Pashov Audit Group

stHYPE Security Review

October 13th 2025 - October 16th 2025

Contents

1. About Pashov Audit Group

Pashov Audit Group consists of 40+ freelance security researchers, who are well proven in the space - most have earned over $100k in public contest rewards, are multi-time champions or have truly excelled in audits with us. We only work with proven and motivated talent.

With over 300 security audits completed — uncovering and helping patch thousands of vulnerabilities — the group strives to create the absolute very best audit journey possible. While 100% security is never possible to guarantee, we do guarantee you our team's best efforts for your project.

Check out 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. Risk Classification

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

Impact

- **High** - leads to a significant material loss of assets in the protocol or significantly harms a group of users
- **Medium** - leads to a moderate material loss of assets in the protocol or moderately harms a group of users
- **Low** - leads to a minor material loss of assets in the protocol or harms a small group of users

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

4. About stHYPE

stHYPE is a Liquid Staking Protocol built for Hyperliquid's HyperEVM that allows users to stake native HYPE tokens and receive stHYPE, a liquid staking derivative that accrues rewards while remaining tradable. It features modular staking through the Overseer contract, a rebasing stHYPE token, and a wrapped non-rebasing wstHYPE version for DeFi integrations.

5. Executive Summary

A time-boxed security review of the **ValantisLabs/sthype-contracts** repository was done by Pashov Audit Group, during which **Shaka**, **Tejas Warambhe**, **iamnmt**, **IvanFitro** engaged to review **stHYPE**. A total of **9** issues were uncovered.

Protocol Summary	
Project Name	stHYPE
Protocol Type	Liquid Staking
Timeline	October 13th 2025 - October 16th 2025

Review commit hash:
- 19e5a53195be079f6764b4153fcc591b51e2bb97 (ValantisLabs/sthype-contracts)

Fixes review commit hash:
- e97f932c307104c6799bea0c10c45a9da552f14f (ValantisLabs/sthype-contracts)

Scope

`Overseer.sol` `HyperCoreUtils.sol` `HyperCoreStakingModule.sol`

6. Findings

Findings count

Severity	Amount
Medium	4
Low	5
Total findings	**9**

Summary of findings

ID	Title	Severity	Status
[M-01]	Incorrect management of redeemable funds	Medium	Resolved
[M-02]	Deposits might fail if staking module account is not activated	Medium	Resolved
[M-03]	Outdated precompile data might cause miscalculation of total supply	Medium	Resolved
[M-04]	`rebase()` can be blocked by large supply increase	Medium	Resolved
[L-01]	APR can be overestimated by up to 2 basis points	Low	Resolved
[L-02]	Deposit blockage by liability increase may leave dust	Low	Acknowledged
[L-03]	If `aprChange = 0`, `rebase()` will always revert	Low	Resolved
[L-04]	`protocolFee` change without `rebase()` leads to incorrect fees	Low	Resolved
[L-05]	APR calculated before fees in `rebase()`, inflates value	Low	Resolved

Medium findings

[M-01] Incorrect management of redeemable funds

Severity

Impact: Medium

Likelihood: Medium

Description

`Overseer._redeemable()` checks if a burn can be redeemed by making sure that the contract has enough balance to cover the burn amount minus already redeemed amounts and pending protocol fees.

```
    function _redeemable(uint256 burnId) internal view returns (bool) {
        uint256 sum = burns[burnId].sum;
        uint256 difference = sum < redeemed ? 0 : sum - redeemed;
        return burns[burnId].completed == false && difference + protocolPendingFee <=
address(this).balance;
    }
```

This calculation assumes that any redemption with a higher burn ID has left enough balance for all previous burns to be redeemed, which is correct. However, it does not take into account that after that, new fees may have been accrued and withdrawn, which would leave an insufficient balance for the earlier burns to be redeemed.

