Hyperstable Security Review

Pashov Audit Group

Conducted by: Hals, Ch_301, merlinboii, shaflow

February 26th 2025 - March 3rd 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 - 00dedbdb52474ae34caa3e2571c4b29f34b14c00

fixes review commit hash - cb30a683971974f6205c453a0d172a5431daa935

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, shaflow engaged with Hyperstable to review Hyperstable. In this period of time a total of **30** issues were uncovered.

Protocol Summary

Protocol Name	Hyperstable
Repository	https://github.com/hyperstable/contracts
Date	February 26th 2025 - March 3rd 2025
Protocol Type	Stablecoin

Findings Count

Severity	Amount
High	1
Medium	9
Low	20
Total Findings	**30**

Summary of Findings

ID	Title	Severity	Status
[H-01]	Attacker can make his vePeg NFT unpokeable	High	Resolved
[M-01]	EmissionScheduler and Minter contracts can't be initialized due to an incorrect startTime check	Medium	Resolved
[M-02]	Dead gauges are not handled correctly in Voter._vote() function	Medium	Resolved
[M-03]	User deposits may be vulnerable to sandwich attacks	Medium	Resolved
[M-04]	Fixed liquidator reward based on MCR may cause protocol losses	Medium	Acknowledged
[M-05]	Self-liquidation can help reduce losses when health factor is low	Medium	Acknowledged
[M-06]	DOS for removal of delegates	Medium	Resolved
[M-07]	Incorrect update of ve_supply in checkpoint_total_supply()	Medium	Resolved
[M-08]	Lack of minimum deposited amount can result in bad debt	Medium	Resolved
[M-09]	Incorrect decimal handling	Medium	Resolved
[L-01]	RewardsDistributor : reward claim fails when time equals lock expiration time	Low	Resolved
[L-02]	Missing revocation of token approval in Voter.killGauge() function	Low	Resolved
[L-03]	Missing Liquidation event emission when bufferSurplus > 0	Low	Resolved

[L-04]	Potential outdated and security-issued code in governance contracts	Low	Resolved
[L-05]	Lack of ownership management leads to permanent loss of administrative control	Low	Resolved
[L-06]	Potential incompatibility with stHYPE leads to incorrect asset balance calculation	Low	Acknowledged
[L-07]	InterestRateStrategyV1 allows unauthorized initialization of implementation	Low	Resolved
[L-08]	Missing vault registration check enables duplicate vault registrations	Low	Acknowledged
[L-09]	The liquidation reward does not match the documentation	Low	Acknowledged
[L-10]	Check MCR in _registerData in registerVault.	Low	Resolved
[L-11]	Incorrect supply increase in _deposit_for() for MERGE_TYPE deposits	Low	Resolved
[L-12]	Missing claimable reward transfer in killgauge() function	Low	Resolved
[L-13]	Temporary voting disruption after reset	Low	Acknowledged
[L-14]	Initial debt repayment issue due to low USDH circulating supply	Low	Acknowledged
[L-15]	Minter.updatePeriod fails if Voter.totalWeight is zero	Low	Acknowledged
[L-16]	Voter rewards for dead gauges become stuck due to missing transfer	Low	Resolved

[L-17]	Missing minAmountCollateral in liquidate()	Low	Acknowledged
[L-18]	Vault is susceptible to inflation attack by first depositor	Low	Acknowledged
[L-19]	Vault.sharePrice incorrectly assumes 18 decimals for every vault	Low	Resolved
[L-20]	The missingShares could lead to liquidate more portions	Low	Acknowledged

8. Findings

8.1. High Findings

[H-01] Attacker can make his `vePeg` NFT unpokeable

Severity

Impact: Medium

Likelihood: High

Description

Since `vePeg` NFT balances decay linearly over time, votes can become outdated if not updated, this is why we have the `poke` mechanism in the system. It allows admins to update the `vePeg` NFTs voting weight to reflect its current balance. This ensures fairness and accuracy in the voting system.

A user can intentionally make their `vePeg` NFT unpokeable by exploiting a "dust vote" strategy. Here's how:

Suppose the current `vePeg` NFT balance is `10e18`. The user votes for two pools: Allocates `10e18 - 1` weight to their preferred pool. Allocates 1 weight to a random pool (a "dust vote"). After 1 second, the `vePeg` NFT's balance decays to slightly less than 10e18. At this point, any attempt to poke the `vePeg` NFT will fail because: The calculation `1 * veWeight / 10e18` for the dust vote will round down to 0 due to the reduced weight.

