Formal Verification of Balancer (TimelockAuthorizer)

Summary

This document describes the specification and the verification of Balancer's contract TimelockAuthorizer using the Certora Prover. The latest commit reviewed and ran through the Certora Prover was 36465b52.

The scope of our verification was the following contracts:

- TimelockAuthorizer.sol
- TimelockAuthorizerManagement.sol

The Certora Prover proved that the implementation of the contract above 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 focus on validating the correct behavior of the contracts above as described by the Balancer team and the contract documentation. The rules verify valid states of the system, proper transitions between states, method integrity, and high-level properties (which often describe more than one element of the system and can even be cross-system).

Main Issues Discovered

Severity: Medium

Issue	**(M1) Unexpected root claims**
Description	Executing a previously scheduled `scheduleRootChange` can result in an unexpected root claim. Here are two exemplary plausible scenarios that can lead to this issue: **Erroneous Double Scheduling:** The root can mistakenly schedule a root change twice to Alice. Alice then claims root powers. If Alice transfers root permissions to Bob at any point in the future, she could still claim herself as root by executing the other scheduled root transfer to herself. **Intentional Backdoor:** A root can schedule a root transfer to itself without executing it. At any later time, the original root can execute this root change and revoke root privileges from the current root. The security assumption is that the root is not malicious. It does not prevent this attack unless it is assumed that the current root and all previous roots in the system's history are not and will never be malicious. Additionally, the first scenario above does not necessarily stem from malicious intentions but rather from mistrust between the roots or execution mistakes. For example, if the system was deployed and initialized with an EOA root address, and later the root privileges were claimed by a DAO, we may not trust the EOA anymore, as it has no time delays or transparency, and it can be compromised.

Response	Balancer is aware of this issue. Similarly, it can be used to avoid delays if a scheduled action is already in place. However, it is difficult to keep track of the scheduled executions for specific actions and update them if the corresponding execution is executed or canceled. Balancer's team has deemed the added complexity of such a process to be not worthwhile.

Severity: Low

Issue	**(L1) Non-root users could get roots permissions**
Description	Somebody could get root permissions if there was a mistake in granting arguments and then make another malicious user an executor for an action they weren't supposed to execute.
Response	The permission system was rewritten. Under the new system, there is no non-root global granter.

Severity: Low

Issue	**(L2) Can create a canceler for a non existing `scheduledExecutionId`**
Description	The `addCanceler()` function does not prevent the root from mistakenly adding a canceler to a non-existent execution. This is because there are no checks that the `scheduledExecutionId` exists and because `_isCanceler` is a mapping, no out-of-bounds exception will occur. If a malicious user cancels future executions with that `scheduledExecutionId`, it can cause delays for sensitive tasks that must be rescheduled.

Response	The `scheduledExecutionId` for an action cannot be predicted because it depends on others scheduling actions. Creating a canceler for an unknown id is unreasonable and, therefore, always a mistake. We updated the code to only allow creating cancelers for existing executions that have not been canceled or executed, or global cancelers.

Severity: Low

Issue	**(L3) The system can be stuck with address(0) root**
Description	If the default address value, `address(0)`, is used as both the current and pending root when deploying the contract, the root will be invalid. This error cannot be recovered because only the root can set a new pending root.
Response	This is not an issue because the migrator forces the root to interact when deploying the contract.

Severity: Low

Issue	**(L4) Nobody with root permissions**
Description	If `_pendingRoot` and `_root` were equal, then after claiming root, there would be no one with root permissions.
Response	The issue was resolved after a rewrite of the permission system.

Severity: Informational

Issue	**Inconsistency in adding and removing permissions**
Description	There is an option to grant specific permissions to someone who already has global permissions for a given actionId and account. But there isn't the opposite option, to revoke a specific permission from someone if they have global permission.
Response	We disallowed granting specific permission to someone who already has global permissions.

