Contents

1. Summary
2. Engagement Overview
3. Risk Classification
4. Vulnerability Summary
5. Findings
6. Disclaimer

Summary

About 0xKato and 0xWeiss

0xKato is an independent security researcher and Security Lead at Espresso Systems, specializing in auditing DeFi protocols and blockchain infrastructure. He contributes to the ecosystem through in-depth security reviews and educational resources. Reach out on Twitter @0xkato.

0xWeiss is an independent security researcher. Having found numerous security vulnerabilities in various defi protocols, he does his best to contribute to the blockchain ecosystem and its protocols by putting time and effort into security research & reviews. Reach out on Twitter @0xWeisss

About Hyperstable

Hyperstable is a fully on-chain, over-collateralized stablecoin protocol in which anyone can mint USDH by depositing supported assets into non-custodial vaults that use dynamically adjusted interest rates to keep the peg tight. Its governance token PEG emits most of its supply as liquidity incentives, and when locked into vePEG grants holders voting power over emissions and a share of protocol revenues. Permissionless liquidations enforce each vault’s collateral ratio, with all fees flowing back to vePEG stakers to safeguard solvency .

Engagement Overview

Hyperstable engaged 0xWeiss & 0xKato to review the security of its codebase. From the 12th of March to the 26th of March, 0xWeiss & 0xKato reviewed the source code in scope. At
the end, there were 14 issues identified. All findings have been recorded in the following report. Notice that the examined smart contracts are not resistant to internal exploit. For a detailed understanding of risk severity, source code vulnerability, and potential attack vectors, refer to the complete audit report below.

The following repositories were reviewed at the specified commits:

Repository	Commit
hyperstable/contracts	`75974171a452398f356b4025f81b3bbcce9ec6d3`

Risk Classification

Severity	Description
Critical	Vulnerabilities that lead to a loss of a significant portion of funds of the system.
High	Exploitable, causing loss or manipulation of assets or data.
Medium	Risk of future exploits that may or may not impact the smart contract execution.
Low	Minor code errors that may or may not impact the smart contract execution.
Informational	Non-critical observations or suggestions for improving code quality, readability, or best practices.

Vulnerability Summary

Severity	Count	Fixed	Acknowledged
critical	1	1	0
high	3	3	0
medium	6	5	1
low	1	0	1
inf	3	0	3

Findings

Index	Issue Title	Status
C-01	Anyone can delegate other users positions	Fixed
H-01	Slope and bias are miscalculated for perpetual locks	Fixed
H-02	Delegations delegate both, perpetual and non-perpetual locks.	Fixed
H-03	Miss-accounting of perpetuallyLockedBalance on deposits	Fixed
M-01	missing threshold on liquidations	Fixed
M-02	Liquidations may incur slippage	Acknowledged
M-03	Updating the liquidation manager is corrupted	Fixed
M-04	Excess claiming lacks slippage control	Fixed
M-05	Change in interest rate will miss-calculate debt	Fixed
M-06	Perpetual locks will DoS multiple functionalities	Fixed
L-01	Initialization can happen more than once	Acknowledged
I-01	Price feed implementation is unfinished	Acknowledged
I-02	Architectural improvements and global patterns	Acknowledged
I-03	Un-used code across the repository	Acknowledged

Detailed Findings

Critical Risk

C-01 - Anyone can delegate other users positions

Severity: critical

Description

The `delegate` function has no restrictions on msg.sender, allowing anyone to specify an arbitrary `from` address:

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

This allows to delegate from other users without their permission.