The `require(_poolWeight != 0)` check will revert the transaction, making the `vePeg` NFT effectively unpokeable.

```
File: Voter.sol#_vote()

            if (isGauge[_gauge]) {

              uint256 _poolWeight = _weights[i] * _weight / _totalVoteWeight;
                require(votes[_tokenId][_pool] == 0);
                require(_poolWeight != 0);
```

Users can intentionally make their `vePeg` NFTs unpokeable, preventing their voting weights from being updated to reflect their current balance. the `vePeg` NFTs continue to contribute votes based on outdated balances, skewing the voting system and potentially disadvantaging other users.

Recommendations

The `_vote()` function should not revert on 0 vote, but instead continues with the loop.

8.2. Medium Findings

[M-01] `EmissionScheduler` and `Minter` contracts can't be initialized due to an incorrect `startTime` check

Severity

Impact: Medium

Likelihood: Medium

Description

In the `EmissionScheduler` contract, the `initialize()` function is intended to initialize the contract and is invoked via `Minter.initialize()` function. However, the function incorrectly reverts if the `startTime` equals zero. Since the `startTime` should be zero to initialize the contract first; the check made to prevent **reinitialization will incorrectly prevent the contract from being initialized** , where this consequently blocks the initialization of the `Minter` contract as well:

```
function initialize(uint256 _startTime) external {
        _onlyMinter();
        if (startTime == 0) {
            revert Errors.AlreadyInitialized();
        }
        startTime = _startTime;
    }
```

Recommendations

```
function initialize(uint256 _startTime) external {
        _onlyMinter();
-       if (startTime == 0) {
+       if (startTime != 0) {
            revert Errors.AlreadyInitialized();
        }
        startTime = _startTime;
    }
```

[M-02] Dead gauges are not handled correctly in `Voter._vote()` function

Severity

Impact: Medium

Likelihood: Medium

Description

In the `Voter` contract, the `_vote()` function is called when a user casts a vote via `vote()` or when they update their votes using `poke()`. However, the function currently does not include a check to verify whether the gauge is alive or not.

This missing check will result in:

1. Stuck rewards: when the function processes voting on a dead gauge, the accrued rewards (shares) of the dead gauge via `_updateFor()` function will be **stuck** in the contract because they will not be assigned to the dead gauge as there's a check for gauge liveliness in the `updateFor()` function, which leads to the accumulation of rewards that remain locked within the contract .
2. Incorrect deposit in the bribe contracts.
3. Inflated `totalWeight`: the failure to account for a dead gauge also results in the incorrect increase of `totalWeight`, where this value impacts the `index` calculations, which is tied to reward notifications via `notifyRewards()`, as a result; accounting for dead gauges will inflate the `totalWeight` leading to reduced reward allocations for other gauges.

```
function _vote(
  uint256_tokenId,
  address[]memory_poolVote,
  uint256[]memory_weights
) internal {
        _reset(_tokenId);
        //...

        for (uint256 i = 0; i < _poolCnt; i++) {
            _totalVoteWeight += _weights[i];
        }

        for (uint256 i = 0; i < _poolCnt; i++) {
            address _pool = _poolVote[i];
            address _gauge = gauges[_pool];

            if (isGauge[_gauge]) {
                //...
                _updateFor(_gauge);

                //...
                IBribe(internal_bribes[_gauge])._deposit(uint256
                  (_poolWeight), _tokenId);
                IBribe(external_bribes[_gauge])._deposit(uint256
                  (_poolWeight), _tokenId);
                _usedWeight += _poolWeight;
                _totalWeight += _poolWeight;
                emit Voted(msg.sender, _tokenId, _poolWeight);
            }
        }
        if (_usedWeight > 0) IVotingEscrow(_ve).voting(_tokenId);
        totalWeight += uint256(_totalWeight);
        usedWeights[_tokenId] = uint256(_usedWeight);
    }
```

```
function _updateFor(address _gauge) internal {
        //...
        if (_supplied > 0) {
            //...
            if (_delta > 0) {
                uint256 _share = uint256
                //(_supplied) * _delta / 1e18; // add accrued difference for each supp
                if (isAlive[_gauge]) {
                    claimable[_gauge] += _share;
                }
            }
        } else {
            supplyIndex[_gauge] = index; // new users are set to the default
            // global state
        }
    }
```

Recommendations

Consider updating the `_vote()` function to check if the gauge is alive, and continue the function without reverting by skipping dead gauges.

[M-03] User deposits may be vulnerable to sandwich attacks

Severity

Impact: Medium

Likelihood: Medium

Description

In the `PositionManager` contract, the `deposit()` function allows users to deposit the underlying asset of a specified vault and lock them for collateral. However, the function does not introduce `minAmountOut` (the minimum amount of shares a user is willing to receive) .

The lack of this parameter exposes the depositors to **sandwich attacks**, where a malicious actor could front-run the deposit transaction to manipulate the share price, resulting in depositors receiving fewer shares than expected, reducing their effective collateral.