Severity: Informational

Issue	**Execution delay can be smaller than MIN_DELAY**
Description	The check for the `MIN_DELAY` is not on the parameter that the `data` contains, so it was possible to set the delay below the `MIN_DELAY`.
Response	Balancer renamed the variable to `_MINIMUM_CHANGE_DELAY_EXECUTION_DELAY` to convey its meaning better.

Disclaimer

The Certora Prover takes a contract and a specification as an input and formally proves that the contract satisfies the specification in all possible 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.

Summary of formal verification

Notations

✔️ indicates the rule is formally verified on the latest reviewed commit.

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

✍️ indicates the rule is not yet formally specified.

⏱️ 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).

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.

Assumptions and simplifications for verification

- We unroll loops. Violations that require a loop to execute more than 8 times will not be detected.
- `_scheduledExecutions` array length was limited to `max_uint / 4` due to tool limitations.

Verification of TimelockAuthorizer.sol

Summary

`TimelockAuthorizer` combines a classic `Timelock` contract, which handles execution delay, and an `Authorizer` contract which handles authorization.

Properties

(✓) *`notGreaterThanMax`*

Each `_delaysPerActionId` is less than or equal to `MAX_DELAY`.

(✓) *`notFarFuture`*

Any `executableAt` from `_scheduledExecutions` is not further in the future than `MAX_DELAY`.

(✓) *`arrayHierarchy`*

For any two scheduled executions with the same action ID, the action with the smaller index should not be scheduled after the one with the larger index.

(✓) *`oneOfThree`*

No `scheduledExecution` can be executed and canceled simultaneously. It can be:

- neither canceled nor executed;
- canceled, not executed;
- executed, not canceled.

(✓) *`immutableExecuteAt / Where / Protected`*

`executableAt` / `where` / `protected` are immutable.

- `data` cannot be checked due to current tool limitations.

(✓) ***`onlyOneExecuteOrCancelCanChangeAtTime`***

Only one `executed`/`canceled` flag can be changed by a single transaction.

(✓) ***`onlyExecuteAndCancelCanChangeTheirFlags`***

The flag `executed`/`canceled` can only be changed by the appropriate functions.

(✓) ***`canExecuteAndExecuteUnion`***

An `execute()` reverts if `canExecute()` reverts/returns false.

(✓) ***`executedCanceledForever`***

If an execution reaches one of its final states (`executed` or `canceled`), it should remain in that state forever.

(✓) ***`almightyGlobal`***

An account with `GLOBAL_CANCELER_SCHEDULED_EXECUTION_ID()` permissions can cancel any scheduled execution.

(✗) ***`onlyOneCanceler`***

When an action is scheduled, it has only one non-global canceler.

- Failing because of issue L2.

(✓) ***`scheduledExecutionsArrayIsNeverShortened`***

`_scheduledExecutions` array cannot decrease in length.

(✓) ***`whoCanCancelExecution`***

Checking access privilege for canceling an already scheduled execution.

(✓) ***`schExExecutionCheck`***

If an action was executed, it was unprotected, or a user had the necessary permission. If the action is not executed after a function call, it wasn't executed before it.

(✓) ***`schExeNotExecutedBeforeTime`***

A scheduled execution cannot be executed before the `executableAt` time.

(✓) ***`rootChangesOnlyWithClaimRoot`***

If the function changed the root, the sender was `pendingRoot`; if the function changed the root, the new root is old `pendingRoot`; if the function `f` changed the root, then `f` must be `claimRoot`.

(✓) ***`scheduledExecutionCanBeCanceledOnlyOnce`***

`ScheduledExecution` that has already been canceled or executed cannot be canceled again.

(✓) ***`scheduledExecutionCanBeExecutedOnlyOnce`***

`ScheduledExecution` that has already been canceled or executed cannot be executed again.

(✓) ***`scheduleDelayChangeHasProperDelay`***

When a change of delay is scheduled, the created `ScheduledExecution` has the appropriate `executableAt` (waiting time to be executed). In this rule we assume that `e.block.timestamp + execution delay < max_uint256`.

(✓) ***scheduleRootChangeCreatesSE***

`ScheduleRootChange` creates a newly scheduled execution which doesn't change the current or pending root.

