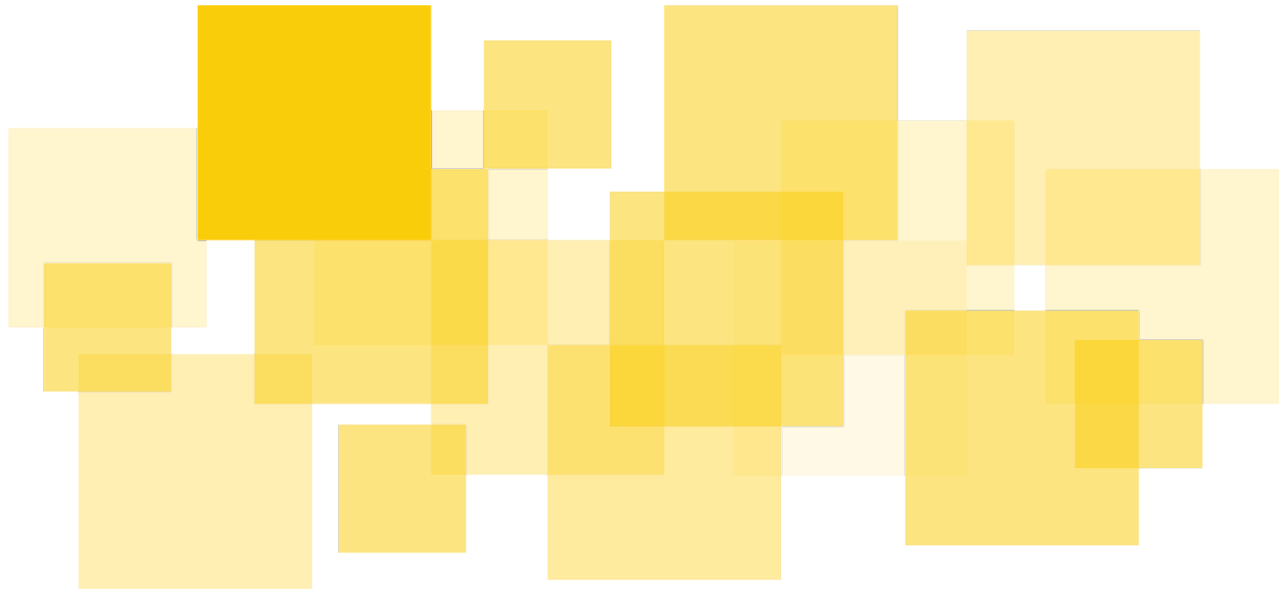




Security Audit Report

Dual Governance Formal Verification Ethereum

Delivered: February 13th, 2025



Prepared for Lido by





Table of Contents

- [Disclaimer](#)
- [Introduction](#)
- [Methodology](#)
- [Tests and Properties](#)



Disclaimer

Not addressed by client

This report does not constitute legal or investment advice. You understand and agree that this report relates to new and emerging technologies and that there are significant risks inherent in using such technologies that cannot be completely protected against. While this report has been prepared based on data and information that has been provided by you or is otherwise publicly available, there are likely additional unknown risks which otherwise exist. This report is also not comprehensive in scope, excluding a number of components critical to the correct operation of this system. This report is for informational purposes only and is provided on an "as-is" basis and you acknowledge and agree that you are making use of this report and the information contained herein at your own risk. The preparers of this report make no representations or warranties of any kind, either express or implied, regarding the information in or the use of this report and shall not be liable to you or any third parties for any acts or omissions undertaken by you or any third parties based on the information contained herein.



Introduction

Not addressed by client

Lido has engaged Runtime Verification to formally verify correctness and safety properties of the smart contracts that comprise the Lido Dual Governance protocol. The formal verification was conducted using Runtime Verification's symbolic execution tool Kontrol. Kontrol's symbolic execution is performed over the compiled EVM bytecode of the smart contracts, thus guaranteeing that it can catch errors introduced by the compiler or identify corner cases that may not be evident on source code inspection.

The formal verification of the Dual Governance contracts was performed over the following commit: <https://github.com/lidofinance/dual-governance/commit/3e0f1ae5740ef8410e928f6cc106e3a5f45a5a75>. The contracts were compiled using Solidity v0.8.26, and the Kontrol version used was v1.0.118.

All Kontrol tests described in this report have passed in the above commit. The full formal verification results, including running time information for the symbolic execution of each test, can be found in the accompanying [KaaS report](#), generated by Runtime Verification's [KaaS](#) platform for cloud-based symbolic execution.