Proof of concept

```
function test_audit_redeemReverts() public {
    // Setup
    _addStakingModule();
    address[] memory stakingModules = overseer.getStakingModules();
    address stakingModule = stakingModules[0];

    address user1 = makeAddr("user1");
    address user2 = makeAddr("user2");
    address user3 = makeAddr("user3");
    vm.deal(user1, 1 ether);
    vm.deal(user2, 1 ether);
    vm.deal(user3, 1 ether);

    vm.startPrank(gov);
    overseer.setSyncInterval(84_600);
    overseer.grantRole(overseer.FEE_RECIPIENT_ROLE(), address(this));
    vm.stopPrank();

    // Users mint stHYPE
    vm.prank(user1);
```

```
    overseer.mint{value: 1 ether}(user1);

    vm.prank(user2);
    overseer.mint{value: 1 ether}(user2);

    vm.prank(user3);
    overseer.mint{value: 1 ether}(user3);

    // Deposit to staking module
    Overseer.DepositInput[] memory inputs = new Overseer.DepositInput[](1);
    inputs[0] = Overseer.DepositInput({index: 0, amount: 1 ether, data: ""});
    vm.prank(gov);
    overseer.deposit(inputs);
    CoreSimulatorLib.nextBlock();

    // Simulate the effect of staking to validators
    stakingModule.call{value: 0.01 ether}("");

    // User 1 burns
    vm.prank(user1);
    uint256 user1BurnId = overseer.burn(user1, 1 ether, "");

    // User 2 burns and redeems
    vm.prank(user2);
    overseer.burnAndRedeemIfPossible(user2, 1 ether, "");

    // Rebase accrues protocol fees
    vm.warp(block.timestamp + 84_600);
    vm.prank(gov);
    overseer.rebase();

    // Protocol fees are claimed
    overseer.claimFee(0, true);

    // User 1 burn is supposed to be redeemable
    assertTrue(overseer.redeemable(user1BurnId));

    // Redemption reverts due to lack of funds
    vm.prank(user1);
    vm.expectRevert(bytes("Transfer failed"));
    overseer.redeem(user1BurnId);
    assertTrue(address(overseer).balance < 1 ether);
}
```

Recommendations

A partial fix would be adding the `burns[burnId].amount <= address(this).balance` condition to the `_redeemable()` function. This will prevent the `_redeemable()` function from returning true if the contract does not have enough balance to cover the specific burn amount.

If it is wanted to guarantee redemptions of a burn ID will always allow all previous burns to be redeemed, then the withdrawal of protocol fees should be adjusted to allow withdrawing only when `burns[latestBurnId].sum - redeemed + protocolPendingFee <= address(this).balance`. Additionally, the `protocolPendingFee` should not be taken into account in the `_redeemable()` function when `burnId < latestBurnId`.

[M-02] Deposits might fail if staking module account is not activated

Severity

Impact: High

Likelihood: Low

Description

On `deposit()`, the staking module sends HYPE to HyperCore's spot balance and then moves it to the staking balance. However, if the staking module account has not been activated in HyperCore (i.e., it has never received funds in the L1), then the deposit will fail silently.

As a result:

- The deposited HYPE will disappear from the total balance, as it is not credited to the spot balance until the account is activated. This will cause the total supply to be lower than expected, similar to the occurrence of a slashing event, leading to different issues, as the share price decreasing and the protocol receiving extra fees once the supply is recovered.
- Once the account is activated, the deposited HYPE will be available in the spot balance. This will require the manager to withdraw the funds to the EVM and then re-deposit them, as there is no function to move funds from spot to staking balance directly.

Recommendations

Add a check in `addStakingModule()` to ensure that the staking module account has been activated in HyperCore, using the `PrecompileLib.coreUserExists()` function.

[M-03] Outdated precompile data might cause miscalculation of total supply

Severity

Impact: High

Likelihood: Low

Description

`Overseer.rebase()` calls the `getNewSupply()` function to determine the total supply of HYPE owned by the protocol. In this calculation, different precompiles are used to get the balances of different addresses in the L1. In this regard, it is important to note that these precompiles return the state from **the start of the current block**.

This means that if previous actions took place in the same block, the total supply might be miscalculated. These actions include the movement of HYPE from and to the L1 by the staking modules or the interim address.

As a result, the total supply can be lower than expected, similar to the occurrence of a slashing event, which will cause different outcomes, as the share price decreases and the protocol receiving extra fees once the supply is recovered.

Recommendations

- Revert in `rebase()` if in the current block the contract has interacted with the staking modules.
- Coordinate movements of HYPE by the `interimAddress` and calls to `rebase()` to avoid them happening in the same block.

[M-04] `rebase()` can be blocked by large supply increase

Severity

Impact: High

Likelihood: Low

Description

`Overseer.rebase()` reverts when the calculated APR change since the last rebase exceeds a predefined threshold (`aprThresholdBps`).

```
        int256 aprChange = _calculateApr(timeElapsed, currentSupply, newSupply);

        if (aprChange > 0) {
            if (aprChange + 1 >= int256(aprThresholdBps)) {
                revert AprTooHigh(aprChange + 1, aprThresholdBps);
```

As `aprThresholdBps` is limited to 10_000 basis points (100%), a large increase in the stHYPE supply might cause the `rebase()` function to be blocked for a long time, or even permanently.