(✓) ***hasPermissionIfIsGrantedOnTarget***

When `isPermissionGrantedOnTarget(id, account, where)` returns `true`, then `hasPermission(id, account, where)` also returns `true`

(✓) ***rootNotZero***

`root` cannot become the zero address.

(✓) ***onlyOneRoleChangeAtATimeForNonClaimRoot***

Any function other than `claimRoot` does not change more than one of the following roles: Granter, Revoker, or Canceler. At the same time, when a change of one of these roles happens by executing a function other than `claimRoot`, it happens for one account only.

(✓) ***pendingRootChangesOnlyWithSetPendingRootOrClaimRoot***

Only two functions can change the pending root: `claimRoot` and `setPendingRoot`, and two different addresses must execute them. The sender must be either the `_pendingRoot` or the `_executionHelper`.

(✓) ***`cannotBecomeExecutorForAlreadyScheduledExecution`***

A scheduled execution can only be executed by those supplied as executors when scheduling it.

(✓) ***`whoCanExecute`***

A protected scheduled execution can be executed only by an executor.

(✓) ***`whatCanBeExecuted`***

An execution can be executed (`canExecute`) only when it has yet to be canceled or executed, and its `executableAt` is not greater than the current timestamp.

(✓) ***`isExecutorChangedBySchedulerInNonScheduleFunction`***

`isExecutor` can only be changed by `_executionHelper`, and it can only happen by executing a non-schedule[1] function.

(✓) ***`isGranterChangesOnlyWithAddOrRemoveGranter`***

`isGranter` can be changed only by `_root` or `_pendingRoot`; A granter can be added only by calling `addGranter` or `claimRoot`; A granter can only be removed by calling `removeGranter` or `claimRoot`.

(✓) ***`isRevokerChangesOnlyWithAddOrRemoveRevoker`***

`isRevoker` can be changed only by `_root` or `_pendingRoot`; A revoker can be added only by calling `addRevoker` or `claimRoot`; A revoker can only be removed by calling `removeRevoker` or `claimRoot`.

(✓) ***isCancelerChangesOnlyWithAddOrRemoveCanceler***

A canceler can be added only by calling `addCanceler`, `claimRoot`, `schedule`, `scheduleRevokePermission`, `scheduleGrantPermission`; A canceler can only be removed by calling `removeCanceler` or `claimRoot`.

(✓) ***delayChangesOnlyBySetDelay***

Delay can be changed only by calling `setDelay`, and the new delay is the parameter supplied to this function. Delay can be changed only by the `_executionHelper`.

(✓) ***scheduledExecutionsCanBeChangedOnlyByScheduleFunctions***

The `_scheduledExecutions` array only changes its length when one of the scheduled functions[1] is called.

(✓) ***grantDelaysCanBeChangedOnlyBySetGrantDelay***

The grant delay of an action (stored in `_grantDelays`) can be changed only by calling `setGrantDelay`, and the caller must be the `_executionHelper`.

(✓) ***revokeDelaysCanBeChangedOnlyBySetRevokeDelay***

The revoke delay of an action (stored in `_revokeDelays`) can be changed only by calling `setRevokeDelay`, and the caller must be the `_executionHelper`.

(✓) ***grantedPermissionsChangeOnlyByAllowedFunctions***

A permission (`_isPermissionGranted`, obtained from `isPermissionGrantedOnTarget`) can be removed only by calling `revokePermission` or `renouncePermission`. If `revokePermission` was called, the permission is removed only by the `_executionHelper` or the revoker (obtained from `isRevoker`). On the other hand, a permission can be added only by calling `grantPermission`, and only by the `_executionHelper` or the granter (obtained from `isGranter`).

Footnotes

1. By schedule functions we refer to the following set of functions: `schedule`, `scheduleRevokePermission`, `scheduleGrantPermission`, `scheduleRevokeDelayChange`, `scheduleGrantDelayChange`, `scheduleRootChange`, `scheduleDelayChange`.