Hyperstable Security Review

Pashov Audit Group

Conducted by: Hals, Ch_301, merlinboii, solidit, Shubham, IvanFitro

March 19th 2025 - March 21rd 2025

Contents

1. About Pashov Audit Group

Pashov Audit Group consists of multiple teams of some of the best smart contract security researchers in the space. Having a combined reported security vulnerabilities count of over 1000, the group strives to create the absolute very best audit journey possible - although 100% security can never be guaranteed, we do guarantee the best efforts of our experienced researchers for your blockchain protocol. Check our previous work here or reach out on Twitter @pashovkrum.

2. Disclaimer

A smart contract security review can never verify the complete absence of vulnerabilities. This is a time, resource and expertise bound effort where we try to find as many vulnerabilities as possible. We can not guarantee 100% security after the review or even if the review will find any problems with your smart contracts. Subsequent security reviews, bug bounty programs and on-chain monitoring are strongly recommended.

3. Introduction

A time-boxed security review of the **hyperstable/contracts** repository was done by **Pashov Audit Group**, with a focus on the security aspects of the application's smart contracts implementation.

4. About Hyperstable

Hyperstable is an over-collateralized stablecoin designed to trade at one US Dollar, where users mint $USDH by depositing supported collateral. Governance and liquidity incentives are managed through voting and gauges, allowing vePEG holders to direct PEG emissions, earn rewards, and influence liquidity distribution.

5. Risk Classification

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

5.1. Impact

- High - leads to a significant material loss of assets in the protocol or significantly harms a group of users.
- Medium - only a small amount of funds can be lost (such as leakage of value) or a core functionality of the protocol is affected.
- Low - can lead to any kind of unexpected behavior with some of the protocol's functionalities that's not so critical.

5.2. Likelihood

- High - attack path is possible with reasonable assumptions that mimic on-chain conditions, and the cost of the attack is relatively low compared to the amount of funds that can be stolen or lost.
- Medium - only a conditionally incentivized attack vector, but still relatively likely.
- Low - has too many or too unlikely assumptions or requires a significant stake by the attacker with little or no incentive.

5.3. Action required for severity levels

- Critical - Must fix as soon as possible (if already deployed)
- High - Must fix (before deployment if not already deployed)
- Medium - Should fix
- Low - Could fix

6. Security Assessment Summary

review commit hash - 4f650122d3927fd45015c40ad58172508b148876

fixes review commit hash - f2cfa5cb9db05b1e30b048f802c2cf3a62c7878a

Scope

The following smart contracts were in scope of the audit:

- `DebtToken`
- `InterestRateStrategyV1`
- `LiquidationBuffer`
- `LiquidationManager`
- `PositionManager`
- `Vault`
- `BribeFactory`
- `EmissionScheduler`
- `ExternalBribe`
- `Gauge`
- `GaugeFactory`
- `InternalBribe`
- `Minter`
- `Peg`
- `PegAirdrop`
- `RewardsDistributor`
- `VeArtProxy`
- `Voter`
- `vePeg`

7. Executive Summary

Over the course of the security review, Hals, Ch_301, merlinboii, solidit, Shubham, IvanFitro engaged with Hyperstable to review Hyperstable. In this period of time a total of **14** issues were uncovered.

Protocol Summary

Protocol Name	Hyperstable
Repository	https://github.com/hyperstable/contracts
Date	March 19th 2025 - March 21rd 2025
Protocol Type	Stablecoin

Findings Count

Severity	**Amount**
Critical	1
High	3
Medium	6
Low	4
Total Findings	**14**

Summary of Findings

ID	Title	Severity	Status
[C-01]	delegate() permits unauthorized vote delegation	Critical	Resolved
[H-01]	Faulty deposit_for() check	High	Resolved
[H-02]	Losing voting power in rewards distribution	High	Resolved
[H-03]	Non-perpetual locks gaining extra delegation power	High	Resolved
[M-01]	Failing liquidation	Medium	Resolved
[M-02]	delegateBySig() wrongly limits delegation	Medium	Resolved
[M-03]	increase_amount() lacks support for perpetual lock amounts	Medium	Resolved
[M-04]	Persistent inflation from uninitialized rates	Medium	Resolved
[M-05]	Attacker can perform DOS attack by locking user	Medium	Resolved
[M-06]	MAX_DELEGATES limit can be bypassed	Medium	Resolved
[L-01]	The delegatee must own an NFT	Low	Resolved
[L-02]	Missing protection against slippage	Low	Resolved
[L-03]	No revocation for previous liquidation manager	Low	Resolved
[L-04]	Misleading error in increase_unlock_time()	Low	Resolved