Same issue with `withdraw()` function, where there's no minimum amount of redeemed assets acceptable to be received by the user.

Recommendations

Introduce a `minAmountOut` parameter to ensure that the user is only willing to accept a minimum amount of shares for the deposit (and for withdrawals), and if the final amount of shares is less than `minAmountOut`, the transaction should be reverted.

[M-04] Fixed liquidator reward based on MCR may cause protocol losses

Severity

Impact: High

Likelihood: Low

Description

The liquidation mechanism calculates liquidator rewards based on the vault's MCR value (`@1>`), which can lead to unnecessary protocol losses even if the position is still solvent (collateral can cover debt).

The issue arises because the reward calculation assumes the position should maintain MCR-level collateralization even during liquidation, creating excessive rewards that may deplete the protocol buffer or distribute losses among debtors.

```
function _getLiquidationValues(
    ...
) internal view returns (LiquidationValues memory) {
    --- SNIPPED ---

    uint256 equivalentCollateral = values.debtToRepay.divWad(values.sharePrice);
@1>  uint256 overcollateralization = equivalentCollateral.mulWad
  (_vaultData.MCR - 1e18);

    uint256 liquidatedCollateral =
        _positionData.collateralSnapshot.mulDiv
          (values.debtToRepay, _positionData.debtSnapshot);

    if (liquidatedCollateral < requiredCollateral) {
        // the amount of shares that are needed to cover the required collateral
@2>     values.missingShares = requiredCollateral - liquidatedCollateral;
        // the shares that are going to be redeemed
        values.sharesToRedeem = liquidatedCollateral;
    } else {
    --- SNIPPED ---
}
```

The current formula calculates rewards as: `Debt_Equivalent * (MCR-100)/100 * liquidatorRewardBps (fixed 25%)`

This means:

- For MCR = 120%: Liquidator gets 5% of debt value (20% * 25%)
- For MCR = 150%: Liquidator gets 12.5% of debt value (50% * 25%)
- For MCR = 200%: Liquidator gets 25% of debt value (100% * 25%)

However, positions with `collateral > debt` but `< debt * (1 + reward rate)` create unnecessary losses, calculated as follows: 0. Assume there is 0 interest rate, MCR = 120%

1. Collateral value worth `1200 USDH` and their debt is `1000 USDH`
2. At a flash-crashed market, the price drops and collateral becomes worth `1020 USDH` (liquidatable state)

3. When liquidating the position, the liquidator will pay : `1000 USDH` (debt) : and get (`1000 * 20%) * 25% = 50 USDH`, requiring `1050 USDH` equivalent of collateral for paying the liquidator.

4. As the collateral now worth only `1020 USDH`, it falls into the case to take the `missingShares` (`@2>`). At this state, the collateral can still cover the debt (`1000 USDH`) but cannot cover the required rewards (`50 USDH`).

5. Therefore, the process takes an extra `30 USDH` equivalent of collateral from (`@3>`) either the `liquidationBuffer` (which normally holds funds from the normal liquidated position fee), **distributes loss to all collateral shares holders of the vault**, or even worse, it becomes **illiquidable**.

6. It can be observed that this mechanism reduces the health of the position during partial liquidation. Additionally, if the MCR is higher, the threshold for this scenario also increases.

```
function _liquidate(...)
    internal
    returns (LiquidationValues memory)
{
    --- SNIPPED ---

    // take from buffer
@3>  if (missingShares > 0) {
        ILiquidationBuffer buffer = liquidationBuffer;
        --- SNIPPED ---
        rewards += buffer.redeem(vaultData.addr, _liquidator, toRedeem);
        }
    }

    // take from vault
@3>  if (missingShares > 0) {
            --- SNIPPED ---
            IVault(vaultData.addr).take(_liquidator, assetsToTake);

            rewards += assetsToTake;
            missingShares -= sharesToTake;
        }
    }

@3>  if (missingShares > 0) {
        revert NotEnoughRewards();
    }

    --- SNIPPED ---
}
```

Recommendation

As for the mentioned case, the CR of the position will be around `100% < CR < ((MR-100%) * 25%)` (solvent state but liquidatable). In this case, the process should cap the incentive to the amount of collateral available.

[M-05] Self-liquidation can help reduce losses when health factor is low

Severity

Impact: High

Likelihood: Low

Description

When the user's collateral is insufficient to cover the liquidation reward, tokens will be withdrawn from the LiquidationBuffer to pay the liquidator. If the LiquidationBuffer also lacks sufficient tokens to cover the liquidation reward, tokens will be withdrawn from the Vault to pay the liquidator.

If the user's collateral is insufficient to pay the liquidation reward, they can opt for self-liquidation, which allows the protocol to absorb part of their loss, as compared to the normal repayment method.

