

# Ventuals

## Smart Contract Security Assessment

VERSION 1.1



AUDIT DATES:

October 7th to October 14th, 2025

AUDITED BY:

cccZ  
ether\_sky

## Contents

<b>1</b>	<b>Introduction</b>	<b>2</b>
1.1	About Zenith	3
1.2	Disclaimer	3
1.3	Risk Classification	3
<hr/>		
<b>2</b>	<b>Executive Summary</b>	<b>3</b>
2.1	About Ventials	4
2.2	Scope	4
2.3	Audit Timeline	5
2.4	Issues Found	5
<hr/>		
<b>3</b>	<b>Findings Summary</b>	<b>5</b>
<hr/>		
<b>4</b>	<b>Findings</b>	<b>7</b>
4.1	High Risk	8
4.2	Medium Risk	17
4.3	Low Risk	36
4.4	Informational	44

# 1

## Introduction

### 1.1 About Zenith

Zenith assembles auditors with proven track records: finding critical vulnerabilities in public audit competitions.

Our audits are carried out by a curated team of the industry's top-performing security researchers, selected for your specific codebase, security needs, and budget.

Learn more about us at <https://zenith.security>.

### 1.2 Disclaimer

This report reflects an analysis conducted within a defined scope and time frame, based on provided materials and documentation. It does not encompass all possible vulnerabilities and should not be considered exhaustive.

The review and accompanying report are presented on an "as-is" and "as-available" basis, without any express or implied warranties.

Furthermore, this report neither endorses any specific project or team nor assures the complete security of the project.

### 1.3 Risk Classification

SEVERITY LEVEL	IMPACT: HIGH	IMPACT: MEDIUM	IMPACT: LOW
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

## 2

### Executive Summary

## 2.1 About Ventials

Ventials brings the power of perps to the most innovative private companies in the world, which is a multi-trillion dollar asset class that normal investors have historically been locked out of.

Ventials is built on Hyperliquid's proven perps orderbook infrastructure, using the HIP-3 standard.

Ventials created its own HYPE LST (vHYPE) to raise the HIP-3 stake requirement. vHYPE is a fully transferable ERC20 token, and serves as the claim to the original HYPE plus accrued native staking yield.

## 2.2 Scope

The engagement involved a review of the following targets:

<b>Target</b>	ventuals-contracts
<b>Repository</b>	<a href="https://github.com/ventuals/ventuals-contracts.git">https://github.com/ventuals/ventuals-contracts.git</a>
<b>Commit Hash</b>	cb1b057e33f114cb029edf1de8886fa9d8e25dc2
<b>Files</b>	contracts/* (excluding tests & mocks)

## 2.3 Audit Timeline

<b>October 7, 2025</b>	Audit start
<b>October 14, 2025</b>	Audit end
<b>October 14, 2025</b>	Report published

## 2.4 Issues Found

SEVERITY	COUNT
Critical Risk	0
High Risk	4
Medium Risk	8
Low Risk	4
Informational	5
<b>Total Issues</b>	<b>21</b>

## 3

## Findings Summary

ID	Description	Status
H-1	stakingWithdraw() may fail due to exceeding the withdrawal limit	Resolved
H-2	The timing of spotSend() in claimWithdraw() will make exchangeRate higher	Resolved
H-3	Attackers can bypass inflation attack protection via VHYPE.burn()	Resolved
H-4	The unstaking amount is calculated incorrectly when slashing occurs	Resolved
M-1	finalizeBatch() will fail when the difference between deposits and withdrawals less than 1e10	Resolved
M-2	The first depositor may receive 0 shares if the total balance becomes 0	Resolved
M-3	Malicious users can prevent the finalization of batches	Resolved
M-4	The attackers can delay other users' withdrawals	Acknowledged
M-5	A undervalued exchangeRate may be fetched in queueWithdraw()	Resolved
M-6	Switching to a new validator is not possible if the previous validator has been fully slashed	Resolved
M-7	Slashing should not be applied to a batch if at least one withdrawal from that batch has already been claimed	Resolved
M-8	The totalHypeProcessed value may become inaccurate due to slashing	Resolved
L-1	A validator with delegated HYPE should not be removed	Resolved
L-2	There's an incorrect size check when splitting the input vHYPE amounts	Resolved

ID	Description	Status
L-3	The snapshot rate should be used in the setMinimumStake-Balance function	Resolved
L-4	StakingVaultManager.deposit() does not refund	Acknowledged
I-1	There are unused errors	Resolved
I-2	The withdrawable amount should be 0 if the withdrawal has been canceled	Resolved
I-3	There is an unnecessary check in the finalizeBatch function	Resolved
I-4	There is an unnecessary check in the applySlash function	Resolved
I-5	An unnecessary condition check exists in the resetBatch function	Resolved

# 4

## Findings

### 4.1 High Risk

A total of 4 high risk findings were identified.

[H-1] `stakingWithdraw()` may fail due to exceeding the withdrawal limit

SEVERITY: High

IMPACT: High

STATUS: Resolved

LIKELIHOOD: Medium

#### Target

- [StakingVaultManager.sol#L564-L570](#)

#### Description:

`finalizeBatch()` may call `stakingVault.unstake()` to initiate a withdrawal from HyperCore:Staking to HyperCore:Spot, which will take 7 days to arrive.

```

    } else if (depositsInBatch < withdrawsInBatch) {
        // Not enough deposits to cover all withdraws; we need to
        withdraw some HYPE from the staking vault

        // Unstake the amount not covered by deposits from the staking
        vault
        uint256 amountToUnstake = withdrawsInBatch - depositsInBatch;
        stakingVault.unstake(validator, amountToUnstake.to8Decimals());
    }
    ...
    function unstake(address validator, uint64 weiAmount)
    external onlyManager whenNotPaused {
        require(weiAmount > 0, ZeroAmount());
        require(whitelistedValidators[validator],
        ValidatorNotWhitelisted(validator));
        _undelegate(validator, weiAmount);
        CoreWriterLibrary.stakingWithdraw(weiAmount);
    }

```

According to the [documentation](#), the pending withdrawals cannot exceed 5.



Each address may have at most 5 pending withdrawals in the unstaking queue.

