



# 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 Unnecessary)
- [ERC1155Receiver.sol](#) (Formal Verification Unnecessary)

The Certora Prover proved the implementation of the OpenZeppelin contracts is correct with respect to the formal rules written by the OpenZeppelin 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 OpenZeppelin. All the rules are publicly available in a [public github](#).

The OpenZeppelin team is continuously verifying these properties as they develop their code. You can see the latest results [here](#).

## List of Main Issues Discovered

---

<b>Issue:</b>	<b>Calling <code>updateQuorumNumerator()</code> can change the output of <code>quorumReached()</code> for previous proposals, leading to unexpected outcomes.</b>
Rules Broken:	<code>quorumReachedEffect</code> , <code>proposalNotCreatedEffects</code> , <code>proposalInOneState</code> , <code>deadlineCantBeUnextended</code>
Description:	<ul style="list-style-type: none"> <li>• <b>High</b> 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 <code>quorumReachedEffect</code> , <code>proposalNotCreatedEffects</code> , and <code>proposalInOneState</code> .</li> <li>• <b>Medium</b> 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 <code>quorumReachedEffect</code> , <code>proposalNotCreatedEffects</code> , and <code>proposalInOneState</code> .</li> <li>• <b>Low</b> 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 <code>deadlineCantBeUnextended</code> .</li> </ul>
Response:	We agree that this is a significant issue and will change <code>GovernorVotesQuorumFraction</code> 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**

<b>Issue:</b>	<b>A governance with a voting token that has 0 total supply will consider all current and future proposals to have reached quorum.</b>
Rules Broken:	<code>quorumReachedEffect</code> , <code>proposalNotCreatedEffects</code> , <code>proposalInOneState</code>

<b>Issue:</b>	<b>A governance with a voting token that has 0 total supply will consider all current and future proposals to have reached quorum.</b>
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 <code>totalSupply</code> 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**

<b>Issue:</b>	<code>TimelockController</code> <b>should not have additional executors beside the governor ( <code>GovernorTimelockControl._execute()</code> )</b>
Rules Broken:	None
Description:	An executor can execute a scheduled operation on the <code>TimelockController</code> by calling <code>TimelockController.execute</code> . If the operation was queued using <code>GovernorTimelockControl.queue</code> , this will cause <code>GovernorTimelockControl.execute</code> to revert as the proposal has already been executed by the <code>TimelockController</code> . (Same issue with calling <code>TimelockController.cancel</code> )
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 has not been checked on the current version.

🕒 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 upgradable 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 should run at most once after the contract is upgraded.

## Assumptions and Simplifications

We assume `initializer()` and `reinitializer(1)` are equivalent if they both guarantee `_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 `initializer` 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 implementations. `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 mimic 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

-  **Not initializing.** A contract must only ever be in an initializing state while in the middle of a transaction execution. ([report](#))
-  **Only initialized once.** An initializable contract with a function that inherits the initializer modifier must be initializable only once ([report](#))
-  **Reinitialize effects.** Successfully calling `reinitialize()` with a version value of 1 must result in `_initialized` being set to 1. ([report](#))
-  **Initialize effects.** Successfully calling `initialize()` must result in `_initialized` being set to 1. ([report](#))
-  **Disabled stays disabled.** A disabled initializable contract must always stay disabled. ([report](#))
-  **Increasing initialized.** The variable `_initialized` must not decrease. ([report](#))

-  **Reinitialize increases** `init`. If `reinitialize(...)` was called successfully, then the variable `_initialized` must increase. ([report](#))
-  **Reinitialize liveness**. `reinitialize(n)` must be callable if the contract is not in an `_initializing` state and `n` is greater than `_initialized` and less than 255 ([report](#))
-  **Reinitialize rule**. if `reinitialize(n)` was called successfully then `n` was greater than `_initialized`. ([report](#))
-  **Reinitialize version check parent**. Functions implemented in the parent contract that require `_initialized` to be a certain value are only callable when it is that value. ([report](#))
-  **Reinitialize version check child**. Functions implemented in the child contract that require `_initialized` to be a certain value are only callable when it is that value. ([report](#))
-  **Reinitialize version check grandchild**. Functions implemented in the grandchild contract that require `_initialized` to be a certain value are only callable when it is that value. ([report](#))
-  **Inheritance check**. Calling parent initializer function must initialize all child contracts. ([report](#))

## 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 functions callable from CVL.

## Properties

These properties are additions to the previous `ERC1155` verification. Please see the file `ERC1155.spec` for earlier contract properties verified.