Methodology

Not addressed by client

Formal verification of the Lido Dual Governance protocol was conducted using Kontrol, Runtime Verification's tool for symbolic execution of Solidity smart contracts.

Kontrol's functionality is integrated with the Forge testing framework, a component of the Foundry toolkit for smart contract development. Kontrol tests are written in the same way as Forge parameterized tests, but instead of using fuzzing to run each test on randomly-generated inputs, Kontrol executes the tests symbolically. This means that test parameters are interpreted as symbolic variables with no concrete value associated with them. Kontrol then uses symbolic execution to step through the test exploring all possible execution paths, employing logical and mathematical reasoning to determine if the assertions are satisfied on every path.

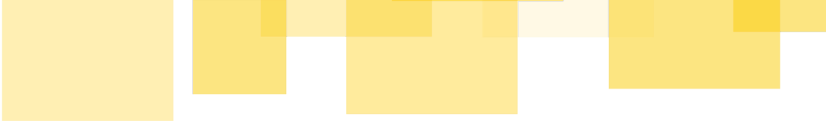
A typical Kontrol test often takes the following form:

1. Initialize the storage of the relevant contracts with symbolic variables with an unspecified value, in order to represent that the contracts can be in an arbitrary state. This can be done using Kontrol-specific cheatcodes such as `kevm.symbolicStorage` and `kevm.freshUInt`.
2. Use the `vm.assume` cheatcode to impose restrictions on the values to be considered for the symbolic variables (test inputs and symbolic storage variables).
3. If the function being tested can only be called by a specific address, use the `vm.prank` cheatcode to impersonate that address.
4. Call the function being verified.
5. Use `assert` s to check that certain properties hold after the function returns.

Other tests can also use cheatcodes to verify other properties, such as that a function reverts under certain conditions, or that external calls are only made to certain contracts.

In order to make symbolic execution of the tests practical, it's occasionally necessary to make simplifying assumptions. The following are examples that apply to many of the tests below:

- Rather than a fully symbolic address, several of the property tests below use an arbitrary concrete address to represent a random user of the protocol. This is done to make symbolic execution easier, since reasoning about symbolic values is harder than reasoning about concrete ones. However, since the exact value of an address is usually not important



for contract execution, we can generally assume that results for one arbitrary address can generalize to other arbitrary addresses.

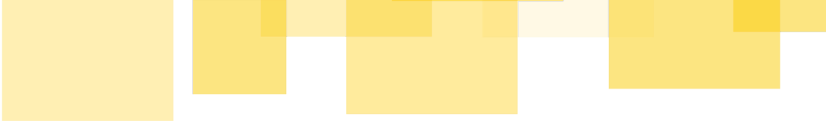
- Most of the symbolic variables representing ETH amounts are assumed to have a value under `2 ** 96`. This is done so that Kontrol can ensure that there won't be an overflow when multiple of these variables are added and multiplied. Similarly, many symbolic variables representing timestamps are assumed to be under `2 ** 35`.

Other simplifying assumptions specific to certain tests are described in the "Tests and Properties" section.

Kontrol Project Structure

In addition to the test files, a number of auxiliary files are included in the repository:

- **kontrol.toml file:** This file works similarly to the `foundry.toml` file, specifying the configurations for Kontrol to use when running the tests. See the [Kontrol documentation](#) for the meaning of different options.
- **kontrol-cheatcodes library:** This submodule, included in the `lib` folder, is necessary in order to import the signature of the Kontrol-specific cheatcodes to the tests.
- **model folder:** This folder contains simplified models of external contracts that the Dual Governance protocol interacts with, in particular the `StETH` token and the `WithdrawalQueue`. These models aim to reproduce the behavior of these contracts in enough detail to simulate their interactions with the Dual Governance contracts, while abstracting away some details that are not necessary to complete verification of the Dual Governance functions.
- **...StorageSetup.sol files:** These files include functions to initialize the storage of important contracts, such as `DualGovernance` and `Escrow`, with symbolic variables. These functions are called by the `setUp` function implemented in the `DualGovernanceSetup` file. By initializing storage variables with symbolic values before each test, we can execute the tests as if the contracts are in an arbitrary state.
- **storage folder:** This folder contains a shell script `storage_setup.sh` that automatically generates a set of Solidity files of the form `...StorageConstants.sol` for different Dual Governance contracts. Each file contains constants indicating the storage slots and offsets for the storage variables of the corresponding contract. These constants are imported by the `...StorageSetup.sol` files, which use them to determine the storage location of each



variable when initializing them symbolically. Therefore, if the storage layout of a contract changes, the `storage_setup.sh` script must be re-run in order to update the constants with the correct storage locations.