Since each batch has a 1 day interval, if 6 of 7 batches are initiating withdrawals, the last batch withdrawal will fail due to exceeding the limit, causing the user's following claim to fail and user assets loss.

```
function _fetchBatch() internal view returns (Batch memory batch) {
    if (currentBatchIndex == batches.length) {
        // Initialize a new batch at the current index
        // Only enforce timing restriction if this is not the first batch
        if (lastFinalizedBatchTime != 0) {
            // There's a 1 day lockup period after HYPE is staked to a
            // validator, so we enforce a 1 day delay between batches
            require(
                block.timestamp > lastFinalizedBatchTime + 1 days,
                BatchNotReady(lastFinalizedBatchTime + 1 days)
            );
        }
    }
}
```

### Recommendations:

It is recommended to check the number of pending withdrawals in `StakingVault.unstake()`.

```
function unstake(address validator, uint64 weiAmount)
    external onlyManager whenNotPaused {
    require(weiAmount > 0, ZeroAmount());
    require(whitelistedValidators[validator],
        ValidatorNotWhitelisted(validator));
    L1ReadLibrary.DelegatorSummary memory delegatorSummary = L1ReadLibrary.
        delegatorSummary(address(this));
    if (delegatorSummary.nPendingWithdrawals ≥ 5 )
        revert MaxPendingWithdrawals();
    _undelgate(validator, weiAmount);
    CoreWriterLibrary.stakingWithdraw(weiAmount);
}
```

**Ventuals:** Resolved with [Obe099e066....](#)

**Zenith:** Verified.

## [H-2] The timing of `spotSend()` in `claimWithdraw()` will make `exchangeRate` higher

SEVERITY: High

IMPACT: High

STATUS: Resolved

LIKELIHOOD: Medium

### Target

- [StakingVaultManager.sol#L387-L389](#)

### Description:

`claimWithdraw()` will call `spotSend()` to send Hype to the user in HyperCore and increase `totalHypeClaimed`.

```
// NOTE: We don't need to worry about transfer to Core timings here, because
// claimable HYPE is excluded
// from the total balance (via `totalHypeProcessed`)
stakingVault.spotSend(destination, stakingVault.HYPE_TOKEN_ID(),
    hypeAmount.to8Decimals());

withdraw.claimedAt = block.timestamp;
totalHypeClaimed += hypeAmount;
```

`spotSend()` is a CoreWriter action. According to the [documentation](#), CoreWriter actions are processed after EVM block is built.

This means that in the same block, when the transaction after `claimWithdraw()` gets `totalBalance()`, `spotAccountBalance()` has not yet decreased. But due to the increase in `totalHypeClaimed`, `reservedHypeForWithdraws` will decrease, making `totalBalance()` larger than it actually is. If `processBatch()` is called at this point, user withdrawals in the new batch will use a higher `exchangeRate`.

```
function totalBalance() public view returns (uint256) {
    // EVM + Spot + Staking account balances
    uint256 accountBalances = stakingAccountBalance() + spotAccountBalance()
    + stakingVault.evmBalance();

    // The total amount of HYPE that is reserved to be returned to users for
    // withdraws, but is still in
```

```
// under the StakingVault accounts because they have not finished
processing or been claimed
uint256 reservedHypeForWithdraws = totalHypeProcessed
- totalHypeClaimed;

// This might happen right after a slash, before we're able to adjust the
slashed exchange rate for the
// processed withdraws that are waiting for the 7-day withdraw period to
pass. In practice, we would
// pause the contract in the case of a slash, but there could be a small
window of time right after a
// slash where this could happen. So we throw an explicit error in this
case.
require(accountBalances >= reservedHypeForWithdraws,
AccountBalanceLessThanReservedHypeForWithdraws());

return accountBalances - reservedHypeForWithdraws;
}
```

### Recommendations:

One option is to only allow calls to `deposit()` and `processBatch()` in blocks after `claimWithdraw()`.

**Ventuals:** Resolved with [68c7696bf2....](#)

**Zenith:** Verified.

### [H-3] Attackers can bypass inflation attack protection via `VHYPE.burn()`

SEVERITY: High

IMPACT: High

STATUS: Resolved

LIKELIHOOD: Medium

#### Target

- [VHYPE.sol#L25-L27](#)

#### Description:

The protocol sets the minimum deposit amount and the withdrawal requires the owner to set `isBatchProcessingPaused` to false as a way to prevent the attacker from leaving dusty shares in the protocol in inflation attacks.

But the attacker can call `VHYPE.burn()` to bypass the protection.

```
function burn(uint256 amount) public override whenNotPaused {
    _burn(msg.sender, amount);
}
```

The attacker can deposit  $2e18$  Hype and then burn  $2e18 - 1$  vHype, at which point the exchange rate is inflated to  $2e36$ , after which the victim deposits  $1e18$  Hype and receives 0 vHype, and the attacker's 1 wei vHype is worth  $3e18$  Hype.

#### PoC:

```
function test_Inflate() public {
    address victim = makeAddr("victim");
    uint256 depositAmount = 2 * 1e18; // 50k HYPE
    vm.deal(user, depositAmount);
    vm.deal(victim, depositAmount);

    vm.startPrank(user);
    stakingVaultManager.deposit{value: 2e18}();
    VHYPE.burn(2e18 - 1);
    uint256 userVHYPEBalance = VHYPE.balanceOf(user);
    console.log(userVHYPEBalance); // 1 wei
    vm.stopPrank();
}
```

```
uint256 HypeAmount = stakingVaultManager.vHYPEtoHYPE(userVHYPEBalance);
console.log(HypeAmount); // 2e18

vm.startPrank(victim);
stakingVaultManager.deposit{value: 1e18}();
uint256 VictimVHYPEBalance = vHYPE.balanceOf(victim);
console.log(VictimVHYPEBalance); // 0 wei
vm.stopPrank();

HypeAmount = stakingVaultManager.vHYPEtoHYPE(userVHYPEBalance);
console.log(HypeAmount); // 3e18
}
```

### Recommendations:

Restrict the `burn()` and `burnFrom()` function so that it can only be called by the manager.

**Ventuals:** Resolved with [8e77b4b080...](#).

**Zenith:** Verified.

## [H-4] The unstaking amount is calculated incorrectly when slashing occurs

SEVERITY: High

IMPACT: High

STATUS: Resolved

LIKELIHOOD: Medium

### Target

- [StakingVaultManager.sol](#)

### Description:

Tokens can exist in the following locations:

- **StakingVault:** Tokens stored here need to be transferred to the Core.
- **Spot balance in the Core**
- **Staking balance in the Core**, including:
  - Delegated
  - Undelegated
  - Total pending withdrawals

For example, consider the following scenario:

- `delegated = 2000`
- All other values (`undelegated`, `totalPendingWithdrawal`, `Spot balance`, `StakingVault`) are 0.

The first batch, containing 1,000 HYPE, has been finalized.

- `delegated = 1000`
- `totalPendingWithdrawal = 1000`
- `totalHypeProcessed = 1000`

When a validator becomes slashed, the `delegated` amount is reduced to 500. As a result, the slashing also affects the first batch, reducing `totalHypeProcessed` to 600. This leaves 400 HYPE that were previously unstaked during the finalization of the first batch available for subsequent batches.

For example, if the second batch requires a withdrawal of 600 HYPE, only 200 HYPE actually need to be unstaked from the validator. However, the `finalizeBatch` function attempts to unstake the full 600 HYPE from the validator without accounting for the 400 HYPE already available from the first batch. This causes the transaction to revert.

