CERTORA

Formal Verification of OpenZeppelin (May - June 2022)

Summary

This document describes the specification and verification of OpenZeppelin’s contracts using the Certora Prover. The work was undertaken from May 9th to June 10th. The latest commit that was reviewed and run through the Certora Prover was commit `109778c`.

The scope of our verification was the following contracts:

- `Initializable.sol` (Verification Result)
- `GovernorPreventLateQuorum.sol` (Verification Result)
- `ERC1155Burnable.sol` (Verification Result)
- `ERC1155Pausable.sol` (Verification Result)
- `ERC1155Supply.sol` (Verification Result)
- `ERC1155Holder.sol` (Formal Verification Unnnecessary)
- `ERC1155Receiver.sol` (Formal Verification Unnnecessary)

The Certora Prover proved the implementation of the Open Zeppelin contracts is correct with respect to the formal rules written by the Open Zeppelin and the Certora teams. During the verification process, the Certora Prover discovered bugs in the code listed in the table below. All issues were promptly corrected, and the fixes were verified to satisfy the specifications up to the limitations of the Certora Prover. The Certora development team is currently handling these limitations. The next section formally defines high level specifications of Open Zeppelin. All the rules are publically available in a public github.

List of Main Issues Discovered

Severity: **High Medium Low**

Issue:	**Calling `updateQuorumNumerator()` can change the output of `quorumReached()` for previous proposals, leading to unexpected outcomes.**
Rules Broken:	`quorumReachedEffect` , `proposalNotCreatedEffects` , `proposalInOneState` , `deadlineCantBeUnextended`
Description:	• **High** Decreasing the number of votes required for a proposal to reach quorum can allow proposals which are currently active, passing, and unexecutable to become immediately executable. Breaks rules `quorumReachedEffect` , `proposalNotCreatedEffects` , and `proposalInOneState` . • **Medium** Decreasing the number of votes required for a proposal to reach quorum can allow proposals to reach quorum late without extending their deadlines. Breaks rules `quorumReachedEffect` , `proposalNotCreatedEffects` , and `proposalInOneState` . • **Low** Increasing the number of votes required for a proposal to reach quorum can cause proposals which had previously reached quorum to no longer be in quorum. Breaks rule `deadlineCantBeUnextended` .
Response:	We agree that this is a significant issue and will change `GovernorVotesQuorumFraction` so that changes to quorum requirements do not affect past proposals. Additionally, we are looking for affected instances of this contract on-chain to reach out and notify of the potential issue.

Severity: **Low**

Issue:	**A governance with a voting token that has 0 totalSupply will consider all current and future proposals to have reached quorum.**
Rules Broken:	`quorumReachedEffect` , `proposalNotCreatedEffects` , `proposalInOneState`
Description:	A voting token with 0 token supply will result in all proposals being considered as having reached quorum. This can be an issue in the case that the token has not been initialized/minted, but this case is not as interesting because there will be no tokens to vote with. A more interesting case can arise if the voting token's `totalSupply` is accidentally set to 0. This will allow all proposals to reach quorum and thus be executable as long as the vote is successful.
Response:	This is an edge case that should never manifest as long as tokens withhold the invariant that total supply is equal to the sum of all balances, as in this case no one will be able to vote for a proposal and the condition for a successful proposal will never be met (more for votes than against votes).

Severity: Low

Issue:	**TimelockController should not have additional executors beside the governor [GovernorTimelockControl _execute()]**
Rules Broken:	None
Description:	An executor can execute a scheduled operation on the TimelockController by calling TimelockController.execute. If the operation was queued using GovernorTimelockControl.queue, this will cause GovernorTimelockControl.execute to revert as the proposal has already been executed by the TimelockController. (Same issue with calling TimelockController.cancel)
Response:	Agreed, but probably not any significant consequence. The only consequence is that if the proposal is executed directly in the timelock, the "ProposalExecuted" event will never be emitted.

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.

Notations

✔️ indicates the rule is formally verified on the latest reviewed commit. We write ✓* when the rule was verified on a simplified version of the code (or under some assumptions).

❌ indicates the rule was violated under one of the tested versions of the code.

✍️ indicates the rule is not yet formally specified.

🔁 indicates the rule is postponed (<due to other issues, low priority>) .

⏱️ indicates that some functions cannot be verified because the rules timed out
Footnotes describe any simplifications or assumptions used while verifying the rules (beyond the general assumptions listed above).

Verification of Initializable

`Initializable` is a contract used to make constructors for upgradeable contracts. This is accomplished by applying the `initializer` modifier to any function that serves as a constructor, which makes this function only callable once. The secondary modifier `reinitializer` allows for upgrades that change the contract's initializations.

Assumptions and Simplifications

We assume `initializer()` and `reinitializer(1)` are equivalent if they both guarentee `_initialized` to be set to 1 after a successful call. This allows us to use `reinitializer(n)` as a general version that also handles the regular `initialzer` case.