- **`lido-lemmas.k` file:** This file contains auxiliary lemmas written in the K specification language, which are added to help Kontrol simplify challenging expressions.

Reproducibility Instructions

The results of the tests can be reproduced using the Kontrol tool by running the following commands from the top-level directory of the repository.

First, if storage locations in the contracts have changed, run the `storage_setup.sh` script described above in order to update the constants for each storage slot:

```
bash test/kontrol/storage/storage_setup.sh
```

Then, build the project and run the proofs symbolically using Kontrol:

```
export FOUNDRY_PROFILE=kprove # set the Foundry profile to compile the Kontrol tests
kontrol build
kontrol prove
```

Be aware that the execution time of the entire test suite may take over 24h. See the [KaaS report](#) for more detailed timing information for each test, but note that running time may vary depending on the machine.



Tests and Properties

Not addressed by client

The following are descriptions of each test contract and the tests that they contain, briefly summarizing the properties being verified by each test.

`EscrowAccounting.t.sol`

This file consists of tests checking that the accounting performed by different functions of the `Escrow` contract are correct, including that funds are correctly transferred. It also checks certain consistency conditions of the `Escrow` contract are preserved by these functions, as specified in the `EscrowInvariants` file.

`testRageQuitSupport`

Tests that the rage quit support returned by `Escrow.getRageQuitSupport` is calculated correctly based on the funds locked in the escrow.

`testEscrowInvariantsHoldInitially`

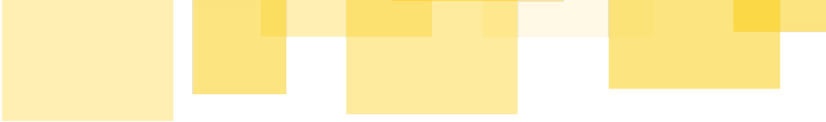
Tests that the consistency conditions of the `Escrow` contract (see the `EscrowInvariants` file) hold when a signalling escrow is created.

We check this by cloning a new escrow from the master copy (in the same way that the `DualGovernance` contract does when creating a new signalling escrow), then checking that the invariants hold in this initial state.

`testRequestNextWithdrawalsBatch`

Tests that after calling `Escrow.requestNextWithdrawalsBatch` on the `RageQuitEscrow`, the number of stETH shares of the escrow is decreased by the amount corresponding to the withdrawal request (as long as there were enough shares to submit the request in the first place). Additionally, the `WithdrawalBatchesQueue` is closed if and only if the remaining stETH in the escrow is under the minimum withdrawable amount. For simplicity, the batch size of the request is only a single unstETH.

`testClaimNextWithdrawalsBatch`



Tests that after calling `Escrow.claimNextWithdrawalsBatch` on the `RageQuitEscrow`, the amount of ETH received by the escrow from claiming the request is accounted in the `claimedETH` field of `AssetsAccounting`. For simplicity, the test only claims a single unstETH.

EscrowLockUnlock.t.sol

This file consists of tests for the `Escrow.lockStETH` and `Escrow.unlockStETH` functions, checking their postconditions and that they preserve the invariants of the `SignallingEscrow` (as specified in the `EscrowInvariants` file).

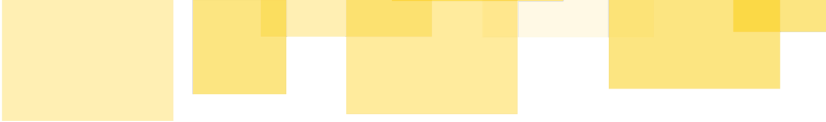
Because these two functions call `DualGovernance.activateNextState` twice each, these tests are particularly time-consuming and resource-intensive. Therefore, in practice we split them into 5 smaller tests each, where each of the smaller tests executes the base test starting from a different `DualGovernance` state.