- [StakingVaultManager.sol#L567-L568](#)

```
function finalizeBatch()
    external whenNotPaused whenBatchProcessingNotPaused {
        // Always transfer the full deposit amount to HyperCore spot
        if (depositsInBatch > 0) {
            stakingVault.transferHypeToCore(depositsInBatch);
        }

        // Net out the deposits and withdraws in the batch
        if (depositsInBatch > withdrawsInBatch) {
            uint256 amountToStake = depositsInBatch - withdrawsInBatch;
            stakingVault.stake validator, amountToStake.to8Decimals();
        } else if (depositsInBatch < withdrawsInBatch) {
            @-> uint256 amountToUnstake = withdrawsInBatch - depositsInBatch;
            stakingVault.unstake validator, amountToUnstake.to8Decimals();
        }
    }
}
```

## Recommendations:

```
function finalizeBatch()
    external whenNotPaused whenBatchProcessingNotPaused {
        Batch memory batch = batches[currentBatchIndex];
        ...
        uint256 depositsInBatch = stakingVault.evmBalance();
        uint256 withdrawsInBatch = _vHYPEtoHYPE(batch.vhypeProcessed,
            batch.snapshotExchangeRate);

        // Update totalHypeProcessed to track reserved HYPE for withdrawals
        totalHypeProcessed += withdrawsInBatch;

        L1ReadLibrary.DelegatorSummary memory delegatorSummary = stakingVault.
            delegatorSummary();

        L1ReadLibrary.SpotBalance memory spotBalance = stakingVault.spotBalance(
            stakingVault.HYPE_TOKEN_ID());

        // Always transfer the full deposit amount to HyperCore spot
        if (depositsInBatch > 0) {
            stakingVault.transferHypeToCore(depositsInBatch);
        }
    }
}
```

```
uint256 available = delegatorSummary.totalPendingWithdrawal.to18Decimals()
    +
    spotBalance.total.to18Decimals() + depositsInBatch;
uint256 need = totalHypeProcessed - totalHypeClaimed;

// Net out the deposits and withdraws in the batch
if (depositsInBatch > withdrawsInBatch) {
if (depositsInBatch + spotBalance.total.to18Decimals() > need) {
    // All withdraws are covered by deposits

    // Stake the excess HYPE
    uint256 amountToStake = depositsInBatch - withdrawsInBatch;
    uint256 amountToStake = depositsInBatch + spotBalance.total.
        to18Decimals() - need;
    stakingVault.stake validator, amountToStake.to8Decimals();
} else if (depositsInBatch < withdrawsInBatch) {
} else if (available < need) {
    // Not enough deposits to cover all withdraws; we need to withdraw
    some HYPE from the staking vault

    // Unstake the amount not covered by deposits from the staking vault
    uint256 amountToUnstake = withdrawsInBatch - depositsInBatch;
    uint256 amountToUnstake = need - available;
    stakingVault.unstake validator, amountToUnstake.to8Decimals();
}

emit FinalizeBatch(currentBatchIndex, batches[currentBatchIndex]);

// Increment the batch index
currentBatchIndex++;
}
```

**Ventuals:** Resolved with [d889f6dda0...](#)

**Zenith:** Verified.



## 4.2 Medium Risk

A total of 8 medium risk findings were identified.

[M-1] `finalizeBatch()` will fail when the difference between deposits and withdrawals less than `1e10`

SEVERITY: Medium

IMPACT: Medium

STATUS: Resolved

LIKELIHOOD: Medium

### Target

- [StakingVaultManager.sol#L558-L570](#)

### Description:

When deposits and withdrawals are not equal, `finalizeBatch()` will either stake or unstake.

```
if (depositsInBatch > withdrawalsInBatch) {
    // All withdrawals are covered by deposits

    // Stake the excess HYPE
    uint256 amountToStake = depositsInBatch - withdrawalsInBatch;
    stakingVault.stake(validator, amountToStake.to8Decimals());
} else if (depositsInBatch < withdrawalsInBatch) {
    // Not enough deposits to cover all withdrawals; we need to withdraw some
    // HYPE from the staking vault

    // Unstake the amount not covered by deposits from the staking vault
    uint256 amountToUnstake = withdrawalsInBatch - depositsInBatch;
    stakingVault.unstake(validator, amountToUnstake.to8Decimals());
}
```

The `stake()` or `unstake()` function will fail if `weiAmount == 0`, and `amountToStake` or `amountToUnstake` will be rounded to 8 decimals (divided by `1e10`).

```
function to8Decimals(uint256 amount) internal pure returns (uint64) {
    return SafeCast.toUint64(amount / 1e10);
}
```

So if the difference between deposits and withdrawals is less than 1e10, weiAmount will be 0, causing the stake() or unstake() function to fail.

```
/// @inheritdoc IStakingVault
function stake(address validator, uint64 weiAmount)
    external onlyManager whenNotPaused {
    require(weiAmount > 0, ZeroAmount());
    require(whitelistedValidators[validator],
        ValidatorNotWhitelisted(validator));
    CoreWriterLibrary.stakingDeposit(weiAmount);
    _delegate(validator, weiAmount);
}

/// @inheritdoc IStakingVault
function unstake(address validator, uint64 weiAmount)
    external onlyManager whenNotPaused {
    require(weiAmount > 0, ZeroAmount());
    require(whitelistedValidators[validator],
        ValidatorNotWhitelisted(validator));
    _undelegate(validator, weiAmount);
    CoreWriterLibrary.stakingWithdraw(weiAmount);
}
```

## Recommendations:

It is recommended to return directly instead of reverting when weiAmount == 0 in stake() and unstake() function.

**Ventuals:** Resolved with [5dad7651e5...](#).

**Zenith:** Verified.

## [M-2] The first depositor may receive 0 shares if the total balance becomes 0

SEVERITY: Medium

IMPACT: Medium

STATUS: Resolved

LIKELIHOOD: Low

### Target

- [StakingVaultManager.sol](#)

### Description:

If the total balance is 0, the exchangeRate function returns 0.

- [StakingVaultManager.sol#L719](#)

```
function exchangeRate() public view returns (uint256) {
    uint256 balance = totalBalance();
    uint256 totalSupply = vHYPE.totalSupply();

    // If we have no vHYPE in circulation, the exchange rate is 1
    if (totalSupply == 0) {
        return 1e18;
    }

    // If we have no HYPE in the vault, the exchange rate is 0
    if (balance == 0) {
        return 0;
    }

    return Math.mulDiv(balance, 1e18, totalSupply);
}
```

In this case, the \_HYPETovHYPE function also returns 0.

- [StakingVaultManager.sol#L681](#)

```
function _HYPETovHYPE(uint256 hypeAmount, uint256 _exchangeRate)
    internal pure returns (uint256) {
    if (_exchangeRate == 0) {
```

```
        return 0;
    }
    return Math.mulDiv(hypeAmount, 1e18, _exchangeRate);
}
```

Suppose users deposit HYPE tokens that are then delegated to a validator. In the worst case, the total balance could become 0 if the validator is fully slashed. In this situation, the owner can pause the contract, but before that happens, users can still deposit normally.

