments

The addresses of deployed Rage Quit Escrow contracts can be determined using emitted events, off-chain tools, or by reviewing the transaction history of the vetoer's interactions with the contract. To simplify the process for vetoers, a separate UI will be implemented for ETH withdrawals.

| INFORMATIONAL-06 | Ensure that updating DualGovernance.configProvider doesn't affect the current state of DualGovernanceStateMachine | Acknowledged |
|---|---|---|

### Description

Line: DualGovernance.sol#L187

When updating the **DualGovernance.configProvider** via the **DualGovernance.setConfigProvider()** function, the state machine **DualGovernanceStateMachine** transitions might result in unintended changes due to differences between the old and new configurations. This could lead to an inconsistency between the state before and after the configuration update.

### Recommendation

We recommend implementing a state preservation mechanism within the **DualGovernance.setConfigProvider()** function. This can be achieved by comparing the state before and after the configuration update:

```
function setConfigProvider(IDualGovernanceConfigProvider newConfigProvider) external {
  _checkCallerIsAdminExecutor();

  _stateMachine.activateNextState(_configProvider.getDualGovernanceConfig(), ESCROW_MASTER_COPY);
  State stateBeforeUpdate = _stateMachine.getState();

  _setConfigProvider(newConfigProvider);

  _stateMachine.signallingEscrow.setMinAssetsLockDuration(
    newConfigProvider.getDualGovernanceConfig().minAssetsLockDuration
  );

  _stateMachine.activateNextState(_configProvider.getDualGovernanceConfig(), ESCROW_MASTER_COPY);
  State stateAfterUpdate = _stateMachine.getState();

  if (stateBeforeUpdate != stateAfterUpdate) {
   revert StatePreservationViolated();
  }
}
```

The double call to **DualGovernanceStateMachine.activateNextState()** before and after the configuration update ensures that any pending state transitions are resolved before applying the new configuration.

### Client's comments

Inconsistency between the state before and after a configuration update is allowed and, in some scenarios, may be inevitable. For example, if the DAO decides to lower the first seal threshold, it could trigger the activation of the Veto Signalling state immediately after the update. With the proposed changes, such proposals would remain unexecutable until the Rage Quit support drops below the required threshold.

| INFORMATIONAL-07 | Gas optimisations | Fixed at: 8e0aaa0 |
|---|---|---|

### Description

Lines:

- DualGovernanceStateMachine.sol#L86
- SealableCalls.sol#L50
- AssetsAccounting.sol#L198
- EscrowState.sol#L132
- ExternalCalls.sol#L20
- HashConsensus.sol#L73-L84

In function **DualGovernanceStateMachine.activateNextState()** value of **Timestamps.now()** is used several times. It is better to call it once to avoid wrapping and unwrapping actions every time.
On SealableCalls.sol#L50 **false** value is assigned to variable **success**, but by default, it is already **false**.
We can save **self.assets[holder]** to some variable so as not to waste gas when accessing mapping.
**self.rageQuitExtensionPeriodStartedAt** can be saved in a local variable.
The **ExternalCalls** library makes multiple calls to the **Executor** contract instead of passing all data once.
The **HashState** struct occupies one slot. The **HashState._markUsed()** reads the **_hashStates** mapping three times, instead of one.

### Recommendation

We recommend avoiding multiple calls to get the same value, avoiding assigning the same value twice, reading the storage once, and optimizing these parts of the code.

### Client's comments

Fixed in 8e0aaa0b14ca6732dae74a3fa6de83e6336be630. **SealableCalls** was updated in 00e2514deb5da087063aad771adb1ca330e9741f.

| INFORMATIONAL-08 | Absence of zero check | Fixed at: f6f5433 |
|---|---|---|

### Description

Line: Escrow.sol#L207
Call of this function with empty calldata will result in emitting event and activating next state.

### Recommendation

We recommend setting a zero check as like in Escrow.sol#L189.

| INFORMATIONAL-09 | Event emission for unfinalized records | Fixed at: 0ce5711 |
|---|---|---|

### Description

Lines:

- Escrow.sol#L225
- libraries/AssetsAccounting.sol#L374

In the **AssetsAccounting.accountUnstETHFinalized()** function, it is possible to pass unfinalized records, specifically, the **AssetsAccounting._finalizeUnstETHRecord()** function can return **(0, 0)** when the **claimableAmount** is 0 or when the status of **UnstETHRecord** is not **Locked**.
This will not lead to problems with assets accounting, but all the passed records will be included in the **UnstETHFinalized** event, which may cause issues with off-chain parsing.

