CERTORA

Formal Verification Report for OpenZeppelin Governance Contracts

Summary

This document describes the specification and verification of OpenZeppelin's Governor module using the Certora Prover. The work was undertaken from October 31 to November 23, 2021. The latest commit that was reviewed and ran through the Certora Prover was 4088540a.

The scope of this verification is OpenZeppelin's governance system, particularly the following contracts:

- Governor.sol
- extensions/GovernorCountingSimple.sol
- extensions/GovernorProposalThreshold.sol
- extensions/GovernorTimelockControl.sol
- extensions/GovernorVotes.sol
- extensions/GovernorVotesQuorumFraction.sol

The Certora Prover proved the implementation of the Governance system is correct with respect to formal specifications written by the Certora team. The team also performed a manual audit of these contracts.

The formal specifications are focused on validating the integrity of the governance system — valid states of proposals, correct transitions between proposal states, invocation privileges and integrity of vote casting and counting. The formal specifications have been submitted as a pull request against OpenZeppelin's public git repository.

Disclaimer

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

We hope that this information is useful, but 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.

Main Issues Discovered

Severity: High

Issue:	**Setting the proposalThreshold too high breaks the proposing system**
Description:	In any case where the proposalThreshold becomes too high for any user to propose, it will be impossible to set a new, more reasonable threshold to the system. That is since the setter setProposalThreshold is guarded by the modifier onlyGovernance, which must be executed via a proposal. This situation may occur by either accidentally setting the threshold too high, or by changes in ERC20Votes token's market value which will price users out of the ability to raise proposals.
Response:	The obvious solution is to set a reasonable maximum to proposalThreshold, however we can't hardcode an opinionated maximum as it is highly dependent on the specifics of the governance system, e.g. different protocols may use different numbers of decimals and in this case a maximum threshold can be orders of magnitude off. An alternative is to provide the maximum as a configurable parameter, but this increases the complexity of setting up governance parameters in a way that could be counterproductive. The assumption is that a critical operation like changing the proposal threshold would be properly tested. Changes in the market value of the voting token are seen as a possibility that needs to be dealt with as part of the governance protocol.

Severity: High

Issue:	**Two systems that use the same timelock can attack each other**
Description:	In case that two rival systems use the same timelock in GovernorTimelockControl, each one of them can propse to invoke onlyGovenrnace functions on behalf the other system. **Since the timelock is the executor of both those systems, the function will be invoked successfuly**. This vulnerability can easily be exploited to achieve a DOS attack in the following manner: one system may raise a proposal to set the proposalThreshold of the other system to a very high value. After execution, it will be practically impossible to raise any proposals on the attacked system which will render it practically unusable.
Response:	We're looking into ways to mitigate this issue. In the meantime we will explain this possibility in our documentation and recommend against sharing a governance timelock with other executors.

Severity: High

Issue:	**Two systems that use the same timelock can not queue the same proposal**
Description:	Two systems that use the same TimelockController (given in the constructor), cannot queue the same proposal. In that case, the first system that calls queue() will queue the proposal and the second will revert. This can be exploited as a DoS attack if one system decides to attack the other. For example a system A, that has a shorter voting period can always propose the same proposal in system B and queue them before system B can. That will make all the proposals in system B inaccessible.
Response:	We're considering including the governor address as a salt so timelock ids do not clash between governors.

Severity: Low

Issue:	**Setting a new timelock removes all queued proposals along with the old timelock**
Description:	When queueing proposals in a timelock, the proposals are being stored and monitored by a specific address (timelock). Therefore, when changing a timelock, all the proposals that were queued but yet to be executed in the old timelock are now inaccessible through the current govenor. In this case, all these proposals that were in queueing state are lost, i.e. cannot be executed nor moved to the queue of the new timelock. **Note** that the community can vote to change the timelock back to the old one, thereby regaining access to these queued proposals.
Response:	Our recommendation would be to avoid changing the timelock while there are queued proposals. That said, we're considering alternatives to mitigate this at the contract level. Note that if the governor remains an executor in the timelock, a newer feature Governor.relay would allow triggering execution of a previously queued proposal.

Severity: Low

Issue:	**Voting period is 1 block less than expected**
Description:	The function state() categorize a proposal as **Active** if the current block number is in the range: [proposal.voteStart, proposal.voteEnd). On the other hand, when casting a vote, a require statement in the function demands: block.number > proposal.voteStart. This makes voting exactly at the start block impossible, even though the proposal is marked as **Active**. Note in particular that if a proposal's voting period is only one block long, the proposal will be entirely blocked.
Response:	Fixed in a more recent commit. A proposal is considered Pending during block number voteStart.

Summary of formal verification

Overview of OpenZeppelin/Governance contracts

Our verification efforts focused on the OpenZeppelin Governance module. The Governance module contains several abstract contracts that implement the core functionality of a governance system. Developers can combine and extend these contracts to implement a governance system that fits their individual needs. See the OpenZeppelin Contracts documentation for a high-level overview of the Governance module, and the Governance API for a detailed description.