Since there's no minimum share amount check in the deposit function, users could deposit their HYPE tokens and receive 0 shares in return.

## Recommendations:

```
function deposit() external payable canDeposit whenNotPaused {
    uint256 amountToDeposit = msg.value.stripUnsafePrecision();

    // Mint vHYPE
    // IMPORTANT: We need to make sure that we mint the vHYPE _before_
    // transferring the HYPE to the staking vault,
    // otherwise the exchange rate will be incorrect. We want the exchange
    // rate to be calculated based on the total
    // HYPE in the vault _before_ the deposit
    uint256 amountToMint = HYPETovHYPE(amountToDeposit);

    +^^I require(amountToMint > 0, ZeroAmount());

    vHYPE.mint(msg.sender, amountToMint);

    // Transfer HYPE to staking vault (HyperEVM → HyperEVM)
    if (amountToDeposit > 0) {
        stakingVault.deposit{value: amountToDeposit}();
    }

    emit Deposit(msg.sender, amountToMint, amountToDeposit);
}
```

**Ventuals:** Resolved with [367396d2cc....](#)

**Zenith:** Verified.

## [M-3] Malicious users can prevent the finalization of batches

SEVERITY: Medium

IMPACT: Medium

STATUS: Resolved

LIKELIHOOD: Low

### Target

- [StakingVaultManager.sol](#)

### Description:

In the `_canFinalizeBatch` function, if there's still a pending `withdrawal` request that could be included in the current `batch`, the finalization is reverted.

- [StakingVaultManager.sol#L605](#)

```
function _canFinalizeBatch(Batch memory batch) internal view {
    uint256 hypeProcessed = _vHYPEtoHYPE(batch.vhypeProcessed,
    batch.snapshotExchangeRate);
    uint256 balance = totalBalance();
    if (balance >= minimumStakeBalance + hypeProcessed) {
        uint256 withdrawCapacityRemaining = balance - minimumStakeBalance
        - hypeProcessed;
        (, uint256 nextWithdrawIdToProcess)
        = withdrawQueue.getNextNode(lastProcessedWithdrawId);
        Withdraw memory withdraw = withdraws[nextWithdrawIdToProcess];
        uint256 expectedHypeAmount = _vHYPEtoHYPE(withdraw.vhypeAmount,
        batch.snapshotExchangeRate);
        @-> require(expectedHypeAmount > withdrawCapacityRemaining,
        HasMoreWithdrawCapacity());
    }
}
```

This behavior allows attackers to prevent batch finalization.

An attacker can continuously repeat the following steps.

1. Queue a withdrawal of 0.5 HYPE (the current minimum withdrawal amount).
2. Cancel the withdrawal.

If the `finalizeBatch` function is called after step 1, the transaction reverts due to the pending withdrawal. As a result, most users will attempt to call `processBatch` instead. However, if this call happens after step 2, nothing is processed. At worst case, the attacker could simply withdraw 0.5 HYPE each time without any actual loss.

This effectively creates a Dos condition on batch finalization.

## Recommendations:

```
function processBatch(uint256 numWithdrawals) external whenNotPaused
    whenBatchProcessingNotPaused {
function processBatch(uint256 numWithdrawals) public whenNotPaused
    whenBatchProcessingNotPaused {

function finalizeBatch()
    external whenNotPaused whenBatchProcessingNotPaused {
    // Check if we have a batch to finalize
    require(currentBatchIndex < batches.length, NothingToFinalize());

    Batch memory batch = batches[currentBatchIndex];

    // Check if we can finalize the batch. This will revert if we cannot
    finalize the batch.
    _canFinalizeBatch(batch);

    processBatch(type(uint256).max);
}

function _canFinalizeBatch(Batch memory batch) internal view {
    // Make sure we have enough balance to cover the withdraws
    if (hypeProcessed > 0) {
        require(balance >= minimumStakeBalance + hypeProcessed,
            NotEnoughBalance());
    }

    // If we've processed all withdraws, we can finalize the batch
    if (lastProcessedWithdrawId == withdrawQueue.getTail()) {
        return;
    }

    // If we haven't processed all withdraws, make sure we've processed all
```

```
        withdraws that we
        // have capacity for
        if (balance ≥ minimumStakeBalance + hypeProcessed) {

            uint256 withdrawCapacityRemaining = balance - minimumStakeBalance -
                hypeProcessed;
            (, uint256 nextWithdrawIdToProcess) = withdrawQueue.getNextNode(
                lastProcessedWithdrawId);
            Withdraw memory withdraw = withdraws[nextWithdrawIdToProcess];
            uint256 expectedHypeAmount = _vHYPEtoHYPE(withdraw.vhypeAmount,
                batch.snapshotExchangeRate);

            require(expectedHypeAmount > withdrawCapacityRemaining, HasMoreWithdra
                wCapacity());
        }
    }
```

**Ventuals:** Resolved with [637644cbdd...](#)

**Zenith:** Verified.

## [M-4] The attackers can delay other users' withdrawals

SEVERITY: Medium

IMPACT: Medium

STATUS: Acknowledged

LIKELIHOOD: Medium

### Target

- [StakingVaultManager.sol](#)

### Description:

The new batch can be finalized one day after the previous batch's finalization.

- [StakingVaultManager.sol#L485](#)

```
function _fetchBatch() internal view returns (Batch memory batch) {
    if (currentBatchIndex == batches.length) {
        // Initialize a new batch at the current index
        // Only enforce timing restriction if this is not the first batch
        if (lastFinalizedBatchTime != 0) {
            // There's a 1 day lockup period after HYPE is staked to a
            // validator, so we enforce a 1 day delay between batches
            require(
                block.timestamp > lastFinalizedBatchTime + 1 days,
                BatchNotReady(lastFinalizedBatchTime + 1 days)
            );
        }
    }
}
```

Also, there can be at most 5 batch finalizations in a week, since unstaking takes 7 days and there can be up to 5 pending withdrawals.

- [StakingVault.sol#L68](#)

```
function unstake(address validator, uint64 weiAmount)
    external onlyManager whenNotPaused {
    L1ReadLibrary.DelegatorSummary memory _delegatorSummary
    = L1ReadLibrary.delegatorSummary(address(this));
    require(_delegatorSummary.nPendingWithdrawals < 5,
        MaxPendingWithdrawals());
}
```



Suppose there are no queued withdrawals and 1 day has passed since the last finalization. In that case, users can withdraw their assets by sequentially calling `queueWithdraw`, `processBatch`, and `finalizeBatch` functions. Under above situation, this allows users to withdraw funds easily.