### Recommendation

We recommend ensuring that unfinalized records are excluded from the event emission by reverting in case NFT can't be finalized.

### Client's comments

To address ambiguity in recognizing contract state changes, the **UnstETHFinalized** event has been modified to include detailed information about the exact amounts finalized for each individual unstETH NFT. This ensures that off-chain tooling can correctly distinguish finalized unstETH ids.

| INFORMATIONAL-10 | Users can unlock only the entire balance | Acknowledged |
|---|---|---|

### Description

Lines:

- Escrow.sol#L148
- Escrow.sol#L174

The **Escrow.unlockStETH()** and **Escrow.unlockWstETH()** functions do not accept call data to determine the token amount to unlock, forcing users to withdraw all available balances from the escrow.
Although, there is the possibility to unlock the portion of the locked funds via the conversion of shares to the withdrawal NFT using **Escrow.requestWithdrawals(uint256[] calldata stETHAmounts)** and **Escrow.unlockUnstETH(uint256[] memory unstETHIds)** functions.

### Recommendation

We recommend adding an ability to specify unlock amount.

### Client's comments

The decision to unlock all funds from the Signalling Escrow instance was made intentionally. Method **requestWithdrawals()** was removed in commit 525089b4634616bc6d6a43c477acc5ce6f1af665.

| INFORMATIONAL-11 | Redundant storage array | Fixed at: 5678106 |
|---|---|---|

### Description

Line: AssetsAccounting.sol#L25

The elements of the **AssetsAccounting.HolderAssets.unstETHIds** array are not used.

The array is populated during the NFT lock process and shrinks on NFT unlock.

The length is queried on the **Escrow.getVetoerState()** view function call.

Apart from the above instances, the array in question is not used.

### Recommendation

We recommend replacing the array with a simple NFT counter.

### Client's comments

The array of locked unstETH ids is preserved to allow users to obtain the list of ids needed for further actions such as **Escrow.unlockUnstETH()** and **Escrow.claimUnstETH()**. To facilitate this, the getter **Escrow.getVetoerUnstETHIds()** has been added in commit 56781068d8be992c5470cca3ded19d3afab6fa1b.

| INFORMATIONAL-12 | Missing parameter sanity checks | Fixed at: a299026 |
|---|---|---|

### Description

Lines:

- DualGovernanceConfig.sol#L43
- Proposers.sol#L68

The **Proposers.register()** does not check if the **proposerAccount** address is the same as the **executor** address.

The parameter **EscrowState.minAssetsLockDuration** is not validated against zero.

### Recommendation

We recommend appending a check for address equality **proposerAccount != executor** and the **minAssetsLockDuration > 0** (or some immutable minimum value for lock duration) check in the **DualGovernanceConfig.validate()** function.

| INFORMATIONAL-13 | The WithdrawalsBatchesQueue functions improvements | Fixed at: 140be7b |
|---|---|---|

### Description

Lines:

- WithdrawalBatchesQueue.sol#L276-L285
- WithdrawalBatchesQueue.sol#L214
- WithdrawalBatchesQueue.sol#L222
- WithdrawalBatchesQueue.sol#L229
- WithdrawalBatchesQueue.sol#L204
- WithdrawalBatchesQueue.sol#L245
- Escrow.sol#L423

In the **WithdrawalsBatchesQueue._getNextClaimableUnstETHIds()** function, the **unstETHIdsCountInTheBatch** variable is calculated on each iteration without changing the value.

```
+ uint256 unstETHIdsCountInTheBatch = currentBatch.lastUnstETHId - currentBatch.firstUnstETHId + 1;
for (uint256 i = 0; i < unstETHIdsCount; ++i) {
    info.lastClaimedUnstETHIdIndex += 1;
-    uint256 unstETHIdsCountInTheBatch = currentBatch.lastUnstETHId - currentBatch.firstUnstETHId + 1;
    if (unstETHIdsCountInTheBatch == info.lastClaimedUnstETHIdIndex) {
        info.lastClaimedBatchIndex += 1;
        info.lastClaimedUnstETHIdIndex = 0;
        currentBatch = self.batches[info.lastClaimedBatchIndex];
+        unstETHIdsCountInTheBatch = currentBatch.lastUnstETHId - currentBatch.firstUnstETHId + 1;
    }
    unstETHIds[i] = currentBatch.firstUnstETHId + info.lastClaimedUnstETHIdIndex;
}
```

The following **WithdrawalsBatchesQueue** internal functions are not used:

1. **WithdrawalsBatchesQueue.getBoundaryUnstETHId()**;
2. **WithdrawalsBatchesQueue.getTotalUnstETHIdsCount()**.