8. Findings

8.1. Critical Findings

[C-01] `delegate()` permits unauthorized vote delegation

Severity

Impact: High

Likelihood: High

Description

The `delegate()` function in the `vePeg` contract is intended to delegate votes from an NFT (lock) to another NFT/lock (delegatee). However, the issue is that this function allows anyone to delegate on behalf of any lock (NFT) owner, which should not be the case. Delegation should only be allowed from the owner or operator of the lock:

```
function delegate(uint256 _from, uint256 _to) public {
        return _delegate(_from, _to);
    }
```

This results in the following risks:

- Malicious actors can delegate votes on behalf of any lock (NFT) owner, thus stealing their votes.
- Malicious actors can delegate votes on behalf of any owner to grief them, causing an increase in their checkpoints beyond the `MAX_DELEGATES` of 128, which leads to a DoS of all functions calling `_delegate()`, such as `lock_perpetually()`, `unlock_perpetual()`, and `delegateBySig()`.

Recommendations

Update the `delegate()` function to ensure that only the lock owner or operator can delegate votes:

```
function delegate(uint256 _from, uint256 _to) public {
+       require(_isApprovedOrOwner(msg.sender, _from));
        return _delegate(_from, _to);
}
```

8.2. High Findings

[H-01] Faulty `deposit_for()` check

Severity

Impact: Medium

Likelihood: High

Description

The `claim()` function in the `RewardsDistributor` contract is designed to claim rewards for `vePeg` locks. It transfers rewards directly if the lock is expired or calls the `vePeg.deposit_for()` function if the lock is not expired or is perpetually locked:

```
function claim(uint256 _tokenId) external returns (uint256) {
        //...
        if (amount != 0) {
            //...
            if (locked.end > block.timestamp || locked.perpetuallyLocked) {
                // lock has not expired
                ve.deposit_for(_tokenId, amount);
            } else {
                // lock expired
                address owner = ve.ownerOf(_tokenId);
                token.safeTransfer(owner, amount);
            }
            //...
        }
        //...
    }
```

The issue arises with perpetual locks. The `deposit_for()` function checks if `lock.end > block.timestamp`, but for perpetual locks, the lock's end time is always set to zero, which is considered expired by the implemented check in the `deposit_for()` function. This causes the `deposit_for()` function to revert when attempting to deposit rewards for perpetual locks:

```
function deposit_for(uint256 _tokenId, uint256 _value) external nonreentrant {
        LockedBalance memory _locked = locked[_tokenId];
        //...
        require(
          _locked.end>block.timestamp,
          "Cannotaddtoexpiredlock.Withdraw"
        );
        //...
    }
```

As a result, perpetual locks cannot receive their rewards, preventing them from increasing their voting power by the amount claimed. The only workaround is for the owner to unlock the perpetual lock using the `unlock_perpetual()` function.

Recommendations

Update the `deposit_for()` function to correctly handle perpetual locks by checking their status and updating the `perpetuallyLockedBalance` accordingly:

```
function deposit_for(uint256 _tokenId, uint256 _value) external nonreentrant {
        LockedBalance memory _locked = locked[_tokenId];
        //...
-       require
- (_locked.end > block.timestamp, "Cannot add to expired lock. Withdraw");
+       require
+ (_locked.end > block.timestamp || _locked.perpetuallyLocked , "Cannot add to expired

+       if (_locked.perpetuallyLocked) {
+           perpetuallyLockedBalance += _value;
+       }

        //...
    }
```

[H-02] Losing voting power in rewards distribution

Severity

Impact: Medium

Likelihood: High

Description

Both `RewardsDistributor` and `TokenRewardsDistributor` contracts lack of properly account for perpetual locks in their voting power and rewards calculations. The contracts calculate voting power using the linear decay formula (`bias - slope * time`), which incorrectly handle the perpetual locks as that perpetual locks should maintain constant voting power over time.