However, attackers can delay this process by at least one day using a small amount of funds daily. The initial `minimumWithdrawAmount` is 0.5 HYPE, so an attacker who has already deposited some HYPE could repeatedly perform the following steps each day:

1. Check whether one day has passed since the last finalization and that there are at most four pending withdrawals (i.e finalization of the batch is possible).
2. If not, wait until the condition is met.
3. If yes, check for queued withdrawals.
4. If any exist, call `processBatch` and `finalizeBatch`.
5. If there are no queued withdrawals, make a withdrawal of 0.5 HYPE and proceed to finalize a new batch.

This effectively means all withdrawals can be delayed. (normally 1~3 days when 5 pending withdrawals happen in 5 days ) The attacker incurs no loss, and the daily cost of performing this attack is small.

### Recommendations:

Maintain an `allowlist` of users. If a non-allowed user (possible attacker) calls `finalizeBatch`, require the batch's `vHYPE` amount to exceed a minimum batch size which is a value the owner can adjust. If an allowed user calls `finalizeBatch`, there is no batch-size restriction (same as current behavior).

**Ventuals:** Acknowledged that it would be possible to do this. However, because it would only happen in scenarios where there are no queued withdrawals, I don't think it would result in a major impact to delay future queued withdrawals. And I don't believe there's a real incentive to delay withdrawals.

**Zenith:** Acknowledged.

## [M-5] A undervalued exchangeRate may be fetched in queueWithdraw()

SEVERITY: Medium

IMPACT: Medium

STATUS: Resolved

LIKELIHOOD: Medium

### Target

- [StakingVaultManager.sol#L299-L307](#)

### Description:

finalizeBatch() sends the Hype from HypeEVM to HypeCore, at which point evmBalance() decreases, but spotAccountBalance() hasn't increased yet due to timing. That is, the exchange rate will be undervalued at this point.

```
        if (depositsInBatch > 0) {
            stakingVault.transferHypeToCore(depositsInBatch);
        }
    ...
    function transferHypeToCore(uint256 amount)
    external onlyManager whenNotPaused {
        require(amount > 0, ZeroAmount());
        require(block.number > lastEvmToCoreTransferBlockNumber,
            CannotTransferToCoreUntilNextBlock());

        // This is an important safety check - ensures that the StakingVault
        account is activated on HyperCore.
        // If the StakingVault is not activated on HyperCore, and a HyperEVM
        → HyperCore HYPE transfer is made,
        // the transferred HYPE will be lost.
        L1ReadLibrary.CoreUserExists memory coreUserExists
        = L1ReadLibrary.coreUserExists(address(this));
        require(coreUserExists.exists, CoreUserDoesNotExist(address(this)));

        _transfer(payable(HYPE_SYSTEM_ADDRESS), amount);

        lastEvmToCoreTransferBlockNumber = block.number;
    }
```

Although the protocol requires that `deposit()`, which depends on the exchange rate, must be called after one block, i.e., after the `spotAccountBalance()` has been updated, there are some unrestricted functions that also depend on the exchange rate, such as `queueWithdraw()`, `getWithdrawAmount()`, etc.

Especially in `queueWithdraw()`, the undervalued exchange rate will:

1. Make `vHYPEtoHYPE(vhypeAmount)` smaller, thus preventing the user from withdrawing.

```
require(vHYPEtoHYPE(vhypeAmount) >= minimumWithdrawAmount,  
BelowMinimumWithdrawAmount());
```

2. Make `HYPETovHYPE(maximumWithdrawAmount)` larger, thus allowing the user to bypass the `maximumWithdrawAmount` limit.

```
function _splitWithdraws(uint256 vhypeAmount)  
internal view returns (uint256[] memory) {  
    uint256 maximumWithdrawVhypeAmount = HYPETovHYPE(maximumWithdrawAmount);  
  
    // Calculate number of withdraws needed  
    uint256 withdrawCount = (vhypeAmount + maximumWithdrawVhypeAmount - 1)  
    / maximumWithdrawVhypeAmount;  
  
    // Check if the last chunk would be below threshold  
    uint256 lastChunkAmount = vhypeAmount % maximumWithdrawVhypeAmount;  
    if (lastChunkAmount > 0 && lastChunkAmount < minimumWithdrawAmount &&  
        withdrawCount > 1) {  
        withdrawCount--; // Merge last chunk into previous one  
    }  
}
```

## Recommendations:

It is recommended to allow calls to `queueWithdraw()` after `finalizeBatch()` has been called one block later.

**Ventuals:** Resolved with [68c7696bf2....](#)

**Zenith:** Verified.

## [M-6] Switching to a new validator is not possible if the previous validator has been fully slashed

SEVERITY: Medium

IMPACT: Medium

STATUS: Resolved

LIKELIHOOD: Low

### Target

- [StakingVaultManager.sol](#)

### Description:

It is possible for a validator to be fully slashed while HYPE tokens are still deposited. In this case, it should be possible to assign a new validator using the switchValidator function.

- [StakingVaultManager.sol#L786](#)

```
function switchValidator(address newValidator) external onlyOwner {
    L1ReadLibrary.DelegatorSummary memory delegatorSummary
    = stakingVault.delegatorSummary();
    stakingVault.tokenRedelegate(validator, newValidator,
    delegatorSummary.delegated);

    validator = newValidator;
}
```

However, because the old validator was fully slashed, delegatorSummary.delegated is 0, causing the switchValidator transaction to revert.

- [StakingVault.sol#L77](#)

```
function tokenRedelegate(address fromValidator, address toValidator,
    uint64 weiAmount)
    external
    onlyManager
    whenNotPaused
{
    require(weiAmount > 0, ZeroAmount());
```

As a result, assigning a new validator becomes impossible.

This situation prevents recovery of the currently deposited HYPE tokens. Withdrawals can only occur through the batch process, and within the `_fetchBatch` function, the transaction will revert at line 497 because the validator no longer exists.

- [StakingVaultManager.sol#L497](#)

```
function _fetchBatch() internal view returns (Batch memory batch) {
    if (currentBatchIndex == batches.length) {
        if (lastFinalizedBatchTime != 0) {
            require(
                block.timestamp > lastFinalizedBatchTime + 1 days,
                BatchNotReady(lastFinalizedBatchTime + 1 days)
            );

497:         (bool exists, L1ReadLibrary.Delegation memory delegation)
            = stakingVault.delegation(validator);
            require(
                exists && block.timestamp > delegation.lockedUntilTimestamp
                / 1000, /* convert to seconds for comparison */
                BatchNotReady(delegation.lockedUntilTimestamp / 1000 /*
                convert to seconds */ )
            );
        }
    }
}
```

## Recommendations:

```
function tokenRedelegate(address fromValidator, address toValidator,
    uint64 weiAmount)
    external
    onlyManager
    whenNotPaused
{
    require(weiAmount > 0, ZeroAmount());
    require(fromValidator != toValidator, RedelegateToSameValidator());
    require(whitelistedValidators[fromValidator],
        ValidatorNotWhitelisted(fromValidator));
    require(whitelistedValidators[toValidator],
        ValidatorNotWhitelisted(toValidator));

    if (!weiAmount) return;

    _undelegate(fromValidator, weiAmount); // Will revert if the stake is
    locked, or if the validator does not have enough HYPE to undelegate
}
```

```
    _delegate(toValidator, weiAmount);  
}
```

**Ventuals:** Resolved with [d30cc66e77...](#)

**Zenith:** Verified.

## [M-7] Slashing should not be applied to a batch if at least one withdrawal from that batch has already been claimed

SEVERITY: Medium

IMPACT: Medium

STATUS: Resolved

LIKELIHOOD: Low

### Target

- [StakingVaultManager.sol](#)

### Description:

This issue was identified in the previous audit but was not fully remediated. This flaw can cause the `totalHypeClaimed` value to exceed `totalHypeProcessed`, which results in the `totalBalance` function reverting during execution.