The **WithdrawalsBatchesQueue.getTotalUnclaimedUnstETHIdsCount()** and **WithdrawalsBatchesQueue.getNextWithdrawalsBatches()** functions (invoked from **Escrow.getUnclaimedUnstETHIdsCount()** and **Escrow.getNextWithdrawalBatch()** respectively) will return data regardless of the queue state. The same applies to the **WithdrawalsBatchesQueue.isAllBatchesClaimed()** function, but it is used in the correct context.

The **Escrow.isWithdrawalsBatchesFinalized()** function returns **WithdrawalsBatchesQueue.isClosed()**. However, the naming might lead to misunderstandings regarding the meaning of function return value.

E.g. users might think that if **Escrow.isWithdrawalsBatchesFinalized()** returns True, all **WithdrawalsBatchesQueue** requests can be claimed.

### Recommendation

We recommend removing unused functions; adding state checks according to the requested data, and renaming functions to accurately represent their actions.

| INFORMATIONAL–14 | Checking arrays' length | Acknowledged |
|---|---|---|

### Description

Line: Escrow.sol#L224

In function **Escrow.markUnstETHFinalized()** two arrays are passed. Every withdrawal request should be associated with its hint. Inside **WITHDRAWAL_QUEUE.getClaimableEther()** there is no check for length equality, so if they are not equal function will revert.

### Recommendation

We recommend adding a check for equality of **unstETHIds** and **hints** arrays.

### Client's comments

> The lengths of the arrays are intentionally not checked to align with the logic of the **WITHDRAWAL_QUEUE.getClaimableEther()** function, which does not enforce this check.

| INFORMATIONAL–15 | Inconsistent naming in Escrow contract and interface | Fixed at: 072ec77 |
|---|---|---|

### Description

Lines:

- Escrow.sol#L250
- IEscrow.sol#L10

The function **Escrow.startRageQuit()** has two parameters. In the **Escrow** contract, they are named **rageQuitExtensionPeriodDuration** and **rageQuitEthWithdrawalsDelay**, which is consistent with the docs. However, the naming of the interface parameters is different.

### Recommendation

We recommend having the same parameters' naming in contracts and interfaces.

| INFORMATIONAL–16 | Rage Quit support changes due to stETH rebalances | Acknowledged |
|---|---|---|

### Description

Rage Quit support in **Escrow** contract is based on the current **stETH** total supply and share rate. These values are changed on every **Lido** oracle report, so they affect the support and possible state transition of **Dual Governance** system.

### Recommendation

We recommend calling activation of the new state in **Dual Governance** with every oracle report or batching these transactions.

### Client's comments

> While changes to the Dual Governance state due to stETH rebalances from an Oracle report are possible, they are expected to be very rare. The current design assumes that **activateNextState()** will be called independently and trustlessly if the Oracle report triggers a change in the Dual Governance state.

| INFORMATIONAL-17 | Incorrect natspec | Fixed at: 83b4c3e |
|---|---|---|

### Description

Lines:

- libraries/AssetsAccounting.sol#L64
- libraries/WithdrawalBatchesQueue.sol#L8
- libraries/Proposers.sol#L31
- libraries/WithdrawalBatchesQueue.sol#L99
- EscrowState.sol#L14
- libraries/AssetsAccounting.sol#L34
- **DualGovernance.sol#L255**

In some cases pointed out above there are natspec inconsistencies and typos as well:

1. **/// @param state** is mentioned in the doc, while in the code, it's declared as **UnstETHRecordStatus status**.
2. **/// @param Empty The initial (uninitialized) state of the WithdrawalBatchesQueue** meanwhile the initial state is declared as **Absent**.
3. a typo: there is supposed to be 'the' instead of 'they':

```
/// @param executor Address of the executor associated with proposer. When proposer submits proposals, they execution
/// will be done with this address.
```

4. **This element doesn't used during the claiming** -> **This element isn't used during the claiming**
5. Describing **EscrowState**'s states the natspec claims:

```
/// @param RageQuitEscrow The final state of the Escrow contract. In this state, the Escrow instance acts as an accumulator
///        for withdrawn funds locked during the VetoSignalling phase.
```

while the funds can be locked during not only **vetoSignalling**, but **Normal** and **VetoSignallingDeactivation** phases too.