For example, if the current value of the user's collateral is 100, and the liquidation of the entire debt requires collateral worth 110 (to reward the liquidator), the user can choose to self-liquidate to reduce their loss due to the decline in the value of the collateral,.

Recommendations

It is recommended to prevent self-liquidation or restrict liquidation of debts in the redistribution state to privileged addresses only.

[M-06] DOS for removal of delegates

Severity

Impact: Low

Likelihood: High

Description

The delegation system on `vePeg.sol` has the feature that `MAX_DELEGATES` is a hardcoded value

On the other hand, users can create locks with only 1 wei LP because `_create_lock()` only has this check `require(_value > 0);`

Now, take this scenario:

Alice owns vePeg_NFT_01 (with big weight) and she decides to delegate it to Bob

Malicious user back-run Alice transaction and delegated to Alice a `MAX_DELEGATES` vePeg_NFT (only 1 wei LP)

After a period, Alice will try to move her delegates of the vePeg_NFT_01 from Bob, but it will fail due to this requirement.

```
File: vePeg.sol#_moveAllDelegates

                require(
                  dstRepOld.length+ownerTokenCount<=MAX_DELEGATES,
                  "dstRepwouldhavetoomanytokenIds"
                );
```

The user is not able to remove his vePeg_NFT from the delegate for 52 Epochs.

Anyone can deprive users of receiving any delegations with this attack.

Recommendations

```
File: VotingEscrow.sol

                require(

-                       dstRepOld.length + ownerTokenCount <= MAX_DELEGATES,

+                       dstRepOld.length <= MAX_DELEGATES,

                     "dstRep would have too many tokenIds"

                 );
```

[M-07] Incorrect update of `ve_supply` in `checkpoint_total_supply()`

Severity

Impact: High

Likelihood: Medium

Description

In the `RewardsDistributor.sol` contract, the `checkpoint_total_supply()` function is responsible for storing the total supply at a given time `t` in the `ve_supply[t]` mapping. This value is used for future reward distribution calculations. The relevant code is as follows:

```
File: RewardsDistributor.sol
158:                    ve_supply[t] = FixedPointMathLib.max(uint256(int256
  (pt.bias - pt.slope * dt)), 0);
```

The `ve_supply[t]` value should only be updated when the week corresponding to time `t` has ended, when `t + 1 weeks <= block.timestamp`. However, the current implementation allows `ve_supply[t]` to be updated incorrectly when `block.timestamp % 1` weeks is zero.

This creates a vulnerability where the balance of a newly created `vePeg` NFT (created immediately after `checkpoint_total_supply()` is called) is not accounted for in `ve_supply[t]`. A malicious user could exploit this flaw to manipulate reward calculations and steal future distribution rewards.

This will lead to reward manipulation. A malicious actor could create a `vePeg` NFT at a specific time to exclude its balance from `ve_supply[t]`, leading to incorrect reward distributions.

Recommendations

To mitigate this issue, ensure that `ve_supply[t]` is only updated when the week corresponding to time `t` has fully elapsed. This can be achieved by:

```diff
File: RewardsDistributor.sol

         for (uint256 i = 0; i < 20; i++) {
-            if (t > rounded_timestamp) {
+            if (t >= rounded_timestamp)
```

[M-08] Lack of minimum deposited amount can result in bad debt

Severity

Impact: Medium

Likelihood: Medium

Description

In the `PositionManager` contract, the `deposit()` function allows users to deposit assets into any supported vault with no minimum deposit size. This enables the creation of very small deposit positions, referred to as "dust deposits", where these small deposits can then be used as collateral to borrow against them.

However, if the **health factor** of these small positions falls below the **minimum collateral ratio (MCR)** threshold, there is **no incentive for liquidators** to liquidate them due to the low value of the collateral. As a result, these dust deposit positions remain open, causing the protocol to accumulate **bad debt**.

Recommendations

Introduce a minimum deposit size for collateral to ensure that users cannot open positions with very small amounts of assets.

[M-09] Incorrect decimal handling

Severity

Impact: High

Likelihood: Low

Description

The `PositionManager` and `LiquidationManager` contracts incorrectly handles decimals when calculating across various processes. The issue stems from assuming all values are 18 decimals when performing calculations, while vault shares inherit decimals from their underlying token through `Vault.decimals()`.