### Recommendations:

```
mapping(uint256 ⇒ bool) private claimedBatch;

function claimWithdraw(uint256 withdrawId, address destination)
    public whenNotPaused {
    Withdraw storage withdraw = withdraws[withdrawId];
    require(withdraw.batchIndex ≠ type(uint256).max,
        WithdrawNotProcessed());
    ...
    withdraw.claimedAt = block.timestamp;
    totalHypeClaimed += hypeAmount;

    emit ClaimWithdraw(msg.sender, withdrawId, withdraw);
    claimedBatch[withdraw.batchIndex] = true;
}

function applySlash(uint256 batchIndex, uint256 slashedExchangeRate)
    external onlyOwner {
    require(batchIndex < batches.length, InvalidBatch(batchIndex));
    require(!claimedBatch[batchIndex], "");
}
```

Or

```
function applySlash(uint256 batchSize, uint256 slashedExchangeRate)
  external onlyOwner {
    require(batchIndex < batches.length, InvalidBatch(batchIndex));
    Batch storage batch = batches[batchIndex];

    require(block.timestamp ≤ batch.finalizedAt +
      7 days + claimWindowBuffer, "");
  }
```

**Ventuals:** Resolved with [@849628537b1...](#)

**Zenith:** Verified.



## [M-8] The totalHypeProcessed value may become inaccurate due to slashing

SEVERITY: Medium

IMPACT: Medium

STATUS: Resolved

LIKELIHOOD: Low

### Target

- [StakingVaultManager.sol](#)

### Description:

The slashing can be applied to a batch by the owner before it is finalized.

- [StakingVaultManager.sol#L890](#)

```
function applySlash(uint256 batchSize, uint256 slashedExchangeRate)
    external onlyOwner {
    require(batchIndex < batches.length, InvalidBatch(batchIndex));
    Batch storage batch = batches[batchIndex];

    uint256 oldExchangeRate = batch.slashed ? batch.slashedExchangeRate :
    batch.snapshotExchangeRate;

    // Only adjust totalHypeProcessed if the batch has been finalized
    if (batch.finalizedAt > 0) {
        totalHypeProcessed -= _vHYPEtoHYPE(batch.vhypeProcessed,
        oldExchangeRate);
        totalHypeProcessed += _vHYPEtoHYPE(batch.vhypeProcessed,
        slashedExchangeRate);
    }
}
```

For example, if the snapshotExchangeRate is 100, the slashedExchangeRate is 80, and vhypeProcessed is 100, the issue arises because the batch is not yet finalized, meaning the totalHypeProcessed value is not updated in the applySlash function.

When the batch is later finalized via the finalizeBatch function, the withdrawsInBatch value is calculated using the snapshotExchangeRate (100) instead of the slashedExchangeRate (80).

- [StakingVaultManager.sol#L539](#)

```
function finalizeBatch()
    external whenNotPaused whenBatchProcessingNotPaused {
        // Check if we have a batch to finalize
        require(currentBatchIndex < batches.length, NothingToFinalize());

        Batch memory batch = batches[currentBatchIndex];

        // Check if we can finalize the batch. This will revert if we cannot
        // finalize the batch.
        _canFinalizeBatch(batch);

        uint256 depositsInBatch = stakingVault.evmBalance();
        uint256 withdrawsInBatch = _vHYPEtoHYPE(batch.vhypeProcessed,
            batch.snapshotExchangeRate);

        // Update totalHypeProcessed to track reserved HYPE for withdrawals
        totalHypeProcessed += withdrawsInBatch;
```

As a result, totalHypeProcessed incorrectly increases by 10,000, even though users can only claim HYPE based on the slashed rate (80).

- [StakingVaultManager.sol#L384-L385](#)

```
function claimWithdraw(uint256 withdrawId, address destination)
    public whenNotPaused {
        Withdraw storage withdraw = withdraws[withdrawId];
        Batch memory batch = batches[withdraw.batchIndex];
        require(
            batch.finalizedAt > 0 && block.timestamp > batch.finalizedAt
            + 7 days + claimWindowBuffer,
            WithdrawUnclaimable()
        );

        uint256 withdrawExchangeRate = batch.slashed ? batch.slashedExchangeRate
            : batch.snapshotExchangeRate;
        uint256 hypeAmount = _vHYPEtoHYPE(withdraw.vhypeAmount,
            withdrawExchangeRate);
```

This discrepancy causes totalHypeProcessed to be overstated by 2,000. The correct increase should be based on the slashedExchangeRate. Additionally, other parts of the code use snapshotExchangeRate without checking for a slashed rate. For example,

- [StakingVaultManager.sol#L433](#)

```
function processBatch(uint256 numWithdrawals)
    external whenNotPaused whenBatchProcessingNotPaused {
        Batch memory batch = _fetchBatch();

        uint256 hypeProcessed = _vHYPEtoHYPE(batch.vhypeProcessed,
            batch.snapshotExchangeRate);
```

### Recommendations:

The slashing should only be applied after the batch has been finalized, or the `finalizeBatch`, `processBatch`, and `_canFinalizeBatch` functions should be updated to account for the slashed rate.

**Ventuals:** Resolved with [468486ef14....](#).

**Zenith:** Verified.

## 4.3 Low Risk

A total of 4 low risk findings were identified.

### [L-1] A validator with delegated HYPE should not be removed

SEVERITY: Low

IMPACT: Low

STATUS: Resolved

LIKELIHOOD: Low

#### Target

- [StakingVault.sol](#)

#### Description:

The operator can remove any validator.

- [StakingVault.sol#L159](#)

```
function removeValidator(address validator)
    external onlyOperator whenNotPaused {
        delete whitelistedValidators[validator];
    }
```

However, the validator must not have any delegated HYPE.

#### Recommendations:

```
function removeValidator(address validator)
    external onlyOperator whenNotPaused {
        L1ReadLibrary.Delegation delegation = _getDelegation(validator);
        require(delegation.amount == 0, "");
        delete whitelistedValidators[validator];
    }
```

**Ventuals:** Resolved with [2027b566fd....](#)

**Zenith:** Verified.

## [L-2] There's an incorrect size check when splitting the input vHYPE amounts

SEVERITY: Low

IMPACT: Low

STATUS: Resolved

LIKELIHOOD: Medium

### Target

- [StakingVaultManager.sol](#)

### Description:

The `minimumWithdrawAmount` is the minimum amount of HYPE that can be withdrawn in a single request.