6. In struct **UnstETHAccounting** comment before second variable should be:

```
struct UnstETHAccounting {
    /// @dev slot0: [0..127]
    SharesValue unfinalizedShares;
    /// @dev slot0: [128..255]
    ETHValue finalizedETH;
}
```

7. The **DualGovernance.canSubmitProposal()** function's documentation states that proposals are forbidden in VetoSignalling and VetoSignallingDeactivation states, however according to the actual implementation proposals are only forbidden in the VetoCooldown and VetoSignallingDeactivation states.

### Recommendation

We recommend keeping the natspec documentation in the up-to-date and correct form.

### Client's comments

The majority of the NatSpec inaccuracies have been eliminated. Since NatSpec changes do not affect contract logic and only result in a metadata hash change in the bytecode, additional updates can be made separately later.

| INFORMATIONAL-18 | Uninitialized TimelockState after deployment | Fixed at: 6328ec1 |
|---|---|---|

### Description

Line: EmergencyProtectedTimelock.sol#L53
After **EmergencyProtectedTimelock** deployment, the **_timelockState** is left with the default zero values, meaning there is no delay for scheduling and executing proposals.
Current flow expects **EmergencyProtectedTimelock.setupDelay()** to be called after deployment to initialize the **_timelockState**.

### Recommendation

We recommend adding a modifier to **EmergencyProtectedTimelock.schedule()** and **EmergencyProtectedTimelock.execute()** that prevent calling them if the corresponding **_timelockState** values are unset.

### Client's comments

> The initial configuration of **afterSubmitDelay** and **afterScheduleDelay** has been moved to the constructor of the **EmergencyProtectedTimelock** contract, along with the introduction of the new sanity check parameter **MIN_EXECUTION_DELAY**. This ensures that the combined duration of **afterSubmitDelay** and **afterScheduleDelay** cannot fall below **MIN_EXECUTION_DELAY**. These changes are intended to reduce the risk of misconfiguration of the **EmergencyProtectedTimelock** during deployment and when updating delay values in the future.

| INFORMATIONAL-19 | Check for proposal.status in ExecutableProposals._isProposalMarkedCancelled() is required only in one case | Acknowledged |
|---|---|---|

### Description

Lines:

- libraries/ExecutableProposals.sol#L114
- libraries/ExecutableProposals.sol#L132
- libraries/ExecutableProposals.sol#L166
- libraries/ExecutableProposals.sol#L177

Function **ExecutableProposals._isProposalMarkedCancelled()** checks whether the proposal is cancelled by comparing **proposalId** to **lastCancelledProposalId** and ensuring that **proposal.status** is not **Executed**.
However, in all the four cases specified above, the **proposalState.status** is checked separately and it is redundant to check status for **Executed** inside the function.
For example, in **ExecutableProposals.schedule()** if the proposal status is anything but **Submitted**, the function would revert:

```
if (proposalState.status != Status.Submitted || _isProposalMarkedCancelled(self, proposalId, proposalState)) {
    revert ProposalNotSubmitted(proposalId);
}
```

Similarly in **ExecutableProposals.canSchedule()**, the function will always return **false** in case of an unexpected status:

```
if (_isProposalMarkedCancelled(self, proposalId, proposalState)) return false;
return proposalState.status == Status.Submitted
    && Timestamps.now() >= afterSubmitDelay.addTo(proposalState.submittedAt);
```

The only case where both checks are needed is inside **ExecutableProposals.getProposalDetails()** function.

### Recommendation

We recommend introducing a new private function that only checks **proposalId** against **lastCancelledProposalId** and using it in all cases except in **ExecutableProposals.getProposalDetails()**.