PoC

```
function test_delegate_not_validated() external {
        uint256 aliceAmountToLock = 100e18;
        uint256 aliceLockDuration = 365 days;

        vm.startPrank(ALICE);

        peg.approve(address(ve), aliceAmountToLock);
        uint256 aliceLockId = ve.create_lock(aliceAmountToLock,
aliceLockDuration);
        ve.lock_perpetually(aliceLockId);

        vm.stopPrank();

        uint256 bobAmountToLock = 200e18;
        uint256 bobLockDuration = 365 days / 2;

        vm.startPrank(BOB);

        peg.approve(address(ve), bobAmountToLock);
        uint256 bobLockId = ve.create_lock(bobAmountToLock,
bobLockDuration);
```

```
        vm.stopPrank();

        assertEq(ve.perpetuallyLockedBalance(), aliceAmountToLock);
        assertEq(ve.ownerOf(aliceLockId), ALICE);
        assertEq(ve.ownerOf(bobLockId), BOB);

        uint256 aliceLocked = ve.balanceOfNFT(aliceLockId);
        uint256 bobLocked = ve.balanceOfNFT(bobLockId);

        assertEq(peg.balanceOf(address(ve)), aliceAmountToLock +
bobAmountToLock);
        assertEq(ve.totalSupply(), aliceLocked + bobLocked);

        vm.warp(vm.getBlockTimestamp() + 1 hours);

        assertEq(ve.balanceOfNFT(aliceLockId), aliceLocked);
        assertLt(ve.balanceOfNFT(bobLockId), bobLocked);
        assertEq(ve.perpetuallyLockedBalance(), aliceAmountToLock);

        aliceLocked = ve.balanceOfNFT(aliceLockId);
        bobLocked = ve.balanceOfNFT(bobLockId);

        vm.warp(vm.getBlockTimestamp() + 365 days);
        vm.stopPrank();
        console.log("allice address", ALICE);
        console.log("bob address", BOB);
        assertEq(ve.delegates(ALICE), ALICE);
        assertEq(ve.delegates(BOB), BOB);

        // 0xdead delegates from alice to bob
        vm.startPrank(address(0xdead));
        ve.delegate(aliceLockId, bobLockId);
        vm.stopPrank();

        assertEq(ve.delegates(ALICE), BOB);
    }
```

Recommendation

Adopt the original Velodrome V1 implementation of delegations, or access control the function in a certain way that does not allow for this scenario to happen.

Developer Response

Fixed at commit:
https://github.com/hyperstable/contracts/commit/7466285da514c96e755d72366dcee29d9d

High Risk

H-01 - Slope and bias are miscalculated for perpetual locks

Severity: high

Description

When creating a perpetual lock, the end time of the new lock is set to 0.

```
LockedBalance memory newLock;
        newLock.end = 0;
        newLock.perpetuallyLocked = true;
        newLock.amount = currentLock.amount;

        perpetuallyLockedBalance += amount;

        _checkpoint(_tokenId, currentLock, newLock);
```

After updating the new lock, `_checkpoint` is called, which interprets the lock as expired as `new_locked.end < block.timestamp` because the end time is reset to 0:

```
if (new_locked.end > block.timestamp && new_locked.amount > 0) {
                u_new.slope = new_locked.amount / iMAXTIME;
                u_new.bias = u_new.slope * int128(int256(new_locked.end -
block.timestamp));
            }
```

Therefore the `slope` and `bias` values are miscalculated.

Remedation

Perpetually locked positions should have the same weight forever. This could be fixed by counting the end time as MAX_TIME on checkpoints always for perpetual locks.

Developer Response

Fixed at commit:
https://github.com/hyperstable/contracts/commit/0adeb0ca27a6945c328c166c6d7dad1504

H-02 - Delegations delegate both, perpetual and non-perpetual locks.

Severity: high

Description

When calling `_delegate` to delegate the power from your positions:

```
 function _delegate(uint256 _from, uint256 _to) internal {
        LockedBalance memory currentLock = locked[_from];
        require(currentLock.perpetuallyLocked == true, "Lock is not
perpetual");
        address delegator = ownerOf(_from);
        address delegatee = _to == 0 ? address(0) : ownerOf(_to);
        address currentDelegate = delegates(delegator);
        _delegates[delegator] = delegatee;
        _moveAllDelegates(delegator, currentDelegate, delegatee);
    }
```

it does call `_moveAllDelegates` which loops through all the NFTs owned by the owner:

```
   // Plus all that's owned
                for (uint256 i = 0; i < ownerTokenCount; i++) {
                    uint256 tId = ownerToNFTokenIdList[owner][i];
                    dstRepNew.push(tId);
                }
```

This does not take into count that to delegate, it explicitly checks that the lock is perpetual:

```
    function _delegate(uint256 _from, uint256 _to) internal {
        LockedBalance memory currentLock = locked[_from];

        require(currentLock.perpetuallyLocked == true, "Lock is not
perpetual");
```

Otherwise the delegation does not happen.

Recommendation

Seems like the architecture is not correct as it loops through all the owners tokens while it passes ids. Re-think whether it is important to only allow perpetual locks to be delegated, otherwise the architecture needs to change to only delegate the positions that are perpetually locked, or all the positions from that owner.

