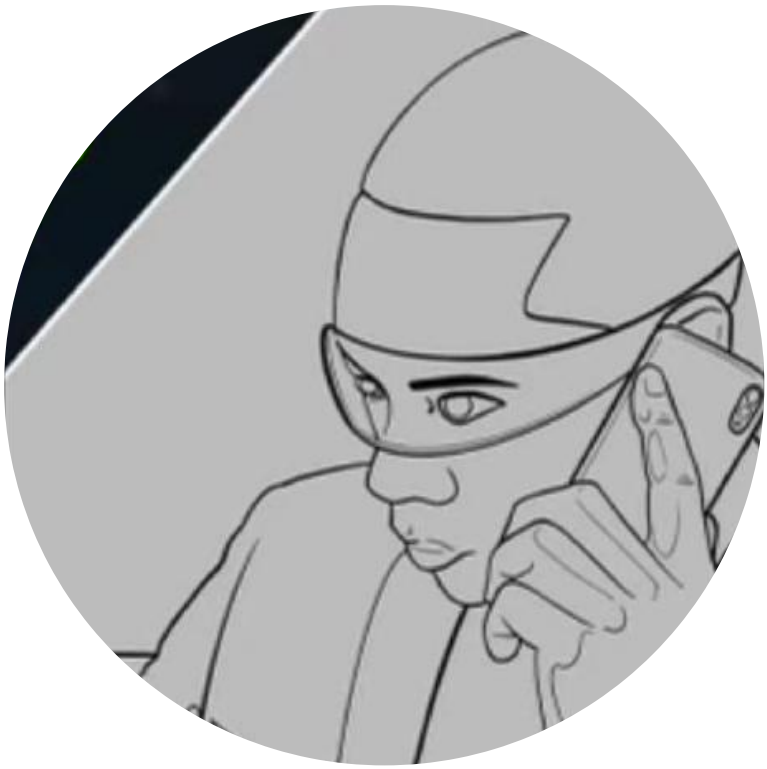

Hyperstable Security Review

Pashov Audit Group

Conducted by: aslanbek, jesjupyter, Ch_301

June 3rd 2025 - June 6th 2025

Contents

1. About Pashov Audit Group 2
2. Disclaimer 2
3. Introduction 2
4. About Hyperstable 2
5. Risk Classification 3
   5.1. Impact 3
   5.2. Likelihood 3
   5.3. Action required for severity levels 4
6. Security Assessment Summary 4
7. Executive Summary 5
8. Findings 7
   8.1. Medium Findings 7
      [M-01] Incorrect variable assignment 7
      [M-02] Reentrancy lock hinders reward distribution 8
   8.2. Low Findings 10
      [L-01] Attacker can fill Gauge array with whitelisted tokens 10
      [L-02] Double inheritance of UUPSUpgradeable and OwnableUpgradeable 10
      [L-03] Vault index missing in NewRewardsBps impedes reward tracking 11
      [L-04] Token transfer recipient incorrect in on-behalf operations 11
      [L-05] State migration missing in LiquidationManager upgrade 12

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 - 4404ccb3b82b329a02c01f05e71fd3b588df7e46

fixes review commit hash - 4190206ce93d4a275777a6b98742a45fb15b9b2a

Scope

The following smart contracts were in scope of the audit:

- `InterestRateStrategyV3`
- `PositionManagerV2`
- `LiquidationManagerV2`
- `GaugeV2`
- `GaugeProviderV2`
- `CurveGaugeFactory`
- `HyperSwapGaugeFactory`

7. Executive Summary

Over the course of the security review, aslanbek, jesjupyter, Ch_301 engaged with Hyperstable to review Hyperstable. In this period of time a total of **7** issues were uncovered.

