Three Sigma

Code Audit



Thunderhead Labs stHype

Disclaimer

Code Audit

Thunderhead Labs stHype

Disclaimer

The ensuing audit offers no assertions or assurances about the code's security. It cannot be deemed an adequate judgment of the contract's correctness on its own. The authors of this audit present it solely as an informational exercise, reporting the thorough research involved in the secure development of the intended contracts, and make no material claims or guarantees regarding the contract's post-deployment operation. The authors of this report disclaim all liability for all kinds of potential consequences of the contract's deployment or use. Due to the possibility of human error occurring during the code's manual review process, we advise the client team to commission several independent audits in addition to a public bug bounty program.

Table of Contents

Code Audit

Thunderhead Labs stHype

Table of Contents

Disclaimer **3**
Summary **8**
Scope **10**
Methodology **12**
Project Dashboard **14**
Code Maturity Evaluation **17**
Findings **20**
3S-Thunderhead Labs-L01 20
3S-Thunderhead Labs-L02 23
3S-Thunderhead Labs-L03 24
3S-Thunderhead Labs-N01 26
3S-Thunderhead Labs-N02 27
3S-Thunderhead Labs-N03 28

Summary

Code Audit

Thunderhead Labs stHype

Summary

Three Sigma audited Thunderhead Labs's stHype and stElx in a 2.4 person week engagement. The audit was conducted from 11/02/2025 to 18/02/2025.

Protocol Description

Thunderhead is a modular liquid staking protocol enabling networks to deploy Ethereum-grade LST ecosystems with decentralized operator sets, deep liquidity, and instant staking rewards. Supports governance, non-custodial staking, and price aggregation for efficient LST trading.

Scope

Code Audit

Thunderhead Labs stHype

Scope

Filepath	nSLOC
src/Overseer.sol	235
src/Ownership.sol	12
src/stHYPE.sol	261
src/tStorage.sol	22
src/wstHYPE.sol	92
Total	**622**

Assumptions

The OpenZeppelin and Solmate libraries are considered secure.

Methodology

Code Audit

Thunderhead Labs stHype

Methodology

To begin, we reasoned meticulously about the contract's business logic, checking security-critical features to ensure that there were no gaps in the business logic and/or inconsistencies between the aforementioned logic and the implementation. Second, we thoroughly examined the code for known security flaws and attack vectors. Finally, we discussed the most catastrophic situations with the team and reasoned backwards to ensure they are not reachable in any unintentional form.

Taxonomy

In this audit we report our findings using as a guideline Immunefi's vulnerability taxonomy, which can be found at immunefi.com/severity-updated/. The final classification takes into account the severity, according to the previous link, and likelihood of the exploit. The following table summarizes the general expected classification according to severity and likelihood; however, each issue will be evaluated on a case-by-case basis and may not strictly follow it.

Severity / Likelihood	LOW	MEDIUM	HIGH
NONE	None		
LOW	Low		
MEDIUM	Low	Medium	Medium
HIGH	Medium	High	High
CRITICAL	High	Critical	Critical

Project Dashboard

Code Audit

Thunderhead Labs stHype

Project Dashboard

Application Summary

Name	Thunderhead Labs
Commit	8e72f83
Language	Solidity
Platform	Hyperliquid

Engagement Summary

Timeline	11/02/2025 to 18/02/2025
№ of Auditors	2
Review Time	6 days

Vulnerability Summary

Issue Classification	Found	Addressed	Acknowledged
Critical	0	0	0
High	0	0	0
Medium	0	0	0
Low	3	2	1

None	3	3	0

Category Breakdown

Suggestion	3
Documentation	0
Bug	3
Optimization	0
Good Code Practices	0

Findings

Code Audit

Thunderhead Labs stHype

Findings

3S-Thunderhead Labs-L01

Inflation attack via minting to zero address in **`stHYPE`** contract

Id	3S-Thunderhead Labs-L01
Classification	Low
Severity	Medium
Likelihood	Low
Category	Bug
Status	Addressed in #536af6e.

Description

The **`stHYPE`** contract functions similarly to an ERC-4626 vault, incorporating a virtual offset mechanism that introduces a six-decimal difference between assets and shares. However, it does not implement virtual shares. In the current deployment, the contract is susceptible to an inflation attack if an attacker can manipulate the price of a share.