Developer Response

Fixed at commit:
https://github.com/hyperstable/contracts/commit/7466285da514c96e755d72366dcee29d9c

H-03 - Miss-accounting of perpetuallyLockedBalance on deposits

Severity: high

Description

In the `deposit_for` function, it anyone can deposit tokens for certain tokenId, which their are added to the lock. This functionality is also meant to be used with perpetual locks:

```
 function deposit_for(uint256 _tokenId, uint256 _value) external
nonreentrant {
        LockedBalance memory _locked = locked[_tokenId];

        require(_value > 0); // dev: need non-zero value
        require(_locked.amount > 0, "No existing lock found");
        require(_locked.end > block.timestamp, "Cannot add to expired lock.
Withdraw");
        _deposit_for(_tokenId, _value, 0, _locked,
DepositType.DEPOSIT_FOR_TYPE);
    }
```

When someone deposits for a perpetual lock, the global `perpetuallyLockedBalance` variable should be incremented respectively.

Recommendation

Increment `perpetuallyLockedBalance` when calling `deposit_for` externally for perpetual locks.

Developer Response

Fixed at commit:
https://github.com/hyperstable/contracts/commit/9f6591bc3864e2365e9e9ac50f0bcd86efcb

Medium Risk

M-01 - missing threshold on liquidations

Severity: medium

Description

In the current _liquidate flow, it's possible for a liquidation to leave behind dust amounts of debt or collateral (e.g., a few wei). This opens the vector for positions to remain open with negligible value, which might need to get cleaned up after extreme price movements.

Remedation

Enforce that liquidations either fully close a position, or ensure that any remaining debt is above some minimum threshold on a partial liquidation.

Developer Response

Fixed at commit:
https://github.com/hyperstable/contracts/commit/cb30a683971974f6205c453a0d172a5431

M-02 - Liquidations may incur slippage

Severity: medium

Description

The `_liquidate` function allows for anyone to liquidated an unhealthy position:

```
function _liquidate(uint8 _index, address _positionOwner, address
_liquidator, uint256 _debtToRepay)
        internal
        returns (LiquidationValues memory)
    {
        IPositionManager manager = positionManager;

        (IPositionManager.VaultData memory vaultData,
IPositionManager.PositionData memory positionData) =
            manager.initLiquidation(_index, _positionOwner);

        LiquidationValues memory values = _getLiquidationValues(vaultData,
positionData, _debtToRepay);

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

        // register liquidation
        manager.finishLiquidation(_index, _positionOwner,
values.debtToRepay, values.sharesToLiquidate);

        // burn debt from liquidator
        manager.DEBT_TOKEN().burn(_liquidator, values.debtToRepay);

        // send assets to liquidator
        uint256 rewards =
IVault(vaultData.addr).redeem(values.sharesToRedeem, _liquidator,
address(manager));
```

It then redeems shares from the vault which is fetched using the index passed as a parameter. This redeem process may incur slippage and it is not checked against what would be an acceptable value of rewards redeemed given the shares burned.

Recommendation

Do sum up all rewards in the liquidation process and compare them to an acceptable value of redeemed assets. If less, then revert. Note it is important the accepted slippage is not super tight as liquidations should not revert. Nevertheless, there needs to be protection against asste/share manipulation, specially when the vault doesn't have much activity.

Developer Response

Acknowledged

M-03 - Updating the liquidation manager is corrupted

Severity: medium

Description

The `setLiquidationManager` allows to update the `liquidationManager` address:

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

When registering a vault, the vault approves max to the `liquidationManager` so that the liquidations can go through:

```
 IVault(_vaultAddress).approve(address(liquidationManager),
type(uint256).max);
```

When calling `setLiquidationManager` there are two fundamental problems, that the previous approvals from the `liquidationManager` to the vaults are not reset to 0, and the new `liquidationManager` is not granted approval of the existing vaults.

Recommendation

Reset approval for the old manager and grant approval for the new manager.

Developer Response

Fixed at commit:
https://github.com/hyperstable/contracts/commit/665d860637e13ddaec0ed0ab3071977942

M-04 - Excess claiming lacks slippage control

Severity: medium

Description

The `claimExcess` function allows for the owner to claim the excess over the threshold when notify deposits gets called.

```
    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));
         }

     }
```

The redeem function calculates the amount of assets sent to the owner depending on variables that can be impacted by other users usage. This allows for a worse amount returned than expected when redeeming, also called slippage.