Harnessing

Two harness versions were implemented, a simple flat contract, and a multi-inheriting contract. The two versions together help us ensure there are no unexpected results because of different implementions. Initializable can be used in many different ways but we believe these 2 cases provide good coverage for all cases. In both harnesses we use getter functions for `_initialized` and `_initializing` and implement `initializer` and `reinitializer` functions that use their respective modifiers. We also implement some versioned functions that are only callable in specific versions of the contract to mimick upgrading contracts.

Munging

Variables `_initialized` and `_initializing` were changed to have internal visibility to be harnessable.

Definitions

`isUninitialized:` A contract's `_initialized` variable is equal to 0.

`isInitialized:` A contract's `_initialized` variable is greater than 0.

`isInitializedOnce:` A contract's `_initialized` variable is equal to 1.

`isReinitialized:` A contract's `_initialized` variable is greater than 1.

`isDisabled:` A contract's `_initialized` variable is equal to 255.

Properties

(✓) invariant `notInitializing`
A contract must only be in the `_initializing` state if and only if the contract is in the middle of an initializing transaction execution.

(✓) rule `initOnce`
An initializeable contract with a function that inherits the initializer modifier must be initializable only once.

(✓) rule `reinitializeEffects`
Successfully calling `reinitialize()` with a version value of 1 must result in `_initialized` being set to 1.

(✓) rule `initalizeEffects`
Successfully calling `initalize()` must result in `_initialized` being set to 1.

(✓) rule `disabledStaysDisabled`
A disabled initializable contract must always stay disabled.

(✓) rule `increasingInitialized`
The variable `_initialized` must not decrease.

(✓) rule `reinitializeIncreasesInit`
If reinitialize(...) was called successfuly, then the variable `_initialized` must increase.

(✓) rule `reinitializeLiveness`
Reinitialize(n) must be callable if the contract is not in an _initializing state and n is greater than _initialized and less than 255.

(✓) rule `reinitializeRule`
If reinitialize(n) was called successfully then n was greater than _initialized.

(✓) rule `reinitVersionCheckParent`
Functions implemented in the parent contract that need `_initialized` to be a equal to some value *n* in order to be called, are only callable when `_initialized` is equal to *n*.

(✓) rule `reinitVersionCheckChild`
Functions implemented in the child contract that need `_initialized` to be a equal to some value *n* in order to be called, are only callable when `_initialized` is equal to *n*.

(✓) rule `reinitVersionCheckGrandchild`
Functions implemented in the grandchild contract that need `_initialized` to be a equal to some value *n* in order to be called, are only callable when `_initialized` is equal to *n*.

(✓) rule `inheritanceCheck`
Calling parent initalizer function must initialize all child contracts.

Verification of GovernorPreventLateQuorum

`GovernorPreventLateQuorum` extends the Governor group of contracts to add the feature of giving voters more time to vote in the case that a proposal reaches quorum with less than `voteExtension` amount of time left to vote.

Assumptions and Simplifications

Harnessing

- The contract that the specification was verified against is `GovernorPreventLateQuorumHarness`, which inherits from all of the Governor contracts — excluding Compound variations — and implements a number of view functions to gain access to values that are impossible/difficult to access in CVL. It also implements all of the required functions not implemented in the abstract contracts it inherits from.
- `_castVote` was overriden to add an additional flag before calling the parent version. This flag stores the `block.number` in a variable `latestCastVoteCall` and is used as a way to check when any of variations of `castVote` are called.

Munging

- Various variables' visibility was changed from private to internal or from internal to public throughout the Governor contracts in order to make them accessible in the spec.
- Arbitrary low level calls are assumed to change nothing and thus the function `_execute` is changed to do nothing. The tool normally havocs in this situation, assuming all storage can change due to possible reentrancy. We assume, however, there is no risk of reentrancy because `_execute` is a protected call locked behind the timelocked governance vote. All other governance functions are verified separately.

Definitions

`deadlineExtendible:` A proposal is defined to be `deadlineExtendible` if its respective `extendedDeadline` variable is unset and quorum on that proposal has not been reached.

`deadlineExtended:` A proposal is defined to be `deadlineExtended` if its respective `extendedDeadline` variable is set and quorum on that proposal has been reached.

`proposalNotCreated:` A proposal is defined to be `proposalNotCreated` if its snapshot (block.number at which voting started), deadline, and `totalVotes` all equal 0.

Properties

(✓) rule `deadlineChangeEffects`

If deadline increases then we are in a `deadlineExtended` state and `castVote` has been called.

(✗) rule `deadlineCantBeUnextended`

A proposal must not leave a `deadlineExtended` state.

(✓) rule `canExtendDeadlineOnce`

A proposal's deadline must not change once in a `deadlineExtended` state.