A governance system allows a community of stakeholders to collectively make decisions about a project by voting on proposals. The governance system defines the requirements for creating proposals, the voting process, the requirements for accepting or rejecting a proposal, and the process for executing a proposal that has been accepted.

For our verification effort, we created multiple concrete governance systems by combining the components defined by the OpenZeppelin Governance module. Our concrete systems were based on the output of the OpenZeppelin Contracts Wizard. We then wrote general-purpose verification conditions that describe the correct operation of a governance system, and verified that the concrete governance systems satisfied those specifications.

The remainder of this section describes the rules and invariants that we have checked.

Assumptions and simplifications for verification

We made the following assumptions during our verification:

- When verifying contracts that make external calls, we assume that those calls can have arbitrary side effects outside of the contracts, but that they do not affect the state of the contract being verified. This means that some reentrancy bugs may not be caught.

- Due to limitations of the Certora Prover, two of the rules listed below timed out on certain methods. The list of unverified methods is included in the detailed description of each rule listed below. The Certora team is working to address these limitations.

- We assume that the values returned by different calls to ERC20Votes.getPastTotalSupply and ERC20Votes.getPastVotes in the same transaction return the same value.
- We assume that hash operations return an arbitrary deterministic value
- We unroll loops. Violations that require a loop to execute more than once will not be detected.

Verification conditions

Notation

✔ indicates the rule is formally verified on the latest reviewed commit. Footnotes describe any simplifications or assumptions used while verifying the rules (beyond the general assumptions listed above).

In this document, verification conditions are either shown as logical formulas or Hoare triples of the form {p} C {q}. A verification condition given by a logical formula denotes an invariant that holds if every reachable state satisfies the condition.

Hoare triples of the form {p} C {q} hold if any non-reverting execution of program C that starts in a state satsifying the precondition p ends in a state satisfying the postcondition q. The notation {p} C@withrevert {q} is similar but applies to both reverting and non-reverting executions. Preconditions and postconditions are similar to the Solidity require and assert statements.

Formulas relate the results of method calls. In most cases, these methods are getters defined in the contracts, but in some cases they are getters we have added to our harness or definitions provided in the rules file. Undefined variables in the formulas are treated as arbitrary: the rule is checked for every possible value of the variables.

Properties

✔ startAndEndDatesNonZero

Start and end dates are either initialized (non zero) or uninitialized (zero) simultaneously.

```
proposalSnapshot(proposalId) ≠ 0 ⇔ proposalDeadline(proposalId) ≠ 0
```

✔ voteStartBeforeVoteEnd

A proposal starting block number must be less than or equal to the proposal end block number.

```
proposalSnapshot(proposalId) > 0
  ⇒  proposalSnapshot(proposalId) ≤ proposalDeadline(proposalId)
```

✔ canceledImplyStartAndEndDateNonZero

If a proposal is canceled it must have a start date and an end date.

```
isCanceled(proposalId) ⇒ proposalSnapshot(proposalId) ≠ 0
```

✔ executedImplyStartAndEndDateNonZero

If a proposal is executed it must have a start date and an end date.

```
isExecuted(proposalId) ⇒ proposalSnapshot(proposalId) ≠ 0
```

✔ noBothExecutedAndCanceled

A proposal cannot be both executed and canceled simultaneously.

```
¬isExecuted(proposalId) v ¬isCanceled(proposalId)
```

executionOnlyIfQuoromReachedAndVoteSucceeded

A proposal can be executed only if quorum was reached and vote succeeded.

```
{
    isExecutedBefore = isExecuted(proposalId) ∧
    quorumReachedBefore = _quorumReached(e, proposalId) ∧
    voteSucceededBefore = _voteSucceeded(proposalId)
}
    <transaction>
{
    isExecutedAfter = isExecuted(proposalId) ∧
    ((¬isExecutedBefore ∧ isExecutedAfter) ⇒ (quorumReachedBefore ∧
voteSucceededBefore))
}
```

doubleVoting[1]

A user cannot vote twice.

```
{
    hasVoted(proposalId, msg.sender)
}
    castVote@withrevert(proposalId, support)
{
    lastReverted
}
```

[1] We verified this property on the castVote method but not the other two vote casting functions. We feel this is reasonable since the bulk of the code for all three functions is contained in the _castVote helper method.