Recommendation

Add a slippage check in the `claimExcess` function

Developer Response

Fixed at commit:
https://github.com/hyperstable/contracts/commit/7530379384c3b2814123288ce35a5249d7

M-05 - Change in interest rate will miss-calculate debt

Severity: medium

Description

The `setInterestRate` allows the owner to suddenly change the interest rate value:

```
 function setInterestRate(uint256 _interestRate) internal {
        setInterestRate(getStorage(), _interestRate);
    }

    function setInterestRate(ManualIRMStorage storage s, uint256
_interestRate) internal {
        s.interestRate = _interestRate;
 }
```

If this is done, there needs to be an accrual of all the debt in the vault before hand so that all the previous debt that was yet not accrued accrues with the correct `_interestRate` value other than the new one.

Otherwise it could cause positions to be liquidable unfairly and users accruing more or less debt than what they should.

Recommendation

Accrue all the debts from the vault inside the `setInterestRate` function before updating the `_interestRate` parameter

Developer Response

Fixed at commit:
https://github.com/hyperstable/contracts/commit/75974171a452398f356b4025f81b3bbcce9

M-06 - Perpetual locks will DoS multiple functionalities

Severity: medium

Description

When someone creates a perpetual lock, the end time of the lock gets reset to 0, meaning the lock acts as expired:

```
      LockedBalance memory newLock;
        newLock.end = 0;
        newLock.perpetuallyLocked = true;
        newLock.amount = currentLock.amount;
```

This is a mistake as it will DoS certain functionalities such as `increase_amount`, given the fact that a timestamp check exists: `require(_locked.end > block.timestamp, "Cannot add to expired lock. Withdraw");` As it is reset to 0, it will revert, while it shouldn't as there is custom logic for perpetual locks at the end of the function:

```
function increase_amount(uint256 _tokenId, uint256 _value) external
nonreentrant {
        assert(_isApprovedOrOwner(msg.sender, _tokenId));

        LockedBalance memory _locked = locked[_tokenId];

        assert(_value > 0); // dev: need non-zero value
        require(_locked.amount > 0, "No existing lock found");

        require(_locked.end > block.timestamp, "Cannot add to expired lock.
Withdraw");

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

Recommendation

For perpetual locks either set an unreachable end date rather than set it to 0. Consider using type(uint64).max as an example for the end date. Or, each time that a perpetual lock is being handled check that is perpetual.

Developer Response

Fixed at commit: https://github.com/hyperstable/contracts/commit/fa26780cccb4edd8ef936ff8fb5d1ba74e46

Low Risk

L-01 - Initialization can happen more than once

Severity: low

Description

The `initialize` is meant to only be called once and set the state variables that are going to be called. The problem is that there is no requirement for this function to only be called once, and `positionManager` and `liquidationBuffer` could be later updated:

```
  function initialize(address _positionManagerAddress, address
_liquidationBufferAddress) external onlyOwner {
        positionManager = IPositionManager(_positionManagerAddress);
        liquidationBuffer = ILiquidationBuffer(_liquidationBufferAddress);
    }
```

Recommendation

Do only allow to call the `initialize` function once.

Developer Response

Acknowledged

Informational

I-01 - Price feed implementation is unfinished

Severity: inf

Description

Currently the final price feeds are not yet implemented, but there is a main functions interacting with them, `sharePrice` which under the hood calls `assetPrice()`

```
  function assetPrice() public view returns (uint256) {
        return priceFeed.fetchPrice();
    }

    function sharePrice() external view returns (uint256) {
        uint256 assets = convertToAssets(1e18) * (10 **
_assetDecimalOffset);
        return assets.mulWad(assetPrice());
    }
```

`assetPrice()` basically fetches a pre-set value by the owner of the contract:

```
  function setPrice(uint256 _newPrice) external onlyOwner {
        _price = _newPrice;
    }

    function fetchPrice() external view returns (uint256) {
        return _price;
    }
```

There are two main problems that need to be taken care of. First, the price can be sandwiched. At this point, you can front-run a price update if its beneficial towards you and sandwich it if needed. Second, there is no validation in regards to staleness of the oracle response and/or invalid responses

Recommendation

Use one of the main price feed providers such as Chainlink or Pyth, or if not available, do use a private rpc for setting the price.

Also, validate staleness of the price feed and invalid responses.

Developer Response

Acknowledged, the oracle will not be implemented yet. The concerns will be followed accordingly when the oracle is implemented

I-02 - Architectural improvements and global patterns

Severity: inf

Description

- The vault contract could have an additional pausable functionality in case of an emergency. The Pausable contract from OZ would be the best choice to implement it.
- There is a global shortage of input validation in multiple functions. Some examples could be:

```
   function setPriceFeed(address _newPriceFeed) external onlyOwner {
        emit NewPriceFeed(address(priceFeed), _newPriceFeed);

        priceFeed = IPriceFeed(_newPriceFeed); //<< validate address(0)
    }


    constructor(address _positionManagerAddress, address
_rewardsDistributorAddress) {
        _initializeOwner(msg.sender);

        POSITION_MANAGER = IPositionManager(_positionManagerAddress); //<<
validate all inputs
        REWARDS_DISTRIBUTOR =
IRewardsDistributor(_rewardsDistributorAddress);
        INTEREST_TOKEN = IERC20(REWARDS_DISTRIBUTOR.token());
    }
```

Skim through all the functions and check that all parameters are validated, even if the validation seems ridiculous.

- Incorrect tracking of events in constructors.

Events must be able to track the variable updates to the initial state backwards. In most constructors it lacks to emit the corresponding event for a state variable update. One example could be the `priceFeed` :

```
 constructor(address _asset, address _priceFeed) ERC4626(ERC20(_asset))
ERC20("", "") {
        _initializeOwner(msg.sender);
+         emit NewPriceFeed(address(priceFeed), _newPriceFeed);
        priceFeed = IPriceFeed(_priceFeed);

        _assetDecimalOffset = 18 - ERC20(_asset).decimals();
    }

    function setPriceFeed(address _newPriceFeed) external onlyOwner {
        emit NewPriceFeed(address(priceFeed), _newPriceFeed);

        priceFeed = IPriceFeed(_newPriceFeed);
    }
```

Skim through all the functions and check for this pattern.

- Consider using the Ownable2Step contract instead than the basic Ownable implementation.

Across the codebase, currently, there is a one-step ownership transfer that is prone to errors. Grep all the contracts using Ownable and update it for the Ownable2Step contract, which allows the pending owner to accept the new ownership role.

Developer Response

Acknowledged

I-03 - Un-used code across the repository

Severity: inf

Description

The following snippets are un-used:

- LiquidationManager

```
    error OnlyPositionManager();
     error NothingToLiquidate();
     error NotEnoughRewards();
```

- VotingEscrow

```
import {IVotes} from "@openzeppelin/contracts/governance/utils/IVotes.sol";
import {IERC165} from
"@openzeppelin/contracts/utils/introspection/IERC165.sol";
import {IVotingEscrow} from "../interfaces/IVotingEscrow.sol";
```

- PegAirdrop

```
error RecoverBeforeDeadline();
```

- ExternalBribe

```
import {IGauge} from "../interfaces/IGauge.sol";
```

- LiquidationBuffer

```
import {ERC20} from "solady/tokens/ERC20.sol";
```

- PositionManager

```
    error NothingToLiquidate();
```

- InterestRateStrategyV1

```
import {IPriceFeed} from "../interfaces/IPriceFeed.sol";
import {Ownable} from "solady/auth/Ownable.sol";
```

- Gauge

```
import {IBribe} from "../interfaces/IBribe.sol";

   address[8] public stakingRewards;
    uint8 private _stakingRewardsCount;
    mapping(address => bool) public isStakingReward;
```

Recommendation

Remove the previously mentioned instances.

Developer Response

Acknowledged

Disclaimer

This report is not, nor should be considered, an “endorsement” or “disapproval” of any particular project or team. This report is not, nor should be considered, an indication of the economics or value of any “product” or “asset” created by any team or project that contracts 0xKato and 0xWeiss to perform a security assessment. This report does not provide any warranty or guarantee regarding the absolute bug-free nature of the technology analyzed, nor do they provide any indication of the technologies proprietors, business, business model or legal compliance.

This report should not be used in any way to make decisions around investment or involvement with any particular project. This report in no way provides investment advice, nor should be leveraged as investment advice of any sort. This report represents an extensive assessing process intending to help our customers increase the quality of their code while reducing the high level of risk presented by cryptographic tokens and blockchain technology.

Blockchain technology and cryptographic assets present a high level of ongoing risk. Our position is that each company and individual are responsible for their own due diligence and continuous security. Our goal is to help reduce the attack vectors and the high level of variance associated with utilizing new and consistently changing technologies, and in no way claims any guarantee of security or functionality of the technology we agree to analyze.

Therefore, we do not guarantee the explicit security of the audited smart contract, regardless of the verdic.