For the same reasons above, these tests also use the `kevm.forgetBranch` cheatcode, which causes Kontrol to forget certain complex formulas that are generated during the symbolic execution but are not actually necessary for verification, such as the precise calculation of the rage-quit support. If not discarded, these formulas can significantly increase the processing time of the logic solver used by Kontrol's backend.

testLockStEth

Tests that after a user calls `Escrow.lockStETH` in the `SignallingEscrow`, the user's and escrow's stETH balance change as expected, and the escrow's internal accounting is updated correctly. If `amount` is the value passed to `lockStETH` and `amountInShares` is the corresponding number of stETH shares for that amount, we check that

- The user's stETH shares decrease by `amountInShares`.
- The escrow's stETH shares increase by `amountInShares`.
- The number of total locked shares in `AssetsAccounting` increases by `amountInShares`.
- The number of locked shares for that user in `AssetsAccounting` increases by `amountInShares`.
- The user's `lastAssetsLockedTimestamp` in `AssetsAccounting` is set to the current timestamp.
- The user's stETH balance decreases by at most `amount` and at least `amount - errorTerm`, where `errorTerm = StETH.getPooledEthByShares(1) + 1` is a



small rounding error introduced by the conversion from and to shares.

- The escrow's stETH balance increases by at most `amount` and at least `amount - errorTerm`.

`testUnlockStEth`

Tests that after a user calls `Escrow.unlockStETH` in the `SignallingEscrow`, the user's and escrow's stETH balance change as expected, and the escrow's internal accounting is updated correctly. If `shares` is the number of stETH shares that the user had previously locked in the escrow, and `amount` is the corresponding value in stETH, we check that

- The user's stETH shares increase by `shares`.
- The escrow's stETH shares decrease by `shares`.
- The number of total locked shares in `AssetsAccounting` decreases by `shares`.
- The number of locked shares for that user in `AssetsAccounting` is updated to 0.
- The user's `lastAssetsLockedTimestamp` in `AssetsAccounting` remains the same.
- The user's stETH balance increases by at least `amount` and at most `amount + 1`, due to rounding error introduced by the conversion from and to shares.
- The escrow's stETH balance decreases by at least `amount` and at most `amount + 1`.

`EscrowOperations.t.sol`

This file consists of tests checking various preconditions of operations from the `Escrow` contract, guaranteeing that the operations revert when these preconditions are violated.

`testCannotUnlockBeforeMinLockTime`

Tests that `Escrow.unlockStETH` reverts if it's called before the min assets lock duration has passed since the last time the caller locked assets in the escrow.

Since `unlockStETH` calls `DualGovernance.activateNextState`, the test needs to include an assumption that the `DualGovernance` will not transition into the `RageQuit` state under the current conditions. This transition would cause the `SignallingEscrow` to turn into a `RageQuitEscrow`, which is covered in the next test below.

`testCanotLockUnlockInRageQuitEscrowState`



Tests that both `Escrow.lockStETH` and `Escrow.unlockStETH` revert if the escrow enters the `RageQuitEscrow` state.

`testCannotWithdrawBeforeEthClaimTimeLockElapsed`

Tests that when `Escrow.withdrawETH` is called on the `RageQuitEscrow` after the rage quit extension period has passed, then a) if the withdrawals delay period has not passed, the function reverts, or b) if it has, then it succeeds and withdraws all of the user's shares.

`ProposalOperations.t.sol`

This file consists of tests checking preconditions for submitting and scheduling proposals based on the `DualGovernance` state, guaranteeing that an operation will revert if it is not allowed in the current state.

`testCannotProposeInInvalidState`

Tests that `DualGovernance.submitProposal` reverts if the effective state is either `VetoSignallingDeactivation` or `VetoCooldown`.

`testCannotScheduleInInvalidStates`

Tests that `DualGovernance.scheduleProposal` reverts if the effective state is either `VetoSignalling`, `VetoSignallingDeactivation` or `RageQuit`.

`testCannotScheduleSubmissionAfterLastVetoSignalling`

Tests that, when the effective state is `VetoCooldown`, `DualGovernance.scheduleProposal` reverts if the proposal being scheduled was submitted after the last time the `VetoSignalling` state was entered.

`testOnlyProposalsCancellerCanCancelProposals`

Tests that `DualGovernance.cancelAllPendingProposals` reverts if it is called by an address other than the proposals canceller.

`RageQuit.t.sol`

This file consists of a single test, `testRageQuitDuration`, which checks that if the `DualGovernance` is in the `RageQuit` state, it will remain in the `RageQuit` state until the `RAGE_QUIT_EXTENSION_PERIOD_DURATION` has passed.



TimeLockInvariants.t.sol

This file consists of tests checking correctness properties of various functions of the `EmergencyProtectedTimeLock` contract. These include that these functions produce the correct state changes and that they have the appropriate access protections.