- ✓ **Single token safe transfer from safe batch transfer from equivalence.** The result of transferring a single token must be equivalent whether done via `safeTransferFrom` OR `safeBatchTransferFrom`. ([report](#))
- ✓ **Multiple token safe transfer from safe batch transfer from equivalence.** The results of transferring multiple tokens must be equivalent whether done separately via `safeTransferFrom` OR together via `safeBatchTransferFrom`. ([report](#))
- ✓ **Transfers have same length input arrays.** If transfer methods do not revert, the input arrays must be the same length. ([report](#))

## 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

- ✓ **Only holder or approved can reduce balance.** 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. ([report](#))
- ✓ **Burn amount proportional to balance reduction.** Burning a larger amount of a token must reduce that token's balance more than burning a smaller amount. n.b. This rule holds for `burnBatch` as well due to rules establishing appropriate equivalence between `burn` and `burnBatch` methods. ([report](#))
- ✓ **Sequential burns equivalent to single burn of sum.** Two sequential burns must be equivalent to a single burn of the sum of their amounts. This rule holds for `burnBatch` as well due to rules establishing appropriate equivalence between `burn` and `burnBatch` methods. ([report](#))
- ✓ **Single token `burn` / `burnBatch` equivalence.** The result of burning a single token must be equivalent whether done via `burn` OR `burnBatch`. ([report](#))

-  **Multiple token `burn` / `burnBatch` equivalence.** The results of burning multiple tokens must be equivalent whether done separately via `burn` or together via `burnBatch`. ([report](#))
-  **Burn batch on empty arrays changes nothing.** If passed empty token and burn amount arrays, `burnBatch` must not change token balances or address permissions. ([report](#))

## 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 by functions callable from CVL
- Dummy functions created to verify `whenPaused` and `whenNotPaused` modifiers

### Properties

-  **Balances unchanged when paused.** When a contract is in a paused state, the token balance for a given user and token must not change. ([report](#))
-  **Transfer methods revert when paused.** When a contract is in a paused state, transfer methods must revert. ([report](#))
-  **Pause method pauses contract.** When a contract is in an unpaused state, calling `pause()` must pause. ([report](#))
-  **Unpause method unpauses contract.** When a contract is in a paused state, calling `unpause()` must unpause. ([report](#))
-  **Cannot pause while paused.** When a contract is in a paused state, calling `pause()` must revert. ([report](#))
-  **Cannot unpause while unpaused.** When a contract is in an unpaused state, calling `unpause()` must revert. ([report](#))
-  **`whenNotPaused` modifier causes revert if paused.** When a contract is in a paused state, functions with the `whenNotPaused` modifier must revert. ([report](#))



-  `whenPaused` **modifier causes revert if unpaused.** When a contract is in an unpaused state, functions with the `whenPaused` modifier must revert. ([report](#))

## 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 harnessing 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

-  **Token total supply independence.** Given two different token ids, if total supply for one changes, then total supply for other must not. ([report](#))
-  **Total supply is sum of balances.** The sum of the balances over all users must equal the total supply for a given token. ([report](#))
-  **Balance of zero address is zero.** The balance of a token for the zero address must be zero. ([report](#))
-  **Held tokens should exist.** If a user has a token, then the token should exist. ([report](#))

## 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

None

### 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 overridden 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 havoc 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

- ❌ **Quorum reached effect.** If a proposal has reached quorum then the proposal snapshot (start `block.number` ) must be non-zero ([report](#))
- ❌ **Proposal not created effects.** A non-existent proposal must meet the definition of one. ([report](#))
- ❌ **Proposal in one state.** A created proposal must be in state `deadlineExtendable` or `deadlineExtended` . ([report](#))

### first set of rules

The rules [deadlineChangeEffects](#) and [deadlineCantBeUnextended](#) are assumed in rule [canExtendDeadlineOnce](#) , so we prove them first.

- ✅ **Deadline change effects.** If deadline increases then we are in `deadlineExtended` state and `castVote` was called. ([report](#))
- ❌ **Deadline can't be unextended.** A proposal can't leave `deadlineExtended` state. ([report](#))
- ✅ **Can extend deadline once.** A proposal's deadline can't change in `deadlineExtended` state. ([report](#))

### second set of rules

The main rule in this section is [the deadline can only be extended if quorum reached with  \$\leq\$  timeOfExtension left to vote](#) The other rules of this section are assumed in the proof, so we prove them first.

- ✅ **Has voted correlation nonzero.** A change in `hasVoted` must be correlated with an increasing of the vote supports, i.e. casting a vote increases the total number of votes. ([report](#))
- ✅ **Against votes don't count.** An against vote does not make a proposal reach quorum. ([report](#))
- ✅ **Extended deadline value set if quorum reached.** `extendedDeadlineField` is set if and only if `_castVote` is called and quorum is reached. ([report](#))
- ✅ **Deadline never reduced.** Deadline can never be reduced. ([report](#))

# 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.

(✔) Bug 1: 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() +
lateQuorumVoteExtension();
+    uint64 extendedDeadlineValue = block.number.toUint64();
```

(✔) Bug 2: 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");
```

(✔) Bug 3: 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()) {
```

(✔) Bug 4: 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()), "ERC1155: caller is not owner nor approved");
+    // require(account == _msgSender() || isApprovedForAll(account,
_msgSender()), "ERC1155: caller is not owner nor approved"// );
```

(✔) Bug 5: 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 transferred (or minted):

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

(✔) Bug 6: 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];
```

(✔) Bug 7: mutate `burn` function in `ERC1155Burnable.sol` : **catching rule(s)**:  
`burnAmountProportionalToBalanceReduction` ,  
`sequentialBurnsEquivalentToSingleBurnOfSum` , `singleTokenBurnBurnBatchEquivalence` ,  
`multipleTokenBurnBurnBatchEquivalence` [Tool Output] : This change will cause `msg.sender`'s token balance to decrease by `value` instead of the appropriate account's balance:

```
-   _burn(account, id, value);
+   _burn(msg.sender, id, value);
```

---