This vulnerability arises due to two primary issues. First, the **`mint`** function does not prevent minting to the zero address. Second, in the **`_transfer`** function of **`stHYPE`**, if both **`from`** and **`to`** are the zero address, the **`preSyncSupply`** variable increases by **`amount`** due to the placement of the **`if (to == address(0))`** condition inside an else statement. However, the **`_transferVotingUnits`** function does not increase the total shares in this scenario. Consequently, minting to the zero address inflates the price of a share, enabling an attacker to front-run legitimate transactions and extract value maliciously.

```
function _transfer(address from, address to, uint256 amount) internal {
    uint256 shares = _balanceToShares(amount);
    if (from == address(0)) {
        preSyncSupply += SafeCast.toUint96(amount);
    } else if (to == address(0)) {
        preSyncSupply -= SafeCast.toUint96(amount);
    }
    _transferVotingUnits(from, to, shares);
    emit Transfer(from, to, amount);
```

```
    }
```

While the contract includes a **notTransferToZeroAddress** modifier to prevent transfers to the zero address, this modifier is not applied to the **mint** function:

```
    modifier notTransferToZeroAddress(address to) {
        if (to == address(0)) {
            revert TransferToZeroAddress();
        }
        _;
    }
/**
     * `
     * @notice Mints new tokens, increasing totalSupply, initSupply, and a
users balance.
     * @dev Limited to onlyMinter modifier
     */
    function mint(
        address to,
        uint256 amount
    ) external onlyRole(MINTER_ROLE) notMintPaused returns (bool) {
        _mint(to, amount);
        return true;
    }
    /**
     * Mint functions
     * @param to Address to mint to
     * @param amount Amount to mint
     */
    function _mint(address to, uint256 amount) internal {
        _transfer(address(0), to, amount);
        emit Mint(to, amount);
    }
```

An attacker can exploit this by minting a minimum share before significantly inflating the price of a share. If a user subsequently attempts to mint, the higher price per share may result in the mint transaction yielding zero shares, effectively transferring their assets to the attacker. The exploit enables the attacker to burn their artificially inflated shares and redeem a disproportionate amount of assets, causing loss to the legitimate minter.

Although the virtual offset mitigates the impact to some extent by requiring a large initial capital, the exploit remains viable.

Recommendation

The contract should explicitly revert transactions where **`mint`** is called with **`to == address(0)`**. Apply the **`notTransferToZeroAddress`** modifier to the **`mint`** function:

```
function mint(
        address to,
        uint256 amount
    ) external onlyRole(MINTER_ROLE) notMintPaused
notTransferToZeroAddress(to) returns (bool) {
        _mint(to, amount);
        return true;
    }
```

3S-Thunderhead Labs-L02

Rounding in **`_transfer`** favors caller on burning, leading to underestimated burned shares

Id	3S-Thunderhead Labs-L02
Classification	Low
Severity	Low
Likelihood	Medium
Category	Bug
Status	Acknowledged

Description

In tokenized protocols, rounding should generally favor the protocol rather than the caller to prevent potential exploitation. However, in the **`stHYPE._transfer`** function, the conversion of balance to shares is always rounded down using **`_balanceToShares`**.

This behavior creates an issue during burns:

- When tokens are burned, the corresponding **`shares`** are rounded down, meaning fewer shares are deducted than expected.

- This results in an underestimation of burned shares, benefiting the caller at the protocol's expense.

Recommendation

- Modify **`_balanceToShares`** to support both rounding up and rounding down.

- In **`_transfer`**:

  - Round **`shares`** up when burning.

  - Round **`shares`** down when minting.

3S-Thunderhead Labs-L03

Unbounded loop in **getBurnIds** and **getBurns** can cause view function failure

Id	3S-Thunderhead Labs-L03
Classification	Low
Severity	Low
Likelihood	Medium
Category	Bug
Status	Addressed in #3b9ee81.

Description

In the **Overseer** contract, the functions **getBurnIds** and **getBurns** are used to retrieve burn information for a given address. These functions rely on an internal function, **_getBurnIds**, which iterates over an unbounded storage array called **burns**.

Although **getBurnIds** and **getBurns** are view functions, they are still subject to gas limits. If the **burns** array grows too large, the gas cost of iterating through it may exceed the limit, causing these functions to fail. This can make it impossible to retrieve burn history on-chain for large datasets.