### Client's comments

> The extra check introduces negligible gas overhead compared to the overall transaction costs, whereas adding a new method would complicate the code.

| INFORMATIONAL-20 | Emergency protection can last longer than MAX_EMERGENCY_PROTECTION_DURATION | Acknowledged |
|---|---|---|

### Description

Lines: EmergencyProtection.sol#L98-L102
There is no start date for the emergency protection, the **EmergencyProtection.setEmergencyProtectionEndDate()** function checks the end date for the current timestamp.
Even if the **EmergencyProtection.emergencyProtectionEndsAfter** timestamp is passed, the DAO can reset the variable, prolonging all the previous committees as **EmergencyProtection.deactivateEmergencyMode()** cannot be called in the regular mode.

### Recommendation

We recommend introducing the emergency protection start timestamp to provide a fixed timeline for emergency protection.

### Client's comments

> The reconfiguration of emergency protection properties is secured by the Dual Governance mechanism and is designed to allow reactivation of emergency protection after updates to the Dual Governance system.

| INFORMATIONAL-21 | Redundant nonce check | Acknowledged |
|---|---|---|

### Description

Lines: TiebreakerCoreCommittee.sol#L113-L115
The **TiebreakerCoreCommittee.sealableResume()** function checks, if the provided nonce is equal to the nonce in the contract **TiebreakerCoreCommittee._sealableResumeNonces[sealable]**.

```
if (nonce != _sealableResumeNonces[sealable]) {
    revert ResumeSealableNonceMismatch();
}
```

When the voting is passed, the **TiebreakerCoreCommittee.executeSealableResume()** function increments the contract nonce value.
At the same time, the **TiebreakerSubCommittee** contract queries the nonce from the **TiebreakerCoreCommittee**, thus it has no control over the value.
As soon as the **TiebreakerCoreCommittee.executeSealableResume()** is executed, all the **TiebreakerSubCommittee** can vote and cast their votes only for new nonce, hence **TiebreakerCoreCommittee.sealableResume()** can't be invoked with outdated nonce.

### Recommendation

We recommend removing the redundant check.

### Client's comments

> The **TiebreakerCoreCommittee** does not restrict membership exclusively to **TiebreakerSubCommittee** contracts. The implemented check ensures that invalid data cannot be submitted, even if a custom subcommittee is used as a member.

| INFORMATIONAL-22 | Sanity check for Emergency governance and usual Governance | Acknowledged |
|---|---|---|

### Description

Lines:

- EmergencyProtection.sol#L81
- TimelockState.sol#L31

When setting a new address for **Emergency Governance** or for usual **Governance**, there is only one check to ensure that it is not equal to the previous address or zero address. This allows for setting the same address for emergency governance and main governance. In such case if **EmergencyProtectedTimelock.emergencyReset()** is called, then it will revert because the same address will be set for main governance.

### Recommendation

We recommend adding sanity checks for inequality of usual **Governance** and **Emergency Governance**.

### Client's comments

> The current behavior is intentional and aligns with the logic of the **emergencyReset()** method, which assigns the **emergencyGovernance** value to the **governance** variable without resetting **emergencyGovernance** to zero.

| INFORMATIONAL-23 | Sanity checks for Emergency committees | Acknowledged |
|---|---|---|

### Description

Lines:

- EmergencyProtection.sol#L131
- EmergencyProtection.sol#L142

When setting a new address for **EmergencyActivationCommittee** or for **EmergencyExecutionCommittee**, there is only one check to ensure that it is not equal to the previous address. So there is a possibility to set the same value for both committees which contradicts the specification and their purposes. They should be different based on logic of their usage and purpose except when one of them is zeroed.

### Recommendation

We recommend adding sanity checks for the inequality of two emergency committees except when they are zeroed.

### Client's comments

> While the specification and typical usage recommend using distinct addresses for these committees to ensure separation of duties, there may be situations where the same address is set for both the activation and execution committees. For example, if the committee implementation allows the same participants to perform actions for both committees but requires different quorums for each type.

| INFORMATIONAL–24 | Add system limit for number of proposers | Acknowledged |
|---|---|---|

### Description

Line: Proposers.sol#L68

When registering a new proposer, there is no limit on their number. This allows the addition of large quantities of governance systems which create risks for the entire protocol. Also unlimited number of proposers may lead to DoS of **Proposers.getAllProposers()** function.

### Recommendation

We recommend adding a general limit on the number of proposers in the system.