For simplicity, tests for functions that deal with proposal submission or execution use a proposal with a single call. When the proposal needs to be concretely executed or loaded into memory, we use a simple example proposal that sets a flag in an external contract.

`_checkStateRemainsUnchanged`

Besides the properties exclusive to each test, all tests in this file also include the modifier `_checkStateRemainsUnchanged`, which checks that the function being tested doesn't modify any fields of the contract that it's not supposed to. This modifier does the following:

1. Saves the state of the `TimeLockState.Context` and `EmergencyProtection.Context` records before the test is executed.
2. Executes the test.
3. Saves the state of the `TimeLockState.Context` and `EmergencyProtection.Context` records after the test is executed.
4. For each field of the two records, checks if the function selector of the tested function corresponds to one that should modify the field. If not, compares the contents of the field before and after the test to check that they are the same.

`testSubmit`

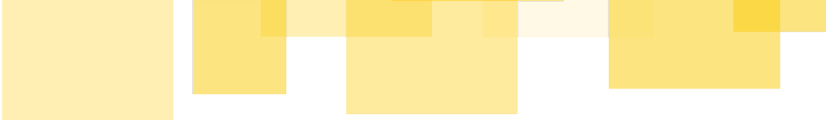
Tests that when `EmergencyProtectedTimeLock.submit` is called by the governance address, it succeeds and the status of the new proposal ID is set as `Submitted`.

`testSubmitRevert`

Tests that `EmergencyProtectedTimeLock.submit` reverts when called by any address other than the governance address.

`testSchedule`

Tests that when `EmergencyProtectedTimeLock.schedule` is called by the governance address after the post-submission delay for the proposal has passed, it succeeds and the status of the



proposal is set to `Scheduled` .

`testScheduleRevert`

Tests that `EmergencyProtectedTimelock.schedule` reverts when called by any address other than the governance address.

`testScheduleDelayHasNotPassedRevert`

Tests that `EmergencyProtectedTimelock.schedule` reverts when the after-submit delay for the proposal has not passed since it was submitted.

`testExecute`

Tests that when `EmergencyProtectedTimelock.execute` is called by the governance address after the post-schedule delay for the proposal has passed, it succeeds and the following are true:

1. The status of the proposal is set to `Executed` .
2. The calls are performed to the target contract.
3. Besides the call from the `EmergencyProtectedTimelock` to the `Executor` and from the `Executor` to the target contract, no other calls are executed.

`testExecuteNonScheduledRevert`

Tests that `EmergencyProtectedTimelock.execute` reverts when the proposal has not been scheduled.

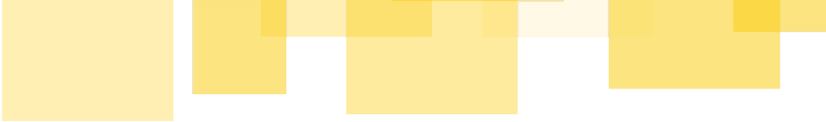
`testExecuteExecutedRevert`

Tests that `EmergencyProtectedTimelock.execute` reverts when the proposal has already been executed.

`testExecuteDelayHasNotPassedRevert`

Tests that `EmergencyProtectedTimelock.execute` reverts when the after-schedule delay for the proposal has not passed since it was scheduled.

`testCancelAllNonExecutedProposals`



Tests that when `EmergencyProtectedTimelock.cancelAllNonExecutedProposals` is called by the governance address, any proposal whose status was not `Executed` has its status set to `Cancelled`.

`testOnlyGovernanceCanCancelProposals`

Tests that `EmergencyProtectedTimelock.cancelAllNonExecutedProposals` reverts if it is called by any address except the governance address.

`testCancelledProposalsCannotBeScheduled`

Tests that `EmergencyProtectedTimelock.schedule` reverts if it is called with a proposal ID that corresponds to a proposal whose status is `Cancelled`.

`testCancelledProposalsCannotBeExecuted`

Tests that `EmergencyProtectedTimelock.execute` reverts if it is called with a proposal ID that corresponds to a proposal whose status is `Cancelled`.

`testCancelledProposalsCannotBeEmergencyExecuted`

Tests that `EmergencyProtectedTimelock.emergencyExecute` reverts if it is called with a proposal ID that corresponds to a proposal whose status is `Cancelled`.

`testSetGovernance`

Tests that when `EmergencyProtectedTimelock.setGovernance` is called by the admin executor, it updates the governance address in `EmergencyProtectedTimelock` to the provided address.