(✓) rule `hasVotedCorrelationNonzero`

A change in `hasVoted` for a given account and proposal must correlate positively with an increase in the number of votes for one of the vote categories, *e.g.* `abstainVotes`, `againstVotes`, or `forVotes`. Additionally, the `totalVotes` must not decrease.

(✓) rule `againstVotesDontCount`

`againstVotes` for a given proposal must not contribute to the proposal's quorum.

(✓) rule `deadlineExtenededIfQuorumReached`

The deadline for a given proposal must only be extended from a `deadlineExtendible` state with quorum being reached and with ≤

`lateQuorumVoteExtension` time left to vote.

(✓) *rule* `extendedDeadlineValueSetIfQuorumReached`
`extendedDeadline` is set if and only if `_castVote` is called and quorum is reached.

(✓) *rule* `deadlineNeverReduced`
The deadline for a given proposal must never be reduced.

(✗) *invariant* `quorumReachedEffect`
If a proposal has reached quorum then the proposal snapshot (`block.number` at which voting started) must be non-zero.

(✗) *invariant* `proposalNotCreatedEffects`
A non-existant proposal’s snapshot, deadline, and `totalVotes` must all equal 0.

(✗) *invariant* `proposalNotCreatedEffects`
A non-existant proposal’s snapshot, deadline, and `totalVotes` must all equal 0.

Verification of ERC1155

`ERC1155` establishes base level support EIP1155, a standard interface for contracts that manage multiple token types. The contract was verified as part of previous work with OpenZeppelin and is included here for the purposes of increased verification coverage with respect to token transfer methods.

Assumptions and Simplifications

- Internal burn and mint methods are wrapped by CVT callable functions.

Properties

The following properties are additions to the previous `ERC1155` verification. Please see the previous report for earlier contract properties verified.

(✓) *rule* `singleTokenSafeTransferFromSafeBatchTransferFromEquivalence`
The result of transferring a single token must be equivalent whether done via safeTransferFrom or safeBatchTransferFrom.

(✓) *rule* `multipleTokenSafeTransferFromSafeBatchTransferFromEquivalence`
The results of transferring multiple tokens must be equivalent whether done separately via safeTransferFrom or together via safeBatchTransferFrom.

(✓) *rule* `transfersHaveSameLengthInputArrays`
If transfer methods do not revert, the input arrays must be the same length.

Verification of ERC1155Burnable

`ERC1155Burnable` extends the `ERC1155` functionality by wrapping the internal methods `_burn` and `_burnBatch` in the public methods `burn` and `burnBatch`, adding a requirement that the caller of either method be the account holding the tokens or approved to act on that account's behalf.

Assumptions and Simplifications

- No changes made using the harness

Properties

(✓) rule ***`onlyHolderOrApprovedCanReduceBalance`***
If a method call reduces account balances, the caller must be either the holder of the account or approved to act on the holder's behalf.

(✓) rule ***`burnAmountProportionalToBalanceReduction`***
Burning a larger amount of a token must reduce that token's balance more than burning a smaller amount. This rule holds for `burnBatch` as well due to rules establishing appropriate equivance between `burn` and `burnBatch` methods.

(✓) rule ***`sequentialBurnsEquivalentToSingleBurnOfSum`***
Two sequential burns must be equivalent to a single burn of the sum of their amounts. This rule holds for also `burnBatch` due to rules establishing appropriate equivance between `burn` and `burnBatch` methods.

(✓) rule ***`singleTokenBurnBurnBatchEquivalence`***
The result of burning a single token must be equivalent whether done via burn or burnBatch.

(✓) rule ***`multipleTokenBurnBurnBatchEquivalence`***
The results of burning multiple tokens must be equivalent whether done separately via burn or together via burnBatch.

(✓) rule ***`burnBatchOnEmptyArraysChangesNothing`***
If passed empty token and burn amount arrays, burnBatch must not change token balances or address permissions.

Verification of ERC1155Pausable

`ERC1155Pausable` extends existing `Pausable` functionality by requiring that a contract not be in a `paused` state prior to a token transfer.

Assumptions and Simplifications

- Internal methods `_pause` and `_unpause` wrapped in CVT callable versions
- Dummy functions created to verify `whenPaused` and `whenNotPaused` modifiers

Properties

(✓) *rule* `balancesUnchangedWhenPaused`
When a contract is in a paused state, the token balance for a given user and token must not change.

(✓) *rule* `transferMethodsRevertWhenPaused`
When a contract is in a paused state, transfer methods must revert.

(✓) *rule* `pauseMethodPausesContract`
When a contract is in an unpaused state, calling `pause` must transition to a paused state.

(✓) *rule* `unpauseMethodUnpausesContract`
When a contract is in a paused state, calling `unpause` must transition to an unpaused state.