### Client's comments

Only the Admin Executor will have the authority to register new proposers, ensuring that the total number of proposers remains strictly limited. Additionally, the **Proposers.getAllProposers()** function is solely used for displaying the list of proposers and does not impact the operation of the Dual Governance system.

| INFORMATIONAL–25 | Imprecise conditions for pause check in ResealManager | Fixed at: c5e0359 |
|---|---|---|

### Description

Lines:

- ResealManager.sol#L36
- ResealManager.sol#L53

In functions **ResealManager.reseal()** and **ResealManager.resume()** there is a check if **sealable** is in paused state. It is needed, because during the call of **resume()** function in contracts inherited from **PausableUntil** it may revert if pause state is passed. This is the condition to check if a contract is paused based on **PausableUntil** implementation:

```
function _checkPaused() internal view {
    if (!isPaused()) {
        revert PausedExpected();
    }
}

function isPaused() public view returns (bool) {
    return block.timestamp < RESUME_SINCE_TIMESTAMP_POSITION.getStorageUint256();
}
```

So, if **RESUME_SINCE_TIMESTAMP** equals **block.timestamp,** then the contract is no longer in a pausable state.

### Recommendation

We recommend making checks for the pause state of **sealable** more precise in **ResealManager**.

```
...
if (sealableResumeSinceTimestamp <= block.timestamp) {
    revert SealableWrongPauseState();
}
...
```

| INFORMATIONAL-26 | Missing zero checks | Fixed at: 821c379 |
|---|---|---|

### Description

Lines:

- TimelockedGovernance.sol#L21
- TimelockedGovernance.sol#L22

In **TimelockedGovernance** constructor values for **GOVERNANCE** and **TIMELOCK** variables are set, but there is no check for zero address.

### Recommendation

We recommend adding checks for zero addresses.

| INFORMATIONAL-27 | Additional check when converting to IndexOneBased type | Fixed at: 2ece799 |
|---|---|---|

### Description

Line: IndexOneBased.sol#L33

In function **IndexOneBased.fromOneBasedValue() uint256** value is converted to **IndexOneBased** type which is **uint32**. It is used to represent position in an array as if the numbering began not from zero but from one. During conversion, there is only one check if the value exceeds **type(uint32).max**. According to the logic of using this type zero value also shouldn't be converted.

### Recommendation

We recommend adding a check for zero when converting **uint256** value to **IndexOneBased** type.

```
function fromOneBasedValue(uint256 oneBasedIndexValue) internal pure returns (IndexOneBased) {
    if (oneBasedIndexValue > type(uint32).max || oneBasedIndexValue == 0) {
        revert IndexOneBasedOverflow();
    }
    return IndexOneBased.wrap(uint32(oneBasedIndexValue));
}
```

| INFORMATIONAL-28 | Redundant underflow checks | Fixed at: 2ece799 |
|---|---|---|

### Description

Solidity 0.8.0 and up has built-in overflow and underflow checks.
The custom data types include their own underflow checks, but execute the subtraction in the base type, which results in a slight increase in gas costs.

1. Duration.minus()
2. Duration.minusSeconds()
3. ETHValue.minus()
4. PercentD16.minus()

### Recommendation

We recommend removing the duplicate underflow checks or adding the **unchecked** block to use custom errors.

| INFORMATIONAL-29 | HashConsensus._getHashState() doesn't return historical quorum and support | Fixed at: f0dc233 |
|---|---|---|

### Description

Lines: HashConsensus.sol#L103-L104
The function **HashConsensus._getHashState()** retrieves the **quorum** value from the current state, rather than storing and returning the historical quorum value associated with each specific proposal hash.
This can lead to inconsistencies, particularly if the **quorum** is changed via **HashConsensus.setQuorum()** or **HashConsensus.addMembers()**/**HashConsensus.removeMembers()** after a proposal is already scheduled or used, showing that the **support** is less then **quorum**.
Additionally, if a member that voted for a proposal is removed after the proposal has passed, the dynamically calculated **HashConsensus._getSupport()** return value will change.