Protocol Summary

Protocol Name	Hyperstable
Repository	https://github.com/hyperstable/contracts
Date	June 3rd 2025 - June 6th 2025
Protocol Type	Stablecoin

Findings Count

Severity	**Amount**
Medium	2
Low	5
Total Findings	**7**

Summary of Findings

ID	Title	Severity	Status
[M-01]	Incorrect variable assignment	Medium	Resolved
[M-02]	Reentrancy lock hinders reward distribution	Medium	Resolved
[L-01]	Attacker can fill Gauge array with whitelisted tokens	Low	Acknowledged
[L-02]	Double inheritance of UUPSUpgradeable and OwnableUpgradeable	Low	Resolved
[L-03]	Vault index missing in NewRewardsBps impedes reward tracking	Low	Resolved
[L-04]	Token transfer recipient incorrect in on-behalf operations	Low	Acknowledged
[L-05]	State migration missing in LiquidationManager upgrade	Low	Acknowledged

8. Findings

8.1. Medium Findings

[M-01] Incorrect variable assignment

Severity

Impact: Medium

Likelihood: Medium

Description

The constructor in `GaugeV2.sol` contains a critical logic error where the internal and external bribe contract addresses are incorrectly swapped during assignment. This results in the internal bribe contract being assigned the external bribe address, and vice versa.

Vulnerable Code:

```
constructor(
        address _stake,
        address _internalBribe,
        address _externalBribe,
      /***/
    ) {
        stake = _stake;
        internal_bribe = _externalBribe;
        external_bribe = _internalBribe;
```

Recommendations

This issue can be fixed as follows:

```
constructor(
        /***/
    ) {
        stake = _stake;
-        internal_bribe = _externalBribe;
-        external_bribe = _internalBribe;
+        internal_bribe = _internalBribe;
+        external_bribe = _externalBribe;
```

[M-02] Reentrancy lock hinders reward distribution

Severity

Impact: Medium

Likelihood: Medium

Description

A reentrancy lock conflict exists in the reward distribution flow between `GaugeV2` and `Voter` contracts.

1. User calls `GaugeV2.getReward()`. The first `nonReentrant` lock is acquired when `getReward` is called:

```
function getReward
  (address account, address[] memory tokens) external nonReentrant {
    require(msg.sender == account || msg.sender == voter);
    IVoter(voter).distribute(address(this));
    // ... rest of the function
}
```

2. This calls `Voter.distribute()` which then calls back to GaugeV2:

```
IGauge(_gauge).notifyRewardAmount(base, _claimable);
emit DistributeReward(msg.sender, _gauge, _claimable);
```

3. The call to notifyRewardAmount will revert because the reentrancy lock is still active:

```
function notifyRewardAmount
  (address token, uint256 amount) external nonReentrant {
    require(token != stake, "invalid reward");
    // ... rest of the function
}
```

This effectively breaks the reward distribution mechanism.

Recommendations

Follow `Gauge`:

```
_unlocked = 1;
        IVoter(voter).distribute(address(this));
        _unlocked = 2;
```

8.2. Low Findings

[L-01] Attacker can fill `Gauge` array with whitelisted tokens

In Gauge/GaugeV2, notifyRewardAmount adds the token to the array of reward tokens, if it was not added yet. There is no other validation, except that the token must be whitelisted in the Voter contract.

If there are more than 16 whitelisted tokens, and less than 16 reward tokens in the gauge, via `notifyRewardAmount`, an attacker can choose which tokens the gauge will be able to distribute until the end of its lifetime (and will not be able to distribute any others except these 16 tokens).

Proof of Concept

1. There are 50 whitelisted tokens.
2. Gauge is created with 3 reward tokens.
3. The attacker calls Gauge#notifyRewardAmount with 13 "worst" whitelisted reward tokens, which will discourage LPs from depositing into the Gauge, as it will not be possible to deposit any "better" tokens as rewards.

Recommendation: Introduce a way to pop reward tokens from `rewards` array.

[L-02] Double inheritance of `UUPSUpgradeable` and `OwnableUpgradeable`

`PositionManagerV2.sol` inheritance `UUPSUpgradeable` and `OwnableUpgradeable`. Also, it inherits `OperableUpgradeable`, which also has both `UUPSUpgradeable` and `OwnableUpgradeable` in the inheritance tree:

```
File: PositionManagerV2.sol
contract PositionManagerV2 is
    UUPSUpgradeable,
    OwnableUpgradeable,
    AccessControlUpgradeable,
    OperableUpgradeable,
    IPositionManager
{

File: OperableUpgradeable.sol
abstract contract OperableUpgradeable is UUPSUpgradeable, OwnableUpgradeable {
```

So, the `PositionManagerV2.sol` contract inheritance structure is like this:

```
UUPSUpgradeable    OwnableUpgradeable
           \                  /
            \                /
           OperableUpgradeable (abstract)
                    |
            PositionManagerV2
           /                  \
    UUPSUpgradeable    OwnableUpgradeable
```

This can be resolved by removing the inheritance in the abstract contract, as the direct inheritance already provides what you need.