Therefore, for the vault supported assets with decimals != 18, ot the decimal offset > 0, the current process will treat the share amount as 18 decimals despite the fact that the `colalteralSnapshots` are stored with vault decimals (`@1>`).

```
//File: src/core/PositionManager.sol

function deposit(uint8 _index, uint256 _amountToDeposit) external {

    --- SNIPPED ---

    asset.transferFrom(msg.sender, address(this), _amountToDeposit);

@1>  uint256 shares = vault.deposit(_amountToDeposit, address(this));

    _accruePositionDebt(_index, vaultData, positionData);

@1>  vaultData.collateralSnapshot += shares;

@1>  positionData.collateralSnapshot += shares;

    emit Deposit(vaultData.addr, msg.sender, _amountToDeposit);

}
```

This causes:

1. Incorrect value to calculate `CR` for the position across the `PositionManager` and `LiquidationManager` contracts:

- `PositionManager._checkCR()`
- `PositionManager.accountCr()`
- `LiquidationManager._getLiquidationValues()`
- `LiquidationManager._calculateCr()`

This can both bypass the `CR` checks (for > 18 decimals) or always present the `CR` less that `MCR` (for < 18 decimals) that will cause position to present as always liquidatable.

2. Incorrect value to calculate `requiredCollateral` in the `LiquidationManager._getLiquidationValues()` as it always present the 18 decimals, when it compares and processes with the `liquidatedCollateral` (`_positionData.collateralSnapshot`) the value is wrong in that process.

Note that the `WBTC` asset with `8` decimals value is potetially to be used as per the documentation.

Recommendation

Correctly normalize the decimals in the `PositionManager` and `LiquidationManager` contracts for the vaults with decimals != 18 when handling those values in both calculation, comparison, and validation processes.

8.3. Low Findings

[L-01] `RewardsDistributor` : reward claim fails when time equals lock expiration time

In the `RewardsDistributor` contract, the `claim()` and `claim_many()` functions distribute rewards to the lock owner. If the lock is expired, rewards are transferred directly to the lock owner. If the lock is not expired, the rewards are deposited for the owner in the voting escrow via `ve.deposit_for()`.

```
function claim(uint256 _tokenId) external returns (uint256) {
        //...
        if (amount != 0) {
            IVotingEscrow ve = IVotingEscrow(veAddress);
            if (block.timestamp > ve.locked(_tokenId).end) {
                address owner = ve.ownerOf(_tokenId);
                token.safeTransfer(owner, amount);
            } else {
                ve.deposit_for(_tokenId, amount);
            }
            token_last_balance -= amount;
        }
        //...
    }
```

However, according to the logic in the voting escrow contract (`vePeg`), a lock is considered expired if `locked.end <= block.timestamp`. This creates an issue when `block.timestamp == locked.end` because the `ve.deposit_for()` function will attempt to deposit the rewards, but it will fail since the lock is considered expired at that exact moment. This results in the user being unable to call the `claim()` or `claim_many()` function when the current time is exactly equal to the end of the lock.

```
// vePeg contract:
    function deposit_for
      (uint256 _tokenId, uint256 _value) external nonreentrant {
        //...
        require(
          _locked.end>block.timestamp,
          "Cannotaddtoexpiredlock.Withdraw"
        );
        //...
    }
```

Recommendation: consider handling the case where the lock's end time equals the current timestamp.

[L-02] Missing revocation of token approval in `Voter.killGauge()` function

In the `Voter` contract, the `killGauge()` function is responsible for removing a gauge when called by the `emergencyCouncil`. While the function sets the gauge's `isAlive[_gauge]` status to `false` and resets its claimable rewards to zero, **it does not revoke the token approval** granted to the gauge when it was registered via the `createGauge()` function. So if the killed gauge becomes compromised or maliciously controlled, the lack of approval revocation allows it to misuse the permissions it still holds to drain the `Voter` contract funds.

Recommendation: modify the `killGauge()` function to revoke the token approval granted to the gauge during its creation.

[L-03] Missing `Liquidation` event emission when `bufferSurplus > 0`

In the `LiquidationManager` contract, the `liquidate()` function does not emit the `Liquidation` event when a `bufferSurplus` is > 0 as the function returns before the event is emitted if there is any surplus collateral.

```
function _liquidate(
    uint8_index,
    address_positionOwner,
    address_liquidator,
    uint256_debtToRepay
  )
        internal
        returns (LiquidationValues memory)
    {
        //...

        if (values.bufferSurplus > 0) {
            ILiquidationBuffer buffer = liquidationBuffer;
            IVault(vaultData.addr).transferFrom(address(manager), address
              (buffer), values.bufferSurplus);
            buffer.notifyDeposit(vaultData.addr, values.bufferSurplus);

            assert(values.missingShares == 0);

            // if there is a buffer surplus then there are no missing shares
            return values;
        }

        //...
    }
```

Recommendation: ensure that the `Liquidation` event is emitted when `bufferSurplus` is > 0 .

[L-04] Potential outdated and security-issued code in governance contracts

During a differential review of the code for the governance contracts, we observed that **some code paths appear outdated** despite recent modifications.

Additionally, we found that **some portions of the codebase have been previously audited and flagged for security issues** in past reviews.