This issue affects the following functions in both contracts:

- `ve_for_at()`
- `_checkpoint_total_supply()`
- `_claim()`
- `_claimable()`

Since perpetual locks are initialized with `bias = 0` and `slope = 0`, the current reward logic calculates `zero` voting power for them.

As a result, users with perpetual locks unable to claim their rewards, despite having an active and constant lock. The only way for such users to receive rewards is to first convert their perpetual lock into a normal lock.

Recommendation

Update the implementation to handle the case for perpetual locks.

[H-03] Non-perpetual locks gaining extra delegation power

Severity

Impact: Medium

Likelihood: High

Description

The `vePeg._delegate()` function requires that the source token (`_from`) must be perpetually locked before allowing delegation. However, the `_moveAllDelegates()` function ignores this restriction and moves ALL tokens owned by the user to the new delegate, including non-perpetual locks.

This completely bypasses the intention where only perpetual locks should have delegation power.

```
function _delegate(uint256 _from, uint256 _to) internal {
    LockedBalance memory currentLock = locked[_from];
    require(currentLock.perpetuallyLocked == true, "Lock is not perpetual");
    --- SNIPPED ---
    _moveAllDelegates(delegator, currentDelegate, delegatee);
}
```

The `_moveAllDelegates()` function adds ALL tokens owned by the user to the delegation, not just perpetual ones.

```
//File: src/governance/vePeg.sol

function _moveAllDelegates
  (address owner, address srcRep, address dstRep) internal {
    --- SNIPPED ---

    if (dstRep != address(0)) {
        --- SNIPPED ---

        // Plus all that's owned
        for (uint256 i = 0; i < ownerTokenCount; i++) {
@>          uint256 tId = ownerToNFTokenIdList[owner][i];    //@audit This
// contains all locks, including non-perpetual locks
            dstRepNew.push(tId);
        }

        --- SNIPPED ---
    }
}
```

Recommendation

The `_moveTokenDelegates()` function could be used to delegate a specific token to a target user.

However, the `_delegate()` function currently takes token IDs but performs delegation at the address level, creating inconsistency and making the to token ID redundant. If the intended behavior is to delegate a single token's voting power, it would be clearer to update `_delegate()` to accept a delegatee address and use `_moveTokenDelegates()` accordingly. Otherwise, if address-level delegation is the intended design, the function interface should reflect that to avoid confusion.

8.3. Medium Findings

[M-01] Failing liquidation

Severity

Impact: Medium

Likelihood: Medium

Description

In the `setLiquidationManager()` function of the `PositionManager` contract, when the `liquidationManager` address is changed, the new contract is not granted approval on the share tokens for all registered vaults. This results in the **failure of liquidation for these vaults** when the `liquidationManager` attempts to redeem assets to pay for the liquidator:

```
function setLiquidationManager
   (address _newLiquidationManager) external onlyOwner {
        emit NewLiquidationManager(liquidationManager, _newLiquidationManager);

        liquidationManager = _newLiquidationManager;
    }
```

Recommendations

Implement a mechanism to grant the new `liquidationManager` address approval on all registered vaults shares.

[M-02] `delegateBySig()` wrongly limits delegation

Severity

Impact: Medium

Likelihood: Medium

Description

The `delegateBySig()` function in the `vePeg` contract is intended to allow anyone with a valid signature to delegate votes on behalf of a delegator. However, the issue is that this function includes a check requiring the caller to be the owner or operator (approved address) of the lock (NFT) to access it. This makes the function effectively useless for delegation by signature, as it should be callable by anyone with a valid signature, regardless of the caller status:

```
function delegateBySig(
  uint256from,
  uint256to,
  uint256nonce,
  uint256expiry,
  uint8v,
  bytes32r,
  bytes32s
)
        public
    {
        //...
        require(_isApprovedOrOwner(msg.sender, from), "Not approved");
        //...
    }
```

This behavior prevents the delegation from being done by an external party with a valid signature, contradicting the intended design and functionality of delegation by signature.