[L-03] Vault index missing in `NewRewardsBps` impedes reward tracking

The `NewRewardsBps` event in `LiquidationManagerV2` is missing the vault index parameter, making it impossible to track reward configuration changes for specific vaults.

```
emit NewRewardsBps(
  config.liquidatorRewardBps, config.bufferRewardsBps,
  _newLiquidatorRewardBps, _newBufferRewardsBps
  );
```

To mitigate this issue, update the `NewRewardsBps` event to include the vault index as an indexed parameter.

[L-04] Token transfer recipient incorrect in on-behalf operations

The PositionManagerV2 contract implements `on-behalf-of` functionality through the `onlyUserOrOperator` modifier, allowing operators to perform actions for users.

However, in `onlyUserOrOperator`, we can see there are 2 kinds of operators:

```
$.isSystemOperator[msg.sender] || $.isAllowedOperator[_user][msg.sender]
```

However, in several critical functions, the contract incorrectly uses `msg.sender` (could be the `operator`) instead of `_user` (the intended recipient) for token operations:

```
...
DEBT_TOKEN.mint(msg.sender, _amountToBorrow);
...
DEBT_TOKEN.burn(msg.sender, _amountToRepay);
```

The problem is that the operators will receive the assets/tokens instead of the intended users. This breaks the fundamental trust model of `on-behalf-of` operations

It's true that the `HypeOperator` will help people dealing with `Native_TOKEN`, however, for operators that are appointed via `allowOperator`, it's not the same case for them, and transferring the token to these operators is incorrect.

Recommendations: Replace all instances of `msg.sender` with `_user`.

[L-05] State migration missing in `LiquidationManager` upgrade

During the upgrade from `LiquidationManager` to `LiquidationManagerV2`, the global liquidation reward parameters (`liquidatorRewardBps` and `bufferRewardsBps`) were replaced.

```
uint256 public liquidatorRewardBps = 0.25e18; // 25%
uint256 public bufferRewardsBps = 0.25e18; // 25%
```

In LiquidationManagerV2, these were replaced with a per-vault mapping:

```
mapping(uint8 => LiquidationConfiguration) public liquidationConfiguration;
```

The critical issue is that when upgrading to V2, there was no state migration process to set the `liquidationConfiguration` values for existing vaults.

This means that after the upgrade, all vaults will have zero values for both `liquidatorRewardBps` and `bufferRewardsBps` in their `liquidationConfiguration`(unless `setRewardsBps` is called later).

The problem is that the `_liquidate` function in V2 uses `liquidationConfiguration[_index]` to determine reward distributions:

```
(
          IPositionManager.VaultDatamemoryvaultData,
          IPositionManager.PositionDatamemorypositionData
        ) =
            manager.initLiquidation(_index, _positionOwner);

        LiquidationValues memory values =
            _getLiquidationValues(
              vaultData,
              positionData,
              liquidationConfiguration[_index],
              _debtToRepay
            );

        if (values.CR >= vaultData.MCR) {
            revert NothingToLiquidate();
        }
```

With zero reward parameters, this could cause unintended consequences.

Recommendations: Implement a migration function in the upgrade process that sets the `liquidationConfiguration` values for all existing vaults to match the previous global parameters.

Could refer to `interestRateStrategyV3`:

```
function initialize(bytes memory) external reinitializer(3) {
        uint256 currentRate = ManualIRM.interestRate();

        uint256 vaultCount = IPositionManager
          (positionManagerAddress).lastVaultIndex();

        for (uint8 i; i < vaultCount; i++) {
            ManualIRMV2.setInterestRate(i, currentRate);
        }
    }
```