To ensure the integrity and security of the governance module, we recommend the team to cross-check the existing code with past security reports, particularly:

- https://github.com/spearbit/portfolio/blob/master/pdfs/Velodrome-Spearbit-Security-Review.pdf
- Assess and apply necessary security fixes to address any unresolved vulnerabilities.

[L-05] Lack of ownership management leads to permanent loss of administrative control

In the `Peg` contract, once the `minter` role is transferred to the `Minter` contract, there is no way to update the `setMerkleClaim` address again.

The issue stems from the fact that:

1. The `minter` role is initially set to `msg.sender` in the constructor
2. The contract provides a `setMinter()` function that can only be called by the current minter
3. Once transferred to the `Minter` contract, the `Minter` contract has no functionality to control other administrative functions: `setMerkleClaim()`

This creates a permanent lock of administrative control, preventing any future updates to the contract's parameters, including the ability to set a new merkle claim address if needed.

Consider introducing other ownership management functionality into the `Peg` contract separately from the `Minter`.

[L-06] Potential incompatibility with `stHYPE` leads to incorrect asset balance calculation

The `Vault` contract inherits OpenZeppelin's `ERC4626` implementation which uses `balanceOf()` to determine `totalAssets()`.

However, for `stHYPE` tokens, the actual balance including staking rewards must be obtained through the `assetsOf()` function instead of `balanceOf()` as per the current `sthype-docs` for integration guide.

This incompatibility means that the vault will undervalue `stHYPE` collateral by not accounting for staking rewards, leading to:

1. Incorrect collateral valuations in position health checks
2. Wrong share prices for deposits/withdrawals

3. Inaccurate liquidation thresholds

Note that the `stHYPE` asset is potentially intended to be used as per protocol documentation, but the stHYPE implementation code has not yet been confirmed.

[L-07] `InterestRateStrategyV1` allows unauthorized initialization of implementation

The `InterestRateStrategyV1` contract does not disable initialization. This allows the ownership of the implementation being taken.

```
contract
   InterestRateStrategyV1 is OwnableUpgradeable, UUPSUpgradeable, IInterestRateStrateg
    --- SNIPPED ---
    function initialize(
      address_owner,
      address_newPositionManagerAddress,
      address_usdhPriceFeed
    )
        external
        initializer
    {
        _transferOwnership(_owner);
        positionManagerAddress = _newPositionManagerAddress;
        PegIRM.setPriceFeed(_usdhPriceFeed);
    }
    --- SNIPPED ---
}
```

Consider adding the constructor to disable initializers for the implementation.

```
/// @custom:oz-upgrades-unsafe-allow constructor
constructor() {
    _disableInitializers();
}
```

[L-08] Missing vault registration check enables duplicate vault registrations

The `PositionManager.registerVault()` function does not verify if a vault is already registered before adding it to the `vaults` mapping. This allows the same vault address to be registered multiple times with potentially different MCR values and interest indices.

Consider apply checking to avoid adding the same vault multiple times.

[L-09] The liquidation reward does not match the documentation

The documentation states that the liquidator receives a collateral reward equal to 105% of the liquidated asset value. However, in the contract, the liquidation reward is not fixed at 105% but instead depends on the vault's MCR and `liquidatorRewardBps`.

```
values.debtToRepay = FixedPointMathLib.min
          (_debtToRepay, _positionData.debtSnapshot);
        uint256 equivalentCollateral = values.debtToRepay.divWad
          (values.sharePrice);
        uint256 overcollateralization = equivalentCollateral.mulWad
          (_vaultData.MCR - 1e18);
```

It is recommended to either update the documentation or adjust the contract logic to ensure consistency between them.

[L-10] Check MCR in `_registerData` in `registerVault`.

The `registerVault` operation takes in two MCR values—one explicitly passed as a variable and recorded in the global `VaultData`, while the other is encoded in `_registerData` and used to calculate the initial `interestRate`. These two MCR values must be the same; otherwise, the initial `interestRate` calculation may be incorrect.

```
function registerVault
      (address _vaultAddress, uint256 _mcr, bytes memory _registerData)
        external
        onlyOwner
        returns (uint8)
    {
        uint8 index = lastVaultIndex;
+       (uint256 mcr, ) = abi.decode(_registerData, (uint256, uint256));
+       require(mcr == _mcr, "mcr misMatch");
```

[L-11] Incorrect supply increase in `_deposit_for()` for MERGE_TYPE deposits

In the _deposit_for() function, the supply is incorrectly increased when the deposit type is MERGE_TYPE. This occurs because the supply is updated unconditionally, regardless of the deposit type. However, for MERGE_TYPE deposits (triggered by the merge() function), the supply should not be increased, as merging two positions does not introduce new value into the system.