For example, with a rebase interval of one day, an increase by 0.28% of the stHYPE supply will cause the APR to exceed 100% and the `rebase()` function to revert. And the lower the rebase frequency, the smaller the increase in stHYPE supply is required to block the `rebase()` function. While an attacker would be required to donate that amount, if the stHYPE supply decreases over time (e.g., due to users burning their tokens), the amount required to block the `rebase()` function also decreases.

The result would be that the protocol would not be able to adjust the stHYPE supply for the accrued staking rewards.

Recommendations

Remove the restriction for the maximum value of the `aprThresholdBps` parameter in the `setAprThresholdBps()` function. This will allow the governance to prevent the `rebase()` function from being blocked by setting a sufficiently high threshold, which, combined with a large `syncInterval` value, can process a high increase in the stHYPE supply without causing a major impact.

Low findings

[L-01] APR can be overestimated by up to 2 basis points

`Overseer.rebase()` calls the internal function `_calculateApr()` to calculate the APR change since the last rebase. This function rounds up the calculated APR change by adding 1 to the result before returning it. Additionally, in `rebase()`, the check for the APR threshold also adds 1 to the calculated APR change before comparing it to the threshold. This means that the APR can be overestimated by up to 2 basis points.

```
    function rebase() public onlyRole(REBASER_ROLE) {
(...)
        int256 aprChange = _calculateApr(timeElapsed, currentSupply, newSupply);

        if (aprChange > 0) {
    @>      if (aprChange + 1 >= int256(aprThresholdBps)) {
                revert AprTooHigh(aprChange + 1, aprThresholdBps);

(...)

    function _calculateApr(uint256 timeElapsed, uint256 currentSupply, uint256 newSupply)
(...)
            uint256 apr =
                (((newSupply * E18) / currentSupply - E18) * E18) / ((timeElapsed * E18) /
ONE_YEAR) / (E18 / BIPS);

    @>      return int256(apr + 1);
```

It is recommended to remove the `+ 1` adjustment in the `rebase()` function.

[L-02] Deposit blockage by liability increase may leave dust

The `Overseer::deposit()` enables the manager to deposit HYPE into the staking module, which contains sufficient checks to ensure that the contract contains at least a bare minimum balance to fulfill the redemptions and protocol fee claim:

```
    function deposit(DepositInput[] calldata inputs) external onlyRole(MANAGER_ROLE) {
        uint256 numStakingModules = _getOverseerStorage().stakingModules.length();

        for (uint256 i = 0; i < inputs.length; i++) {
            // . . .
            uint256 preBalance = address(this).balance;
            // Cannot deposit more than native token balance minus total liability
            require(input.amount + totalLiability() <= preBalance, "Excessive deposit
amount");        <<@

            IStakingModule(stakingModule).deposit{value: input.amount}(input.data);

            require(address(this).balance + input.amount >= preBalance, "Unexpected balance
```

```
decrease");
        }
    }
```

However, it can lead to a state where the manager is unable to fully utilize the balance, as it can always happen that a burn request is created right before the `deposit()` call, which would increase the `totalLiability()`, and even the last call failing would lead to revert of `deposit()`. Malicious actors can also target the same for blocking deposits. Hence, it can always happen that the entire balance is not being utilized.

It is recommended to use an epoch-based redemption system where the burn requests would not affect the current epoch.

[L-03] If `aprChange = 0`, `rebase()` will always revert

In `rebase()`, `aprThresholdBps` is checked when `aprChange > 0`, and `slashThresholdBps` is checked when `aprChange <= 0`. The issue is that `aprChange` can be 0 if `newSupply == currentSupply` in `_calculateApr()`.

```
function _calculateApr(uint256 timeElapsed, uint256 currentSupply, uint256 newSupply)
        internal
        pure
        returns (int256)
    {
        if (newSupply > currentSupply) {
            uint256 apr =
                (((newSupply * E18) / currentSupply - E18) * E18) / ((timeElapsed * E18) /
ONE_YEAR) / (E18 / BIPS);

            return int256(apr + 1);
        } else {
@>          uint256 supplyDecrease = BIPS - ((newSupply * BIPS) / currentSupply);

            return -int256(supplyDecrease);
        }
    }
```

As mentioned in the documentation, for now, `slashThresholdBps` is set to `0` because slashing is not yet enabled on Hyperliquid. This causes any case where `aprChange = 0` to always revert with the `SupplyDecreaseTooHigh` custom error.

```
if (aprChange > 0) {
            if (aprChange + 1 >= int256(aprThresholdBps)) {
                revert AprTooHigh(aprChange + 1, aprThresholdBps);
            }
        } else {
@>          if (-aprChange >= int256(slashThresholdBps)) {
                revert SupplyDecreaseTooHigh(aprChange);
            }
        }
```

Recommendation: Modify the condition to use `>=` for the `aprThresholdBps` check, or set `slashThresholdBps` to a nonzero value.