Recommendation

- Introduce a mapping to track burn IDs per address:

```
  + mapping(address => uint256[]) private accountToIds;
```

- In the **burn** function, append new burn IDs to this mapping:

```
  function burn(address to, uint256 amount, string memory code) public
returns (uint256) {
      sthype.burn(msg.sender, amount);
      burns.push(Burn(SafeCast.toUint88(amount), to, false, amount +
burns[burns.length - 1].sum));
+     // Track the burn ID for the sender
```

```
+     accountToIds[msg.sender].push(burns.length - 1);
      emit Burn_(msg.sender, to, amount, burns.length - 1,
bytes32(bytes(code)));
      return burns.length - 1;
  }
```

- Modify **_getBurnIds** to return the pre-tracked burn IDs instead of iterating over burns:

```
function _getBurnIds(address account) internal view returns (uint256[]
memory) {
-     uint256[] memory burnIds = new uint256[](burns.length);
-     uint256 t;
-
-     uint256 burnsLength = burns.length;
-     for (uint256 i; i < burnsLength; ++i) {
-         if (burns[i].user == account) {
-             burnIds[t] = i;
-             t++;
-         }
-     }
-
-     uint256[] memory filteredBurnIds = new uint256[](t);
-     for (uint256 i; i < t; ++i) {
-         filteredBurnIds[i] = burnIds[i];
-     }
-
-     return filteredBurnIds;
+     return accountToIds[account];
  }
```

3S-Thunderhead Labs-N01

Missing event emission in Pause functions

Id	3S-Thunderhead Labs-N01
Classification	None
Category	Suggestion
Status	Addressed in #24de282.

Description

The functions **`pauseTransfer`**, **`pauseMint`**, **`pauseBurn`**, and **`pauseRebase`** are designed to control the operational status of stHYPE token's transfer, minting, burning, and rebase functionalities, respectively. They currently lack event emissions, which are important for providing transparency and traceability of state changes on the blockchain. Without emitting events, it becomes challenging for off-chain systems and users to track when these critical operations are paused or resumed.

Recommendation

It is recommended to emit events within the **`pauseTransfer`**, **`pauseMint`**, **`pauseBurn`**, and **`pauseRebase`** functions.

3S-Thunderhead Labs-N02

Inconsistency between **`maxRedeemable`** and **`_redeemable`** functions regarding protocol fees

Id	3S-Thunderhead Labs-N02
Classification	None
Category	Suggestion
Status	Addressed in #c0494ec.

Description

In the **`Overseer`** contract, the **`maxRedeemable`** function is designed to calculate the maximum amount of tokens that can be redeemed for burns. It does this by ensuring that the contract's balance is greater than the sum of the total pending burns and the pending protocol fees. This effectively prioritizes the protocol fees over burn redemptions. On the other hand, the **`_redeemable`** function, which determines if a specific burn is redeemable, does not account for the pending protocol fees in its checks. This inconsistency could lead to confusion, as the **`maxRedeemable`** function is stricter than **`_redeemable`**. However, due to the call flow in the **`burnAndRedeemIfPossible`** function, where **`maxRedeemable`** is checked before **`_redeemable`**, no reversion occurs. Despite this, it is advisable to align the behavior of these two functions to ensure consistency and clarity in the contract's logic.

Recommendation

To ensure consistency between the **`maxRedeemable`** and **`_redeemable`** functions, it is advised to either update the **`_redeemable`** function to account for pending protocol fees, similar to **`maxRedeemable`**, or adjust **`maxRedeemable`** to exclude protocol fees, aligning it with **`_redeemable`**. Choose the approach that best fits the desired behavior for prioritizing burn redemptions versus protocol fees.

3S-Thunderhead Labs-N03

Lack of input validation in setter functions

Id	3S-Thunderhead Labs-N03
Classification	None
Category	Suggestion
Status	Addressed in #b7a8607.

Description

The setter functions **`setAprThreshold`**, **`setSlashThreshold`**, **`changeProtocolFee`**, **`setFreeStakePercentage`**, and **`setInterimAddress`** in the **`Overseer`** contract are designed to update critical parameters and addresses within the contract. However, they currently lack input validation, which could lead to potential issues such as setting parameters to out of bound values or assigning critical addresses to the zero address due to human error.

Recommendation

Implement input validation checks within each setter function to ensure that the provided values are within acceptable bounds. For address parameters, ensure that they are not set to the zero address.