To fix this issue, the `supply` should only be increased when the deposit type is not MERGE_TYPE.

```
// Only increase supply if the deposit type is not MERGE_TYPE
    if (deposit_type != DepositType.MERGE_TYPE) {
        supply = supply_before + _value;
    }
```

[L-12] Missing claimable reward transfer in `killgauge()` function

The new `killGauge()` function fails to return the claimable rewards associated with the gauge back to the minter. Specifically, the logic to transfer any remaining `claimable[_gauge]` rewards to the minter and reset the claimable value is missing. This omission could result in unclaimed rewards being permanently locked in the contract.

[L-13] Temporary voting disruption after reset

When a user calls the `Voter.sol#reset()` function during epoch N, they are unable to vote again in the same epoch (epoch N). This occurs because the lastVoted[_tokenId] timestamp is updated to the current block timestamp, effectively locking the user out of voting until the next epoch (epoch N+1).

So, Users who reset their votes in the current epoch are unable to participate in voting until the next epoch begins, and this will cause them to lose rewards.

To address this issue, consider modifying the logic to allow users who call `reset()` in epoch N to still vote within the same epoch.

[L-14] Initial debt repayment issue due to low USDH circulating supply

When the `PositionManager.sol` contract is deployed, the first users who take on debt may face an issue where they cannot fully repay their debt. This occurs because the circulating supply of USDH is initially too low, making it impossible for users to acquire enough USDH to repay their debt entirely. As a result, these users are forced to keep some collateral locked in the `PositionManager.sol` contract. This will cause: Early users cannot fully repay their debt due to insufficient USDH in circulation. Users are unable to withdraw 100% of their collateral, leading to potential frustration and inefficiency in the system.

To resolve this issue, consider implementing a mechanism to sell the claimed interest in the market. This would allow users with debt to purchase the necessary USDH to fully repay their debt and withdraw their collateral.

[L-15] `Minter.updatePeriod` fails if `Voter.totalWeight` is zero

In the `Minter` contract, the `updatePeriod()` function is responsible for distributing rewards to the `RewardDistributor` contract, `team`, and `Voter` contract. The function calls `VOTER.notifyRewardAmount()` after sending the rewards to the `Voter` contract, however, if `totalWeight` in the `Voter` contract is zero (knowing that `totalWeight` is changing when users vote and reset), this causes a division by zero, which leads to a failure of the `updatePeriod()` function:

```
function updatePeriod() external returns (uint256) {

        //...
        if (toVoter > 0) {
            PEG.approve(address(VOTER), toVoter);
            VOTER.notifyRewardAmount(toVoter);
        }

        if (toLockers > 0) {
            address(PEG).safeTransfer(address(REWARDS_DISTRIBUTOR), toLockers);
            REWARDS_DISTRIBUTOR.checkpoint_token();
            REWARDS_DISTRIBUTOR.checkpoint_total_supply();
        }

        if (toTeam > 0) {
            address(PEG).safeTransfer(team, toTeam);
        }

      //...
    }
```

```
// Voter contract:
 function notifyRewardAmount(uint256 amount) external {
        _safeTransferFrom(base, msg.sender, address
        //(this), amount); // transfer the distro in

        // @audit : revert due div by zero
        uint256 _ratio = amount * 1e18 / totalWeight; // 1e18 adjustment is
        // removed during claim
        if (_ratio > 0) {
            index += _ratio;
        }
        emit NotifyReward(msg.sender, base, amount);
    }
```

This prevents the rewards from being distributed to both the `RewardDistributor` contract and the `team` address until `totalWeight` in the `Voter` contract becomes greater than zero.

Recommendations:

1. Update the `VOTER.notifyRewardAmount()` function to prevent a revert when `totalWeight` is zero.
2. Modify the `updatePeriod()` function to ensure that the rewards are still distributed to the `RewardDistributor` and `team` even if `totalWeight` is zero (and refund `toVoter` back).

[L-16] `Voter` rewards for dead gauges become stuck due to missing transfer

In the `Voter` contract, the `_updateFor()` function is responsible for updating the rewards of a gauge based on its vote weight, which is recorded in the `claimable` mapping. The function checks if the gauge is **alive** before assigning the rewards, ensuring that only active gauges receive their entitled rewards.

However, a significant issue arises when the gauge is dead. While the `index` is updated in the `notifyRewardAmount()` function and increased based on the total voting weight (`totalWeight`), which accounts for the weights of both active and dead gauges, the rewards for dead gauges are not properly handled in the `_updateFor()` function.