Recommendations

Remove the check that restricts the caller to be the owner or operator, as the `delegateBySig()` function should allow anyone with a valid signature to delegate votes:

```
function delegateBySig(
   uint256from,
   uint256to,
   uint256nonce,
   uint256expiry,
   uint8v,
   bytes32r,
   bytes32s
 )
        public
    {
        //...
-       require(_isApprovedOrOwner(msg.sender, from), "Not approved");
        //...
    }
```

[M-03] `increase_amount()` lacks support for perpetual lock amounts

Severity

Impact: Medium

Likelihood: Medium

Description

The `increase_amount()` function in the `vePeg` contract is designed to enable anyone to deposit additional tokens for any lock (`tokenId`) without modifying the unlock time. However, while this function is supposed to be compatible with both permanent and perpetual locks, **the current implementation only allows increasing the amount for non-perpetual locks**. This is due to a check that ensures `_locked.end > block.timestamp`, which is not applicable to perpetual locks as the `lock.end` is set to zero:

```
function increase_amount
  (uint256 _tokenId, uint256 _value) external nonreentrant {
    //...
    require
      (_locked.end > block.timestamp, "Cannot add to expired lock. Withdraw");
    //...
}
```

This results in perpetual locks being excluded from this functionality unless the owner/operator unlocks it.

Recommendation

Update the check in the `increase_amount()` function to handle perpetual locks, as follows:

```
function increase_amount(
        uint256 _tokenId,
        uint256 _value
    ) external nonreentrant {
     //...
-   require
- (_locked.end > block.timestamp, "Cannot add to expired lock. Withdraw");
+   require
+ (_locked.end > block.timestamp || _locked.perpetuallyLocked , "Cannot add to expired
    //...
}
```

[M-04] Persistent inflation from uninitialized rates

Severity

Impact: High

Likelihood: Low

Description

The `PositionManager.setInterestRateStrategy()` function enables the protocol owner to update the interest rate strategy contract. However, this operation does not include any mechanism to migrate or initialize the internal state required by the new strategy. Parameters such as `targetUtilization`, `endRateAt`, and `lastUpdate` for each vault remain uninitialized in the new strategy, which leads to inaccurate and inflated interest rate calculations.

```
function setInterestRateStrategy
  (address _newInterestRateStrategy) external onlyOwner {
    if (_newInterestRateStrategy == address(0)) {
        revert ZeroAddress();
    }

    emit NewInterestRateStrategy(address(interestRateStrategy), address
      (_newInterestRateStrategy));

    interestRateStrategy = IInterestRateStrategy(_newInterestRateStrategy);
}
```

The interest rate calculation function, `interestRate()`, relies on vault-specific parameters to compute a smooth and adaptive rate curve based on the difference between current utilization and the target utilization. If these parameters are missing or zero-initialized, the function calculates a large error

(`err`), resulting in excessive linear adaptation and an artificially high interest rate.

However, this function is strictly controlled by the owner and is expected to be used only in exceptional circumstances.

Recommendation

The new `interestRateStrategy` contract should either be initialized with the current state of each vault or properly migrate relevant parameters from the previous strategy to ensure accuracy and consistency in interest rate calculations.

[M-05] Attacker can perform DOS attack by locking user

Severity

Impact: Medium

Likelihood: Medium

Description

An attacker can use `create_lock_for()` to create locks on behalf of other users using just 1 wei and `MAXTIME`, reaching `MAX_DELEGATES`. As a result, when the affected user attempts to create a lock, the transaction will revert with `"dstRep would have too many tokenIds"` preventing them from creating new locks.

```
function create_lock_for(uint256 _value, uint256 _lock_duration, address _to)
        external
        nonreentrant
        returns (uint256)
    {
        return _create_lock(_value, _lock_duration, _to);
    }
```

This is a significant issue because the user cannot resolve it by calling `withdraw()` to eliminate the locks, as they must wait until the lock period expires. The attacker can set the lock to `MAXTIME`, forcing the user to wait a full year before being able to create a new lock.

```
require(block.timestamp >= _locked.end, "The lock didn't expire");
```