### Recommendation

We recommend reflecting this behaviour in documentation and saving **quorum** and **suppport** values off-chain for past proposals.

| INFORMATIONAL-30 | TimelockedGovernance.executeProposal() is not part of IGovernance interface | Fixed at: 140175e |
|---|---|---|

### Description

Lines:

- IGovernance.sol#L8
- TimelockedGovernance.sol#L44

If the governance system scheduled a proposal and it is ready to be executed, anyone can execute it by making a call to the permissionless function **EmergencyProtectedTimelock.execute()**.
**TimelockedGovernance** contract contains a proxy function **TimelockedGovernance.executeProposal()** that makes the corresponding call to the **Timelock** contract, but it is not present in the **IGovernance** interface and isn't implemented in **DualGovernance**.

### Recommendation

We recommend removing the **TimelockedGovernance.executeProposal()** or implementing a similar function in the **DualGovernance** contract, as well as updating the **IGovernance** interface.

| INFORMATIONAL-31 | Define functions as external in HashConsensus | Fixed at: a428d7c |
|---|---|---|

### Description

Lines:

- HashConsensus.sol#L115
- HashConsensus.sol#L128
- HashConsensus.sol#L137
- HashConsensus.sol#L145
- HashConsensus.sol#L152
- HashConsensus.sol#L164
- HashConsensus.sol#L177

In these instances the functions are defined as **public**, but they aren't used within the child contract.

### Recommendation

We recommend changing functions visibility modifier to **external**.

| INFORMATIONAL-32 | Use >= instead of == in HashConsensus._vote() | Fixed at: f0dc233 |
|---|---|---|

### Description

Line: HashConsensus.sol#L64

The proposal is scheduled automatically if after the call to **HashConsensus._vote()** the **quorum** is reached.
However, it is possible for the contract owner to lower the **quorum** by calling **HashConsensus.setQuorum()** affecting all the pending votes. If after lowering the required quorum the necessary support level is reached, the voters should call **HashConsensus.schedule()** function to get the proposal scheduled. Because the strict **==** sign is used, excess voting will not schedule the proposal, unless the votes are retracted.

### Recommendation

We recommend using **>=** sign, scheduling proposals automatically even if the **quorum** was lowered.

| INFORMATIONAL-33 | Some custom errors are never used and incorrect error name is used | Fixed at: cc108d3 |
|---|---|---|

### Description

Lines:

- DualGovernanceConfig.sol#13
- Timestamp.sol#7
- types/PercentD16.sol#L30

Custom errors **InvalidSecondSealRageSupport(PercentD16 secondSealRageQuitSupport)** and **TimestampOverflow()** are defined, but never used.
Also in the definition of type **PercentD16::minus**, the error **Overflow()** is raised, but the name **Underflow()** would be more appropriate in this case.

### Recommendation

We recommend removing redundant definitions and using relevant custom error names.

### Client's comments

Fixed in 2ece7998e3d9ec4b4db09571c8359c1513f3143c and cc108d39c2301e3c194242702b0e7011258c859b

| INFORMATIONAL-34 | No sanity check of SanityCheckParams | Fixed at: b754de5 |
|---|---|---|

### Description

Lines:

- DualGovernance.sol#L56-L61
- EmergencyProtectedTimelock.sol#L30-L35

In **DualGovernance**, and **EmergencyProtectedTimelock** SanityCheckParams are passed in the constructor, but they are not checked in the code.

### Recommendation

We recommend adding checks for zero value and minimum < maximum.

### Client's comments

Fixed in 70542a52b92167ca8123aed08a1ce64bed39b734 and b754de5f5a908ddab83cfdc1abce70f07b870923. Checks on the limit ranges have been implemented. Zero checks will be handled through deployment scripts and validation.

| INFORMATIONAL-35 | Consistency in using Timestamps.now() | Fixed at: cc4f0e2 |
|---|---|---|

### Description

Line: HashConsensus.sol#L186

Here **Timestamps.now()** can be used instead of **Timestamps.from(block.timestamp)**.

### Recommendation

We recommend using **Timestamps.now()** instead of converting **block.timestamp**

| INFORMATIONAL-36 | EscrowState.setMinAssetsLockDuration() can be called out of initialization | Fixed at: 5aaab5b |
|---|---|---|

### Description

Line: EscrowState.sol#L98

If a malicious proposal is proposed by DAO to increase **minAssetsLockDuration**, some users can abstain from depositing to Escrow.