(✓) *rule* `cannotPauseWhilePaused`
When a contract is in a paused state, calling `pause` must revert.

(✓) *rule* `cannotUnpauseWhileUnpaused`
When a contract is in an unpaused state, calling `unpause` must revert.

(✓) *rule* `whenNotPausedModifierCausesRevertIfPaused`
When a contract is in a paused state, functions with the `whenNotPaused` modifier must revert.

(✓) *rule* `whenPausedModifierCausesRevertIfUnpaused`
When a contract is in an unpaused state, functions with the `whenPaused` modifier must revert.

Verification of ERC1155Supply

`ERC1155Supply` extends the `ERC1155` functionality. The contract creates a publicly callable `totalSupply` wrapper for the private `_totalSupply` method, a public `exists` method to check for a positive balance of a given token, and updates `_beforeTokenTransfer` to appropriately change the mapping `_totalSupply` in the context of minting and burning tokens.

Assumptions and Simplifications

- The `exists` method was wrapped in the `exists_wrapper` method because `exists` is a keyword in CVL.

- The public functions `burn` , `burnBatch` , `mint` , and `mintBatch` were implemented in the harnesssing contract make their respective internal functions callable by the CVL. This was used to test the increase and decrease of `totalSupply` when tokens are minted and burned.
- We created the `onlyOwner` modifier to be used in the above functions so that they are not called in unrelated rules.

Properties

(✓) invariant `total_supply_is_sum_of_balances`
The sum of the balances over all users must equal the total supply for a given token.

(✓) invariant `balanceOfZeroAddressIsZero`
The balance of a token for `address(0)` must be zero.

(✓) rule `token_totalSupply_independence`
Given two different token ids, if `totalSupply` for one changes, then `totalSupply` for the other must not.

(✓) rule `held_tokens_should_exist`
If a user has a token, then the token must exist.

Bug Injection Test

In this section we intentionally create bugs to check if we have coverage for those type of bugs.
We do this to make sure that even if an attacker managed to get into such a situation he would not be able to harm the system.

(✓) Bug1: mutate `_castVote` function in `GovernorPreventLateQuorum.sol`
catching rule(s): `extendedDeadlineValueSetIfQuorumReached` [Tool Output]

This change will cause the deadline be equal to the block time instead expanding it:

```
-    uint64 extendedDeadlineValue = block.number.toUint64() + lateQuorumVoteE
+    uint64 extendedDeadlineValue = block.number.toUint64();
```

(✓) Bug2: mutate `_beforeTokenTransfer` function in `ERC1155Pausable.sol`
catching rule(s): `balancesUnchangedWhenPaused` , `transferMethodsRevertWhenPaused` [Tool Output]

This lack of require will allow transfer while paused:

```
-    require(!paused(), "ERC1155Pausable: token transfer while paused");
+    // require(!paused(), "ERC1155Pausable: token transfer while paused");
```

(✓) Bug3: mutate `_castVote` function in `GovernorPreventLateQuorum.sol`

catching rule(s): `deadlineChangeEffects` [Tool Output]

This change will allow a proposal to extend the deadline even if it doesn't reach quorum:

```
-    if (extendedDeadline.isUnset() && _quorumReached(proposalId)) {
+    // if (extendedDeadline.isUnset() && _quorumReached(proposalId)) {
+    if (extendedDeadline.isUnset()) {
```

(✓) Bug4: mutate `burn` function in `ERC1155Burnable.sol`

catching rule(s): `onlyHolderOrApprovedCanReduceBalance` [Tool Output]

This lack of require will allow anyone to burn tokens for an account:

```
-    require(account == _msgSender() || isApprovedForAll(account, _msgSender(
+    // require(account == _msgSender() || isApprovedForAll(account, _msgSenc
```

(✓) Bug5: mutate `_beforeTokenTransfer` function in `ERC1155Supply.sol`

catching rule(s): `total_supply_is_sum_of_balances` [Tool Output]

This change will cause the total supply not to increase when a token is transfered (or minted):

```
-    _totalSupply[ids[i]] += amounts[i];
+    // _totalSupply[ids[i]] += amounts[i];
```

(✓) Bug6: mutate `_beforeTokenTransfer` function in `ERC1155Supply.sol`

catching rule(s): `total_supply_is_sum_of_balances` [Tool Output]

This change will cause total supply to increase upon token transfer only for the account at `i = 0` instead of for all appropriate accounts:

```
-    _totalSupply[ids[i]] += amounts[i];
+    // _totalSupply[ids[i]] += amounts[i];
+    _totalSupply[ids[0]] += amounts[i];
```

(✓) Bug7: mutate `burn` function in `ERC1155Burnable.sol`

catching rule(s): `burnAmountProportionalToBalanceReduction` , `sequentialBurnsEquivalentToSingleBurnOfSum` ,