To better understand the issue, copy the following POC into `vePeg.t.sol`.

```
function test_DOS_createlockfor() external {

        uint256 aliceAmountToLock = 1;
        uint256 aliceLockDuration = 365 days;

        vm.startPrank(ALICE);

        peg.approve(address(ve), type(uint256).max);

        //Alice creates 128 locks, each with 1 wei and the maximum duration, to
        // perform a DOS attack on Bob
        for(uint256 i = 0; i < 128; i++) {

            uint256 aliceLockId = ve.create_lock_for
              (aliceAmountToLock, aliceLockDuration, BOB);
        }

        vm.stopPrank();

        vm.startPrank(BOB);

        uint256 bobAmountToLock = 1e18;
        uint256 bobLockDuration = 365 days;

        peg.approve(address(ve), bobAmountToLock);

        //Bob attempts to create a lock, but the transaction reverts due to
        // Alice's DoS attack.
        vm.expectRevert("dstRep would have too many tokenIds");
        ve.create_lock(bobAmountToLock, bobLockDuration);

        vm.stopPrank();
    }
```

Recommendations

Two potential fixes:

1. Set a minimum value requirement for the amount necessary for creating a lock to make the attack less feasible.
2. Restrict `create_lock_for()` access to trusted entities only.

[M-06] `MAX_DELEGATES` limit can be bypassed

Severity

Impact: Medium

Likelihood: Medium

Description

`delegate()` allows votes to be delegated from one ID to another. Each owner of an ID has a maximum limit of 128 delegates (`MAX_DELEGATES = 128`). However, this limit can be bypassed because the `require` statement does not validate the newly acquired delegates. This issue can be observed in `_moveAllDelegates()`.

Moreover, if the number of delegates increases significantly, it could trigger DOS. This happens because iterating through the loop may reach the block's gas limit, making it impossible to transfer the NFT or delegate votes.

```
function _moveAllDelegates
  (address owner, address srcRep, address dstRep) internal {

///code...

if (dstRep != address(0)) {
                uint32 dstRepNum = numCheckpoints[dstRep];
                uint256[] storage dstRepOld =

                                        dstRepNum > 0 ? checkpoints[dstRep][dstRepNum

                                    uint256[] storage dstRepNew = checkpoints[dstRep][dst
                uint256 ownerTokenCount = ownerToNFTokenCount[owner];
@>              require
  (dstRepOld.length <= MAX_DELEGATES, "dstRep would have too many tokenIds");
                // All the same
                for (uint256 i = 0; i < dstRepOld.length; i++) {
                    uint256 tId = dstRepOld[i];
                    dstRepNew.push(tId);
                }
                // Plus all that's owned
                for (uint256 i = 0; i < ownerTokenCount; i++) {
                    uint256 tId = ownerToNFTokenIdList[owner][i];
                    dstRepNew.push(tId);
                }

                if (_isCheckpointInNewBlock(dstRep)) {
                    numCheckpoints[dstRep] = dstRepNum + 1;
                    checkpoints[dstRep][dstRepNum].timestamp = _timestamp;
                } else {
                    checkpoints[dstRep][dstRepNum - 1].tokenIds = dstRepNew;
                    delete checkpoints[dstRep][dstRepNum];
                }
            }
        }
    }
```

As you can see, the check for `MAX_DELEGATES` is performed before the new delegates are added, allowing the limit to be bypassed.

To better understand the issue, copy the following POC into `vePeg.t.sol`.

```
function test_MAX_DELEGATES_can_be_bypassed() external {

        uint256 aliceAmountToLock = 1;
        uint256 aliceLockDuration = 365 days;

        vm.startPrank(ALICE);

        peg.approve(address(ve), type(uint256).max);

        for(uint256 i = 0; i < 128; i++) {

            uint256 aliceLockId = ve.create_lock_for
              (aliceAmountToLock, aliceLockDuration, ALICE);
            ve.lock_perpetually(aliceLockId);
        }

        vm.stopPrank();

        vm.startPrank(BOB);

        uint256 bobAmountToLock = 1;
        uint256 bobLockDuration = 365 days;

        peg.approve(address(ve), type(uint256).max);

        for(uint256 i = 0; i < 128; i++) {

            uint256 bobLockId = ve.create_lock_for
              (bobAmountToLock, bobLockDuration, BOB);
            ve.lock_perpetually(bobLockId);
        }

        vm.stopPrank();

        //1 is from Alice and 150 from Bob
        vm.prank(ALICE);
        ve.delegate(1, 150);

        //As a result, Bob ends up with 256 delegates, exceeding the
        // MAX_DELEGATES limit of 128.
    }
```

To observe the result, print `dstRepOld.length` inside `_moveAllDelegates()` after the new delegates are added. To do this, import `import {console} from "forge-std/console.sol";` and use `console.log("dstRepNew:", dstRepNew.length);` in `vePeg.sol`.