### Recommendation

We recommend setting the upper bound for **minAssetsLockDuration** to prevent infinite lock of assets.

| INFORMATIONAL-37 | Unused functions in SealableCalls | Fixed at: 00e2514 |
|---|---|---|

### Description

Lines:

- SealableCalls.sol#L18
- SealableCalls.sol#L63

**SealableCalls.callPauseFor()** and **SealableCalls.callResume()** in SealableCalls are not used in the protocol.

### Recommendation

We recommend removing these functions to reduce the bytecode size of the library.

| INFORMATIONAL-38 | Use Timestamp type in EnumerableProposals.Proposal structure | Fixed at: cc4f0e2 |
|---|---|---|

### Description

Line: EnumerableProposals.sol#L7

**EnumerableProposals.Proposal** structure has a **submittedAt** field defined as **uint40** instead of using a custom **Timestamp** type. It is inconsistent with other structures, such as **ExecutableProposals.ProposalData** where **Timestamp** is used for **submittedAt**.

While this value isn't used anywhere directly, it can be read via **ProposalsList.getProposals()**.

### Recommendation

We recommend using **Timestamp** type and it's auxilary library:

```
-Proposal memory proposal = Proposal(uint40(block.timestamp), proposalType, data);
+Proposal memory proposal = Proposal(Timestamps.now() proposalType, data);
```

| INFORMATIONAL-39 | Proposer is not saved in proposal data | Fixed at: 8c28258 |
|---|---|---|

### Description

Line: DualGovernance.sol#L134.

Dual Governance allows for multiple proposers to have the same executor, however, only the proposer executor is saved in the proposal data.

If a malicious proposer is added with a trusted executor, users will only see the trusted executor in the event and proposal getters.

### Recommendation

We recommend to include the proposer in the proposal data.

### Client's comments

Fixed in 2b85627687415a22ff3547ba2ea559be53a78b1d and 8c282583cc8327b691da97757537da6a9f59d2fb. The proposer address is now emitted in the event.

| INFORMATIONAL-40 | ProposalType.ResumeSealable is not included in encoding data | Fixed at: 9fecfff |
|---|---|---|

### Description

Line: TiebreakerSubCommittee.sol#L143.
In the function **TiebreakerSubCommittee._encodeSealableResume()**, **ProposalType.ResumeSealable** is not included in encoding data like in the function **TiebreakerCoreCommittee._encodeSealableResume()** which is inconsistent.

### Recommendation

We recommend to include **ProposalType.ResumeSealable** in the encoding data.

| INFORMATIONAL-41 | Tiebreaker condition inconsistency | Fixed at: 00e2514 |
|---|---|---|

### Description

Line: Tiebreaker.sol#L174
Per specification one of the tiebreaker conditions is **Rage Quit** state and
**protocol withdrawals are paused for a duration exceeding TiebreakerActivationTimeout**.
Nonetheless, the **Tiebreaker.isTie** implementation does not check the sealable pause duration.

### Recommendation

We recommend validating the tiebreaker conditions.

| INFORMATIONAL-42 | Sanity check for timelock delays | Acknowledged |
|---|---|---|

### Description

Lines: EmergencyProtectedTimelock.sol#L92-L115
The **EmergencyProtectedTimelock** contract imposes delays in the scheduling & execution of proposals.
The immutable parameters bound these delays:

1. **afterSubmitDelay <= MAX_AFTER_SUBMIT_DELAY**;
2. **afterScheduleDelay <= MAX_AFTER_SCHEDULE_DELAY**;
3. **afterSubmitDelay + afterScheduleDelay >= MIN_EXECUTION_DELAY**.

The constructor does not check if these immutable parameters contradict each other, e.g.
**MAX_AFTER_SUBMIT_DELAY + MAX_AFTER_SCHEDULE_DELAY < MIN_EXECUTION_DELAY**

### Recommendation

We recommend adding a sanity check in the constructor to enforce correct parameter values.

```
if (MAX_AFTER_SUBMIT_DELAY + MAX_AFTER_SCHEDULE_DELAY < MIN_EXECUTION_DELAY)
    revert Error();
```

### Client's comments

The existing check **afterSubmitDelay + afterScheduleDelay >= MIN_EXECUTION_DELAY** already ensures that **MAX_AFTER_SUBMIT_DELAY + MAX_AFTER_SCHEDULE_DELAY >= MIN_EXECUTION_DELAY** is satisfied. Adding an additional check would increase deployment gas costs in the common case while only providing an early failure in the unlikely event of misconfiguration.

STATE
MIND