- [StakingVaultManager.sol#L295](#)

```
function queueWithdraw(uint256 vhypeAmount)
    external whenNotPaused returns (uint256[] memory) {
    require(vhypeAmount > 0, ZeroAmount());
    require(vHYPetoHYPE(vhypeAmount) >= minimumWithdrawAmount,
        BelowMinimumWithdrawAmount());
```

In the `_splitWithdraws` function, the input vHYPE amount is divided into several chunks, each not exceeding the maximum allowed vHYPE amount.

- [StakingVaultManager.sol#L341](#)

```
function _splitWithdraws(uint256 vhypeAmount)
    internal view returns (uint256[] memory) {
    uint256 maximumWithdrawVhypeAmount = HYPetoVHYPE(maximumWithdrawAmount);

    // Calculate number of withdraws needed
    uint256 withdrawCount = (vhypeAmount + maximumWithdrawVhypeAmount - 1)
        / maximumWithdrawVhypeAmount;

    // Check if the last chunk would be below threshold
    uint256 lastChunkAmount = vhypeAmount % maximumWithdrawVhypeAmount;
    if (lastChunkAmount > 0 && lastChunkAmount < minimumWithdrawAmount &&
        withdrawCount > 1) {
```

```
withdrawCount--; // Merge last chunk into previous one  
}
```

If the size of the last chunk is smaller than `minimumWithdrawAmount`, it's merged with the previous one. However, this comparison is incorrect because `minimumWithdrawAmount` is measured in HYPE, while `lastChunkAmount` is measured in vHYPE.

## Recommendations:

```
function _splitWithdraws(uint256 vhypeAmount)  
internal view returns (uint256[] memory) {  
    uint256 rate = exchangeRate();  
  
    uint256 maximumWithdrawVhypeAmount = HYPETovHYPE(maximumWithdrawAmount);  
  
    uint256 maximumWithdrawVhypeAmount = _HYPETovHYPE(maximumWithdrawAmount,  
        rate);  
  
    // Calculate number of withdraws needed  
    uint256 withdrawCount = (vhypeAmount + maximumWithdrawVhypeAmount - 1)  
    / maximumWithdrawVhypeAmount;  
  
    // Check if the last chunk would be below threshold  
    uint256 lastChunkAmount = vhypeAmount % maximumWithdrawVhypeAmount;  
  
    uint256 lastChunkHypeAmount = _vHYPEtoHYPE(lastChunkAmount, rate);  
  
    if (lastChunkAmount > 0 && lastChunkAmount < minimumWithdrawAmount &&  
        withdrawCount > 1) {  
  
        if (lastChunkAmount > 0 && lastChunkHypeAmount < minimumWithdrawAmount &&  
            withdrawCount > 1) {  
            withdrawCount--; // Merge last chunk into previous one  
        }  
    }  
}
```

**Ventuals:** Resolved with [@flc826d86c....](#)

**Zenith:** Verified.

### [L-3] The snapshot rate should be used in the setMinimumStakeBalance function

SEVERITY: Low

IMPACT: Low

STATUS: Resolved

LIKELIHOOD: Low

#### Target

- [StakingVaultManager.sol](#)

#### Description:

In the setMinimumStakeBalance function, the current exchange rate is used instead of the snapshot rate.

- [StakingVaultManager.sol#L777](#)

```
function setMinimumStakeBalance(uint256 _minimumStakeBalance)
    external onlyOwner {
    // If we're in the middle of processing a batch, check that we haven't
    // processed more HYPE
    // than what we'd have left after setting the minimum stake balance
    if (currentBatchIndex < batches.length) {
        uint256 newWithdrawCapacity = totalBalance() - _minimumStakeBalance;
        StakingVaultManager.Batch memory batch = batches[currentBatchIndex];
        require(newWithdrawCapacity >= vHYPEtoHYPE(batch.vhypeProcessed),
            MinimumStakeBalanceTooLarge());
    }
    minimumStakeBalance = _minimumStakeBalance;
}
```

However, actual withdrawals are calculated based on the snapshot rate.

#### Recommendations:

```
function setMinimumStakeBalance(uint256 _minimumStakeBalance)
    external onlyOwner {
    // If we're in the middle of processing a batch, check that we haven't
    // processed more HYPE
```



```
// than what we'd have left after setting the minimum stake balance
if (currentBatchIndex < batches.length) {
    uint256 newWithdrawCapacity = totalBalance() - _minimumStakeBalance;
    StakingVaultManager.Batch memory batch = batches[currentBatchIndex];

    uint256 exchangeRate = batch.slashed ? batch.slashedExchangeRate : batch.
        snapshotExchangeRate;

    require(newWithdrawCapacity ≥ vHYPEtoHYPE(batch.vhypeProcessed),
        MinimumStakeBalanceTooLarge());
    require(newWithdrawCapacity ≥ _vHYPEtoHYPE(batch.vhypeProcessed,
        exchangeRate), MinimumStakeBalanceTooLarge());
}
minimumStakeBalance = _minimumStakeBalance;
}
```

**Ventuals:** Resolved with [@a213ba89a2...](#)

**Zenith:** Verified.

## [L-4] StakingVaultManager.deposit() does not refund

SEVERITY: Low

IMPACT: Low

STATUS: Acknowledged

LIKELIHOOD: Low

### Target

- [StakingVaultManager.sol#L280-L296](#)

### Description:

StakingVaultManager.deposit() will normalize msg.value to a multiple of 1e10 and make the deposit, but it does not refund any excess dust.

```
function deposit() external payable canDeposit whenNotPaused {
    uint256 amountToDeposit = msg.value.stripUnsafePrecision();

    // Mint vHYPE
    // IMPORTANT: We need to make sure that we mint the vHYPE _before_
    // transferring the HYPE to the staking vault,
    // otherwise the exchange rate will be incorrect. We want the
    // exchange rate to be calculated based on the total
    // HYPE in the vault _before_ the deposit
    uint256 amountToMint = HYPETovHYPE(amountToDeposit);
    vHYPE.mint(msg.sender, amountToMint);

    // Transfer HYPE to staking vault (HyperEVM → HyperEVM)
    if (amountToDeposit > 0) {
        stakingVault.deposit{value: amountToDeposit}();
    }

    emit Deposit(msg.sender, amountToMint, amountToDeposit);
}

...
function stripUnsafePrecision(uint256 amount)
internal pure returns (uint256) {
    return amount / 1e10 * 1e10;
}
```

**Recommendations:**

It is recommended to refund `msg.value - amountToDeposit`.

**Ventuals:** Acknowledged. We decided not to refund dust in case another contract that composes with this contract is not payable.