This approach is necessary because Foundry cannot directly access an array inside a struct from a public mapping. By doing this, you'll see that Bob's delegates reach 256, exceeding the `MAX_DELEGATES` limit of 128.

Recommendations

To resolve the issue, check the length of the delegates after the new ones have been added.

```
if (dstRep != address(0)) {
                uint32 dstRepNum = numCheckpoints[dstRep];
                uint256[] storage dstRepOld =

                                            dstRepNum > 0 ? checkpoints[dstRep][dstRepNum

                                    uint256[] storage dstRepNew = checkpoints[dstRep][dst
                uint256 ownerTokenCount = ownerToNFTokenCount[owner];
-               require
- (dstRepOld.length <= MAX_DELEGATES, "dstRep would have too many tokenIds");
                // All the same
                for (uint256 i = 0; i < dstRepOld.length; i++) {
                    uint256 tId = dstRepOld[i];
                    dstRepNew.push(tId);
                }
                // Plus all that's owned
                for (uint256 i = 0; i < ownerTokenCount; i++) {
                    uint256 tId = ownerToNFTokenIdList[owner][i];
                    dstRepNew.push(tId);
                }

+               require
+ (dstRepNew.length <= MAX_DELEGATES, "dstRep would have too many tokenIds");

                if (_isCheckpointInNewBlock(dstRep)) {
                    numCheckpoints[dstRep] = dstRepNum + 1;
                    checkpoints[dstRep][dstRepNum].timestamp = _timestamp;
                } else {
                    checkpoints[dstRep][dstRepNum - 1].tokenIds = dstRepNew;
                    delete checkpoints[dstRep][dstRepNum];
                }
            }
        }
    }
```

8.4. Low Findings

[L-01] The delegatee must own an NFT

Delegatee needs to own vePEG NFT in order to receive delegates from other users.

```
File: vePeg.sol

         address delegatee = _to == 0 ? address(0) : ownerOf(_to);
```

[L-02] Missing protection against slippage

In the `claimExcess()` function of the `LiquidationBuffer` contract, the contract owner can claim excess buffer by redeeming shares from a specified vault. However, there is no check for a minimum amount of redeemed assets that should be acceptable to be received by the owner, which makes this function not protected against slippage, and thus receiving less redeemed assets than intended:

```
function claimExcess(uint8 _vaultId) external onlyOwner {
        uint256 excess = claimableExcess[_vaultId];
        claimableExcess[_vaultId] = 0;

        if (excess > 0) {
            IVault(POSITION_MANAGER.getVault(_vaultId).addr).redeem
              (excess, msg.sender, address(this));
        }
    }
```

Recommendation: add a `minAmountOut` parameter and check against it to mitigate slippage.

[L-03] No revocation for previous liquidation manager

In the `setLiquidationManager()` function of the `PositionManager` contract, when changing the `liquidationManager` address, the approval of the previous

`liquidationManager` address **is not revoked from all registered vaults**. This oversight could result in compromising the `PositionManager` contract shares if the previous `liquidationManager` contract is compromised, as it would still retain approval to manage the assets in the vaults:

```
function setLiquidationManager
   (address _newLiquidationManager) external onlyOwner {
        emit NewLiquidationManager(liquidationManager, _newLiquidationManager);

        liquidationManager = _newLiquidationManager;
    }
```

Recommendation: implement a mechanism to revoke vaults approval from the old `liquidationManager` address.

[L-04] Misleading error in `increase_unlock_time()`

The `increase_unlock_time()` function contains a misleading error message when users attempt to increase the lock time for perpetual locks. The function first checks if the lock has expired before checking if it's a perpetual lock:

```
function increase_unlock_time
  (uint256 _tokenId, uint256 _lock_duration) external nonreentrant {
    --- SNIPPED ---
@>  require(_locked.end > block.timestamp, "Lock expired");
    require(_locked.amount > 0, "Nothing is locked");
    require(_locked.perpetuallyLocked == false, "Lock is perpetual");
    --- SNIPPED ---
}
```

Since perpetual locks have their `end` time set to `0`, the function will always revert with `"Lock expired"` instead of `"Lock is perpetual"` message.