`testSetAfterSubmitDelay`

Tests that when `EmergencyProtectedTimelock.setAfterSubmitDelay` is called by the admin executor, it updates the after-submit delay in `EmergencyProtectedTimelock` to the provided value.

`testSetAfterScheduleDelay`

Tests that when `EmergencyProtectedTimelock.setAfterScheduleDelay` is called by the admin executor, it updates the after-schedule delay in `EmergencyProtectedTimelock` to the provided value.



testTransferExecutorOwnership

Tests that when `EmergencyProtectedTimelock.transferExecutorOwnership` is called by the admin executor for an executor owned by the `EmergencyProtectedTimelock`, it updates the owner of the given executor to the provided address.

testSetEmergencyProtectionActivationCommittee

Tests that when `EmergencyProtectedTimelock.setEmergencyProtectionActivationCommittee` is called by the admin executor, it updates the address of the emergency activation committee in `EmergencyProtectedTimelock` to the provided address.

testSetEmergencyProtectionExecutionCommittee

Tests that when `EmergencyProtectedTimelock.setEmergencyProtectionExecutionCommittee` is called by the admin executor, it updates the address of the emergency execution committee in `EmergencyProtectedTimelock` to the provided address.

testSetEmergencyProtectionEndDate

Tests that when `EmergencyProtectedTimelock.setEmergencyProtectionEndDate` is called by the admin executor, it updates the emergency-protection end date in `EmergencyProtectedTimelock` to the provided value.

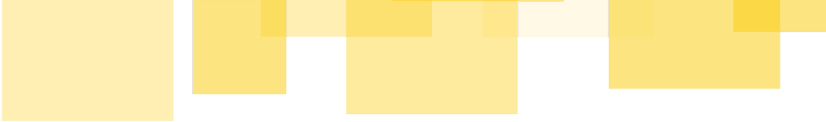
testSetEmergencyModeDuration

Tests that when `EmergencyProtectedTimelock.setEmergencyModeDuration` is called by the admin executor, it updates the emergency-mode duration in `EmergencyProtectedTimelock` to the provided value.

testSetEmergencyGovernance

Tests that when `EmergencyProtectedTimelock.setEmergencyGovernance` is called by the admin executor, it updates the emergency-governance address in `EmergencyProtectedTimelock` to the provided address.

testActivateEmergencyMode



Tests that when `EmergencyProtectedTimelock.activateEmergencyMode` is called by the address of the emergency activation committee and the emergency mode is not already active, the `EmergencyProtectedTimelock` enters the emergency mode.

`testActivateEmergencyModeRevert`

Tests that `EmergencyProtectedTimelock.activateEmergencyMode` reverts when called by any address other than the emergency activation committee.

`testActivateEmergencyModeInEmergencyRevert`

Tests that `EmergencyProtectedTimelock.activateEmergencyMode` reverts when the `EmergencyProtectedTimelock` is already in emergency mode.

`testActivateEmergencyAfterEndDateRevert`

Tests that `EmergencyProtectedTimelock.activateEmergencyMode` reverts when the emergency-protection end date has already passed.

`testEmergencyExecute`

Tests that when `EmergencyProtectedTimelock.emergencyExecute` is called by the emergency execution committee while the emergency mode is active, the calls are performed to the target contract and the status of the proposal is set to `Executed`.

`testEmergencyExecuteNonScheduledRevert`

Tests that `EmergencyProtectedTimelock.emergencyExecute` reverts when the proposal has not been scheduled.

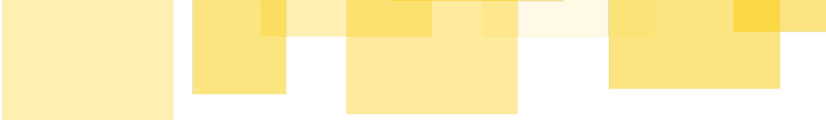
`testEmergencyExecuteExecutedRevert`

Tests that `EmergencyProtectedTimelock.emergencyExecute` reverts when the proposal has already been executed.

`testEmergencyExecuteNormalModeRevert`

Tests that `EmergencyProtectedTimelock.emergencyExecute` reverts when the `EmergencyProtectedTimelock` is not in emergency mode.

`testEmergencyExecuteRevert`



Tests that `EmergencyProtectedTimelock.emergencyExecute` reverts when called by an address other than the emergency execution committee.