## 4.4 Informational

A total of 5 informational findings were identified.

### [I-1] There are unused errors

SEVERITY: Informational	IMPACT: Informational
STATUS: Resolved	LIKELIHOOD: Low

#### Target

- [StakingVaultManager.sol](#)

#### Description:

Below errors are not used.

- [StakingVaultManager.sol#L23](#)

```
error InsufficientBalance();  
error CoreUserDoesNotExist(address account);  
error InvalidWithdrawRequest();
```

#### Recommendations:

```
error InsufficientBalance();  
error CoreUserDoesNotExist(address account);  
error InvalidWithdrawRequest();
```

**Ventuals:** Resolved with [@ccc48624b6....](#)

**Zenith:** Verified.

[I-2] The withdrawable amount should be 0 if the withdrawal has been canceled

SEVERITY: Informational

IMPACT: Informational

STATUS: Resolved

LIKELIHOOD: Low

### Target

- [StakingVaultManager.sol](#)

### Description:

The `getWithdrawAmount` function should return 0 if the `withdrawal` has been canceled.

### Recommendations:

```
function getWithdrawAmount(uint256 withdrawId)
    external view returns (uint256) {
        Withdraw memory withdraw = withdraws[withdrawId];
        uint256 vhypeAmount = withdraw.vhypeAmount;

        if (withdraw.cancelledAt > 0) {
            return 0;
        }

        // If the withdraw hasn't been processed yet, use the current exchange
        rate
        if (withdraw.batchIndex == type(uint256).max) {
            return vHYPEtoHYPE(vhypeAmount);
        }

        // Otherwise, use the exchange rate from the batch
        Batch memory batch = batches[withdraw.batchIndex];
        uint256 _exchangeRate = batch.slashed ? batch.slashedExchangeRate :
        batch.snapshotExchangeRate;
        return _vHYPEtoHYPE(vhypeAmount, _exchangeRate);
    }
```

**Ventuals:** Resolved with [@Oddffcf1c6...](#)

**Zenith:** Verified.

## [I-3] There is an unnecessary check in the finalizeBatch function

SEVERITY: Informational

IMPACT: Informational

STATUS: Resolved

LIKELIHOOD: Low

### Target

- [StakingVaultManager.sol](#)

### Description:

The currentBatchIndex is already checked in the \_canFinalizeBatch function, so the following check is unnecessary.

- [StakingVaultManager.sol#L525](#)

```
function finalizeBatch()
    external whenNotPaused whenBatchProcessingNotPaused {
        // Check if we have a batch to finalize
        @-> require(currentBatchIndex < batches.length, NothingToFinalize());

        Batch memory batch = batches[currentBatchIndex];

        // Check if we can finalize the batch. This will revert if we cannot
        // finalize the batch.
        _canFinalizeBatch(batch);
```

### Recommendations:

```
function finalizeBatch()
    external whenNotPaused whenBatchProcessingNotPaused {
        // Check if we have a batch to finalize
        -^^I require(currentBatchIndex < batches.length, NothingToFinalize());

        Batch memory batch = batches[currentBatchIndex];

        // Check if we can finalize the batch. This will revert if we cannot
        // finalize the batch.
```

```
_canFinalizeBatch(batch);
```

**Ventuals:** Resolved with [@b66480a64f....](#)

**Zenith:** Verified.



## [I-4] There is an unnecessary check in the applySlash function

SEVERITY: Informational

IMPACT: Informational

STATUS: Resolved

LIKELIHOOD: Low

### Target

- [StakingVaultManager.sol](#)

### Description:

Below check is unnecessary.

- [StakingVaultManager.sol#L909](#)

```
function applySlash(uint256 batchSize, uint256 slashedExchangeRate)
    external onlyOwner {
    require(batch.finalizedAt > 0, InvalidBatch(batchIndex));

    uint256 oldExchangeRate = batch.slashed ? batch.slashedExchangeRate :
    batch.snapshotExchangeRate;

    // Only adjust totalHypeProcessed if the batch has been finalized
    @-> if (batch.finalizedAt > 0) {
        totalHypeProcessed -= _vHYPEtoHYPE(batch.vhypeProcessed,
        oldExchangeRate);
        totalHypeProcessed += _vHYPEtoHYPE(batch.vhypeProcessed,
        slashedExchangeRate);
    }
}
```

### Recommendations:

```
function applySlash(uint256 batchSize, uint256 slashedExchangeRate)
    external onlyOwner {
    require(batch.finalizedAt > 0, InvalidBatch(batchIndex));

    uint256 oldExchangeRate = batch.slashed ? batch.slashedExchangeRate :
    batch.snapshotExchangeRate;
```

```
// Only adjust totalHypeProcessed if the batch has been finalized
if (batch.finalizedAt > 0) {
    totalHypeProcessed -= _vHYPEtoHYPE(batch.vhypeProcessed,
oldExchangeRate);
    totalHypeProcessed += _vHYPEtoHYPE(batch.vhypeProcessed,
slashedExchangeRate);
}
}
```

**Ventuals:** Resolved with [@bd1f20489e...](#)

**Zenith:** Verified.

## [I-5] An unnecessary condition check exists in the resetBatch function

SEVERITY: Informational

IMPACT: Informational

STATUS: Resolved

LIKELIHOOD: Low

### Target

- [StakingVaultManager.sol](#)

### Description:

Canceled withdrawals are already removed from the withdrawQueue, making the condition check at line 854 unnecessary.

- [StakingVaultManager.sol#L854](#)

```
function resetBatch(uint256 numWithdrawals) external onlyOwner {
    while (numWithdrawals > 0) {
        if (withdraw.batchIndex == currentBatchIndex) {
            withdraw.batchIndex = type(uint256).max;
            batch.vhypeProcessed -= withdraw.vhypeAmount;

            // Move to the previous withdraw in the queue
            (bool prevNodeExists, uint256 prevNodeId)
= withdrawQueue.getPreviousNode(lastProcessedWithdrawId);
            if (prevNodeExists) {
                lastProcessedWithdrawId = prevNodeId; // Previous withdraw
exists
            } else {
                lastProcessedWithdrawId = 0; // Back at the head of the queue
                break;
            }

            numWithdrawals--;
854:   } else if (withdraw.batchIndex != type(uint256).max) {
            // We've reached a withdrawal that's part of an earlier batch
            // No need to continue since we've reset all withdrawals in the
current batch
            break;
        }
    }
}
```

```
}
```

### Recommendations:

```
function resetBatch(uint256 numWithdrawals) external onlyOwner {  
    while (numWithdrawals > 0) {  
        if (withdraw.batchIndex == currentBatchIndex) {  
            numWithdrawals--;  
        } else if (withdraw.batchIndex != type(uint256).max) {  
        } else {  
            break;  
        }  
    }  
}
```

**Ventuals:** Resolved with [@0495ae32cf...](#)

**Zenith:** Verified.