✔ immutableFieldsAfterProposalCreation

Once a proposal is created, voteStart and voteEnd are immutable.

```
{
    _voteStart = proposalSnapshot(proposalId) ∧
    uint256 _voteEnd = proposalDeadline(proposalId) ∧
    proposalCreated(proposalId)
}
    <transaction>
{
    voteStart_ = proposalSnapshot(proposalId) ∧
    voteEnd_ = proposalDeadline(proposalId) ∧
    _voteStart == voteStart_ ∧ _voteEnd == voteEnd_
}
```

✔ noStartBeforeCreation

Voting cannot start at a block number prior to proposal's creation block number.

```
{
    previousStart = proposalSnapshot(proposalId) ∧
    ¬proposalCreated(proposalId)
}
    propose(e, args)
{
    newStart = proposalSnapshot(proposalId) ∧
    newStart ≠ previousStart ⇒ newStart ≥ e.block.number
}
```

✔ noExecuteOrCancelBeforeDeadline

A proposal can neither be executed nor canceled before it ends.

```
{
    ¬isExecuted(proposalId) /\ ¬isCanceled(proposalId)
}
    <transaction>
{
    e.block.number < proposalDeadline(proposalId) ⇒
(¬isExecuted(proposalId) /\ ¬isCanceled(proposalId))
}
```

✔ executedOnlyAfterExecuteFunc

Proposal can be switched to executed only via execute() function

```
{ ¬isExecuted(proposalId) }

<non-execute operation on proposalId>

{ ¬isExecuted(proposalId) }
```

✔ allFunctionsRevertIfExecuted and allFunctionsRevertIfCanceled[2]

All non-view functions should revert if proposal is executed/canceled.

```
{ isExecuted(proposalId) }

<non-view operation on proposalId>@withrevert

{ lastReverted }
```

[2] due to timeouts, these two properties were not verified on calls to view functions, updateQuorumNumerator(), updateTimelock(), queue() and __acceptAdmin() methods

and

```
{ isCanceled(proposalId) }

<non-view operation on proposalId>@withrevert

{ lastReverted }
```

✔ SumOfVotesCastEqualSumOfPowerOfVotedPerProposal

The sum of all votes casted is equal to the sum of voting power of those who voted, per proposal.

```
tracked_weight(proposalId) == ghost_sum_vote_power_by_id(proposalId)
```

✔ SumOfVotesCastEqualSumOfPowerOfVoted

The sum of all votes casted is equal to the sum of voting power of those who voted.

```
sum_tracked_weight() == sum_all_votes_power()
```

✔ OneIsNotMoreThanAll

The sum of all votes casted is greater or equal to the sum of voting power of those who voted as per specific proposal.

```
sum_all_votes_power() ≥ tracked_weight(proposalId)
```

✔ noVoteForSomeoneElse[3]

Only sender's voting status can be changed by execution of any cast vote function.

```
{
    userVoteBefore = hasVoted(proposalId, user)
}
castVote(proposalId, sup)
{
    userVoteAfter = hasVoted(proposalId, user) ∧
    user ≠ msg.sender ⇒ otherUserVoteBefore = otherUserVoteAfter
}
```

✔ votingWeightMonotonicity

Total voting tally is monotonically non-decreasing in every operation.

```
{
    votingWeightBefore = sum_tracked_weight()
}
<transaction>
{
    votingWeightAfter = sum_tracked_weight() ∧
    votingWeightBefore ≤ votingWeightAfter
}
```

[3] We verified this property on the castVote method but not the other two vote casting functions. We feel this is reasonable since the bulk of the code for all three functions is contained in the _castVote helper method.

✔ hasVotedCorrelation

A change in hasVoted must be correlated with a non-decreasing change of the vote supports (non-decreasing because user is allowed to vote with weight 0).

```
{
    acc = e.msg.sender ∧
    againstBefore = votesAgainst() ∧
    forBefore = votesFor() ∧
    abstainBefore = votesAbstain() ∧
    hasVotedBefore = hasVoted(e, proposalId, acc)
}
<transaction on proposalId>
{
    (againstAfter = votesAgainst() ∧
    forAfter = votesFor() ∧
    abstainAfter = votesAbstain() ∧
    hasVotedAfter = hasVoted(e, proposalId, acc) ∧
    hasVotedAfter_User = hasVoted(e, proposalId, user) c
    ((¬hasVotedBefore ∧ hasVotedAfter) ⇒ againstBefore ≤
againstAfter ∨ forBefore ≤ forAfter v abstainBefore ≤ abstainAfter))
}
```

✔ privilegedOnlyNumerator and privilegedOnlyDenominator

Only privileged users can execute privileged operations, e.g. change _quorumNumerator or _timelock.

```
{
    quorumNumBefore = quorumNumerator(e)
}
    <transaction>
{
    quorumNumAfter = quorumNumerator(e) ∧
    address executorCheck = getExecutor(e) ∧
    (quorumNumBefore ≠ quorumNumAfter ⇒ e.msg.sender ==
executorCheck)
}
```

and

```
{ timelockBefore = timelock() }

<transaction>

{ (timelock() ≠ timeLockBefore ⇒ e.msg.sender == timelockBefore) }
```