`testDeactivateEmergencyMode`

Tests that when `EmergencyProtectedTimelock.deactivateEmergencyMode` is called, emergency mode is deactivated and all the following variables are zeroed out:

- `emergencyActivationCommittee`
- `emergencyExecutionCommittee`
- `emergencyModeDuration`
- `emergencyModeEndsAfter`
- `emergencyProtectionEndsAfter`

Additionally, any proposal whose status was not `Executed` has its status set to `Cancelled`.

`testDeactivateEmergencyModeNormalModeRevert`

Tests that `EmergencyProtectedTimelock.deactivateEmergencyMode` reverts when the `EmergencyProtectedTimelock` is not in emergency mode.

`testDeactivateEmergencyModeRevert`

Tests that `EmergencyProtectedTimelock.deactivateEmergencyMode` reverts when called by an address other than the admin executor.

`testEmergencyReset`

Tests that `EmergencyProtectedTimelock.emergencyReset` deactivates emergency mode and cancels all non-executed proposals in the same way as `deactivateEmergencyMode`. Additionally, it also sets the governance address to the emergency governance.

`testEmergencyResetNormalModeRevert`

Tests that `EmergencyProtectedTimelock.emergencyReset` reverts when the `EmergencyProtectedTimelock` is not in emergency mode.

`testEmergencyResetRevert`

Tests that `EmergencyProtectedTimelock.emergencyReset` reverts when called by an address other than the emergency execution committee.



testSetAdminExecutor

Tests that when `EmergencyProtectedTimelock.setAdminExecutor` is called by the admin executor, it updates the address of the admin executor in `EmergencyProtectedTimelock` to the new address.

testSetGovernanceRevert

Tests that `EmergencyProtectedTimelock.setGovernance` reverts when called by an address other than the admin executor.

testSetAfterSubmitDelayRevert

Tests that `EmergencyProtectedTimelock.setAfterSubmitDelay` reverts when called by an address other than the admin executor.

testSetAfterScheduleDelayRevert

Tests that `EmergencyProtectedTimelock.setAfterScheduleDelay` reverts when called by an address other than the admin executor.

testTransferExecutorOwnershipRevert

Tests that `EmergencyProtectedTimelock.transferExecutorOwnership` reverts when called by an address other than the admin executor.

testSetAdminExecutorRevert

Tests that `EmergencyProtectedTimelock.setAdminExecutor` reverts when called by an address other than the admin executor.

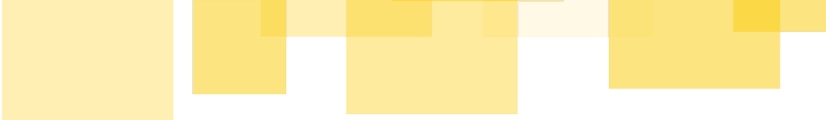
testSetEmergencyProtectionActivationCommitteeRevert

Tests that `EmergencyProtectedTimelock.setEmergencyProtectionActivationCommittee` reverts when called by an address other than the admin executor.

testSetEmergencyProtectionExecutionCommitteeRevert

Tests that `EmergencyProtectedTimelock.setEmergencyProtectionExecutionCommittee` reverts when called by an address other than the admin executor.

testSetEmergencyProtectionEndDateRevert



Tests that `EmergencyProtectedTimelock.setEmergencyProtectionEndDate` reverts when called by an address other than the admin executor.

`testSetEmergencyModeDurationRevert`

Tests that `EmergencyProtectedTimelock.setEmergencyModeDuration` reverts when called by an address other than the admin executor.

`testSetEmergencyGovernanceRevert`

Tests that `EmergencyProtectedTimelock.setEmergencyGovernance` reverts when called by an address other than the admin executor.

`TimeLockedGovernance`

This file contains tests for the `TimeLockedGovernance` contract, checking that certain functions can only be called by the governance address.

`testSubmitProposalRevert`

Tests that `TimeLockedGovernance.submitProposal` reverts when called by an address other than the governance address.

`testCancelAllPendingProposalsRevert`

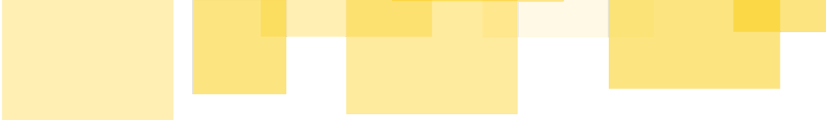
Tests that `TimeLockedGovernance.cancelAllPendingProposals` reverts when called by an address other than the governance address.