So when a gauge is dead, the calculated rewards (`shares`) are not assigned to it, and the function does not transfer the rewards back to the minter (or any other authorized address), as a result, these entitled rewards for dead gauges are **stuck in the contract**, never being distributed or transferred.

```
function _updateFor(address _gauge) internal {
        //...
        if (_supplied > 0) {
            //...
            if (_delta > 0) {
                uint256 _share = uint256
                //(_supplied) * _delta / 1e18; // add accrued difference for each supp
                if (isAlive[_gauge]) {
                    claimable[_gauge] += _share;
                }
            }
        } else {
            supplyIndex[_gauge] = index; // new users are set to the default
            // global state
        }
    }
```

The `_updateFor()` function is called in both the `_reset()` , `vote()` , `updateFor()` and `updateForRanges()` functions, which means that dead gauges are still processed in these functions, but without transferring their rewards to any other authorized address.

Update the `_updateFor()` function to ensure that when a gauge is dead, the calculated rewards are transferred to the minter or another authorized address.

[L-17] Missing `minAmountCollateral` in `liquidate()`

In the `LiquidationManager` contract, the `liquidate()` function lacks a `minAmountCollateral` parameter, which exposes the liquidator to potential losses if the price of the collateral is manipulated or spikes drastically, where the liquidator may receive less collateral in return for the assets they have repaid, which makes it less tempting for liquidators to engage with the protocol and liquidate unhealthy positions.

Update the `liquidate()` function to include a `minAmountCollateral` parameter, which allows the liquidator to specify the minimum amount of collateral they expect to receive.

[L-18] Vault is susceptible to inflation attack by first depositor

The `Vault` contract is vulnerable to an inflation attack where the first user to deposit can manipulate the share valuation, allowing him to redeem a disproportionate amount of assets.

Consider minting an initial share amount (minting dead shares) to a dead address (`address(0)`):

```
constructor(address _asset, address _priceFeed, uint256 _mcr) ERC4626
      (IERC20(_asset)) ERC20("", "") {
        _initializeOwner(msg.sender);
        priceFeed = IPriceFeed(_priceFeed);
        MCR = _mcr;
+       _mint(address(0),1e3);
    }
```

[L-19] `Vault.sharePrice` incorrectly assumes 18 decimals for every vault

The `Vault.sharePrice()` function incorrectly assumes 18 decimals when calculating share prices by using a hardcoded `1e18` value. This assumption breaks when the underlying asset has different decimals (e.g., `WBTC` with 8 decimals or the vault that introduces `_decimalsOffset()` > 0), leading to overestimation of the sharePrice and potential incorrect share price calculations that affect collateral valuations across the system.

```
//File: src/core/Vault.sol

function sharePrice() external view returns (uint256) {
    return (convertToAssets(1e18) * assetPrice()) / 1e18;
}
```

```
//File: src/core/Vault.sol -> ERC4626.sol

function decimals() public view virtual override
  (IERC20Metadata, ERC20) returns (uint8) {
    return _underlyingDecimals + _decimalsOffset();
}
```

For example, the scenario where the shares vault has not been 1:1 with the assets (eg., has some incentive donation), the process will take 1e18 shares for calculation which can be seen as 1e10 multiplied shares for WBTC vaults and as the rate has not been proportional to 1:1 there are potential incorrect precisions.

However, in combination with other calculation processes, the least precision potentially cancels out from the math, but if the `sharePrice()` itself still poses the over price value.

```
(convertToAssets(1e18) * assetPrice()) / 1e18
vault8 share price: 92344394845405751500000

decimals: 8 + 0 = 8
(convertToAssets(1e8) * assetPrice()) / 1e8 (maintain the returned 18 decimals)
vault8 share price after: 92344393931150300000000
```

Scale the share amount based on share decimals, this approach assume that `assetPrice()` is returned in 18 decimals:

```
function sharePrice() external view returns (uint256) {
-    return (convertToAssets(1e18) * assetPrice()) / 1e18;
+    uint256 oneShare = 10 ** decimals();
+    return (convertToAssets(oneShare) * assetPrice()) / oneShare;
}
```

[L-20] The `missingShares` could lead to liquidate more portions

The `liquidate()` function in the `LiquidationManager.sol` contract has the concept of `missingShares`, which is the number of shares needed to cover the required collateral. The protocol builds `LiquidationBuffer.sol` to cover the missing shares. however, if it can't provide all the missing funds, this logic will get executed

```
// take from vault
        if (missingShares > 0) {
            uint256 totalShares = IVault(vaultData.addr).totalSupply();
            if (totalShares > 0) {

                                     uint256 sharesToTake = totalShares > missingShares ?
                uint256 assetsToTake = IVault(vaultData.addr).convertToAssets
                  (sharesToTake);
                IVault(vaultData.addr).take(_liquidator, assetsToTake);
```

Taking assets like this from the vault will affect other users' collaterals because it directly affects the price calculation in `Vault.sol#sharePrice()`, which could lead to liquidating them also. It's better to have a treasury that can supply the buffer contract.