[L-04] `protocolFee` change without `rebase()` leads to incorrect fees

`changeProtocolFee()` is used to update the fees applied when `rebase()` is executed.

```
function changeProtocolFee(uint256 fee_) external onlyRole(DEFAULT_ADMIN_ROLE) {
        if (fee_ > BIPS) revert InvalidInput();

        protocolFee = fee_;

        emit ProtocolFeeSet(fee_);
    }
```

The issue is that `rebase()` is not called before changing the fees. This causes the new fees to be applied, instead of the old ones, resulting in an incorrect `protocolFee` for that period.

Recommendations

Call `rebase()` before changing the `protocolFee` to ensure the correct fee is accrued for the current period. To do this, the `DEFAULT_ADMIN_ROLE` must also have the `REBASER_ROLE`, otherwise calling `rebase()` will revert.

```
function changeProtocolFee(uint256 fee_) external onlyRole(DEFAULT_ADMIN_ROLE) {
        if (fee_ > BIPS) revert InvalidInput();

+       rebase();

        protocolFee = fee_;

        emit ProtocolFeeSet(fee_);
    }
```

[L-05] APR calculated before fees in `rebase()`, inflates value

`rebase()` is used to calculate and update the new APR for stakers.

```
function rebase() public onlyRole(REBASER_ROLE) {
        if (sthype.syncEnd() >= block.timestamp) revert SyncNotEnded();

        uint256 timeElapsed = block.timestamp - lastRebaseTime;

        uint256 newSupply = getNewSupply();
        uint256 currentSupply = sthype.totalSupply();

        uint256 protocolFeeIncrease;
        if (newSupply > currentSupply) {
            protocolFeeIncrease = ((newSupply - currentSupply) * protocolFee) / BIPS;
        }
```

```
        if (protocolFeeIncrease > 0) {
            protocolPendingFee += protocolFeeIncrease;
        }

@>      int256 aprChange = _calculateApr(timeElapsed, currentSupply, newSupply);

        if (aprChange > 0) {
            if (aprChange + 1 >= int256(aprThresholdBps)) {
                revert AprTooHigh(aprChange + 1, aprThresholdBps);
            }
        } else {
            if (-aprChange >= int256(slashThresholdBps)) {
                revert SupplyDecreaseTooHigh(aprChange);
            }
        }

        if (protocolFeeIncrease > 0) {
@>          newSupply -= protocolFeeIncrease;
        }

        sthype.syncSupply(newSupply, syncInterval);

        lastRebaseTime = block.timestamp;

        emit Rebase(currentSupply, newSupply, syncInterval, aprChange,
sthype.balancePerShare(), timeElapsed);
    }
```

The issue is that `aprChange` uses `newSupply` before fees are applied, causing the calculated APR to be higher than the actual one. As a result, users will see an inflated APR compared to what they will actually receive.

This can also cause the `aprThresholdBps` to appear reached when, in reality it isn't, because applying the fees would result in a lower value.

Recommendations

To solve the issue, first deduct `protocolFeeIncrease` from `newSupply`, and then calculate `aprChange`.

```
function rebase() public onlyRole(REBASER_ROLE) {
        if (sthype.syncEnd() >= block.timestamp) revert SyncNotEnded();

        uint256 timeElapsed = block.timestamp - lastRebaseTime;

        uint256 newSupply = getNewSupply();
        uint256 currentSupply = sthype.totalSupply();

        uint256 protocolFeeIncrease;
        if (newSupply > currentSupply) {
            protocolFeeIncrease = ((newSupply - currentSupply) * protocolFee) / BIPS;
        }

        if (protocolFeeIncrease > 0) {
            protocolPendingFee += protocolFeeIncrease;
        }
```

```
+       if (protocolFeeIncrease > 0) {
+           newSupply -= protocolFeeIncrease;
+       }

        int256 aprChange = _calculateApr(timeElapsed, currentSupply, newSupply);

        if (aprChange > 0) {
            if (aprChange + 1 >= int256(aprThresholdBps)) {
                revert AprTooHigh(aprChange + 1, aprThresholdBps);
            }
        } else {
            if (-aprChange >= int256(slashThresholdBps)) {
                revert SupplyDecreaseTooHigh(aprChange);
            }
        }

-       if (protocolFeeIncrease > 0) {
-           newSupply -= protocolFeeIncrease;
-       }

        sthype.syncSupply(newSupply, syncInterval);

        lastRebaseTime = block.timestamp;

        emit Rebase(currentSupply, newSupply, syncInterval, aprChange, sthype.balancePerShare(),
timeElapsed);
    }
```