`VetoCooldown.t.sol`

This file consists of a single test, `testVetoCooldownDuration`, which checks that if the `DualGovernance` is in the `VetoCooldown` state, it will remain in the state until the `VETO_COOLDOWN_DURATION` has passed.

`VetoSignalling.t.sol`

This file consists of tests related to the duration of the `VetoSignalling` state. It defines certain invariants that must always hold when the `DualGovernance` is in the `VetoSignalling` state or its `VetoSignallingDeactivation` sub-state, then uses the tests to verify the following:

- 
1. The invariants hold when the `VetoSignalling` state is first entered (`testVetoSignallingInvariantsHoldInitially`).
 2. The invariants are preserved for as long as the `DualGovernance` remains in the `VetoSignalling` or `VetoSignallingDeactivation` states (`testVetoSignallingInvariantsArePreserved`).
 3. The invariants imply that the duration of the `VetoSignalling` state is bounded by the highest rage-quit support observed during the `VetoSignalling` period (`testDeactivationNotCancelled`).

The relevant invariants are the following:

- `_vetoSignallingTimesInvariant` : The `VetoSignalling` activation and reactivation times must have been in the past, and the reactivation time cannot be more than `VETO_SIGNALLING_MAX_DURATION` than the activation time.
- `_vetoSignallingRageQuitInvariant` : The highest rage-quit support observed since the `VetoSignalling` state was activated cannot be smaller than the current rage-quit support, and cannot be smaller than `FIRST_SEAL_RAGE_QUIT_SUPPORT`.
- `_vetoSignallingDeactivationInvariant` : The protocol is in the `VetoSignallingDeactivation` sub-state if and only if a) the `VetoSignalling` duration for the current rage-quit support has passed since activation, and b) `VETO_SIGNALLING_MIN_ACTIVE_DURATION` has passed since reactivation. Otherwise, it's in the parent state.
- `_vetoSignallingMaxDelayInvariant` : If the `VetoSignalling` duration for the highest rage-quit support observed so far has passed since activation, then the protocol is necessarily in the `VetoSignallingDeactivation` sub-state.

`testTransitionNormalToVetoSignalling`

Tests that if the `DualGovernance` is in the `Normal` state, it will transition to `VetoSignalling` if the total rage-quit support in the `SignallingEscrow` is greater than or equal to the `FIRST_SEAL_RAGE_QUIT_SUPPORT`.

`testVetoSignallingInvariantsHoldInitially`

Tests that when the `DualGovernance` first transitions into the `VetoSignalling` state (from the `Normal`, `VetoCooldown` or `RageQuit` states), the `VetoSignalling` invariants hold.

testVetoSignallingInvariantsArePreserved

Tests that `DualGovernance.activateNextState` preserves the `VetoSignalling` invariants. More specifically, it

1. Assumes that the current state is either `VetoSignalling` or `VetoSignallingDeactivation`.
2. Assumes that the `VetoSignalling` invariants hold.
3. Calls `activateNextState`.
4. Update the highest rage-quit support observed so far.
5. Checks that, if the new state is still either `VetoSignalling` or `VetoSignallingDeactivation`, the invariants still hold.

testDeactivationNotCancelled

Tests that `VetoSignallingDeactivation` cannot be exited back to `VetoSignalling` after the maximum `VetoSignalling` duration has passed. The "maximum `VetoSignalling` duration", in this case, means the `VetoSignalling` duration calculated for the highest rage-quit support observed since the `VetoSignalling` state was entered. In other words, if vetoers want to keep increasing the duration of the `VetoSignalling` state, they necessarily will need to keep increasing the funds locked in the `SignallingEscrow`.

More specifically, this test does the following:

1. Assumes that the current state is either `VetoSignalling` or `VetoSignallingDeactivation`.
2. Assumes that the `VetoSignalling` invariants hold.
3. Assumes that the maximum `VetoSignalling` duration has passed.
4. Checks that, given these conditions, the state must be `VetoSignallingDeactivation`.
5. Calls `DualGovernance.activateNextState`.
6. Checks that the resulting state is either `VetoSignallingDeactivation`, if the `VETO_SIGNALLING_DEACTIVATION_MAX_DURATION` hasn't passed, or `VetoCooldown`, if it has.

Along with the previous invariant tests, this implies that the duration of the `VetoSignalling` state is limited by the maximum amount of funds that the vetoers are able to amass at once, and this limit can't be unduly extended by locking and unlocking funds over time.