Read

Kinetiq Security Review

Pashov Audit Group

Conducted by: unforgiven, btk, zark, 0x37

February 26th 2025 - March 6th 2025

Contents

1. About Pashov Audit Group

Pashov Audit Group consists of multiple teams of some of the best smart contract security researchers in the space. Having a combined reported security vulnerabilities count of over 1000, the group strives to create the absolute very best audit journey possible - although 100% security can never be guaranteed, we do guarantee the best efforts of our experienced researchers for your blockchain protocol. Check our previous work here or reach out on Twitter @pashovkrum.

2. Disclaimer

A smart contract security review can never verify the complete absence of vulnerabilities. This is a time, resource and expertise bound effort where we try to find as many vulnerabilities as possible. We can not guarantee 100% security after the review or even if the review will find any problems with your smart contracts. Subsequent security reviews, bug bounty programs and on-chain monitoring are strongly recommended.

3. Introduction

A time-boxed security review of the **kinetiq-research/lst** repository was done by **Pashov Audit Group**, with a focus on the security aspects of the application's smart contracts implementation.

4. About Kinetiq

Kinetiq is a liquid staking protocol built on Hyperliquid that allows users to stake HYPE tokens and receive kHYPE tokens in return, enabling participation in network security while maintaining liquidity. It features advanced validator management, performance-based stake allocation, gas-optimized rebalancing, and an oracle system for tracking rewards and slashing events, all secured by role-based access control and emergency mechanisms.

5. Risk Classification

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

5.1. Impact

- High - leads to a significant material loss of assets in the protocol or significantly harms a group of users.
- Medium - only a small amount of funds can be lost (such as leakage of value) or a core functionality of the protocol is affected.
- Low - can lead to any kind of unexpected behavior with some of the protocol's functionalities that's not so critical.

5.2. Likelihood

- High - attack path is possible with reasonable assumptions that mimic on-chain conditions, and the cost of the attack is relatively low compared to the amount of funds that can be stolen or lost.
- Medium - only a conditionally incentivized attack vector, but still relatively likely.
- Low - has too many or too unlikely assumptions or requires a significant stake by the attacker with little or no incentive.

5.3. Action required for severity levels

- Critical - Must fix as soon as possible (if already deployed)
- High - Must fix (before deployment if not already deployed)
- Medium - Should fix
- Low - Could fix

6. Security Assessment Summary

review commit hash - c83aa178eb429a7e084bdda402aadafe1a58dcc6

fixes review commit hash - 48cfade5085f695771433094cbb8b01962a50be7

Scope

The following smart contracts were in scope of the audit:

- `KHYPE`
- `OracleManager`
- `PauserRegistry`
- `StakingManager`
- `ValidatorManager`
- `L1Read`
- `L1Write`
- `MinimalImplementation`
- `SystemOracle`
- `DefaultAdapter`

7. Executive Summary

Over the course of the security review, unforgiven, btk, zark, 0x37 engaged with Kinetiq to review Kinetiq. In this period of time a total of **22** issues were uncovered.

Protocol Summary

Protocol Name	Kinetiq
Repository	https://github.com/kinetiq-research/lst
Date	February 26th 2025 - March 6th 2025
Protocol Type	Liquid staking

Findings Count

Severity	**Amount**
High	8
Medium	6
Low	8
Total Findings	**22**

Summary of Findings

ID	Title	Severity	Status
[H-01]	_AddRebalanceRequest may use outdated balance for delegate withdrawal request	High	Resolved
[H-02]	ValidatorManager: missing fund withdrawal from validator in deactivatevalidator function	High	Resolved
[H-03]	Deactivated validator retains old balance after reactivation	High	Resolved
[H-04]	ReportSlashingEvent reverts if outdated balance is below slashing amount	High	Resolved
[H-05]	Funds can be permanently locked due to unsafe type casting	High	Resolved
[H-06]	Some stakers may fail to withdraw staking HYPE	High	Resolved
[H-07]	Exchange rate implementation not used in token operations	High	Resolved
[H-08]	Exchange rate calculation is incorrect	High	Resolved
[M-01]	StakingManager getExchangeRatio does not handle first mint	Medium	Resolved
[M-02]	OracleManage::generatePerformance reverts if deactivating validator with ongoing rebalance request	Medium	Resolved
[M-03]	Invalid max stake amount validation prevents setting when no staking limit exists	Medium	Resolved
[M-04]	New stakes delegated even when	Medium	Resolved

	validator is inactive		
[M-05]	Stake function does not account for totalClaimed when checking stakingLimit	Medium	Resolved
[M-06]	Improper execution order in generatePerformance	Medium	Acknowledged
[L-01]	Missing validator active check in rebalanceWithdrawal()	Low	Resolved
[L-02]	OracleManager::lastUpdateTimestamp updated even if no update performed	Low	Resolved
[L-03]	Missing _disableInitializers() call in implementations' constructor	Low	Resolved
[L-04]	OracleManager performance update may revert due to unbounded loops	Low	Resolved
[L-05]	OracleManager does not check for oracle stale values	Low	Resolved
[L-06]	Front-running generatePerformance to gain rewards or avoid slashing	Low	Acknowledged
[L-07]	OracleManager::generatePerformance() will revert if active validator has 0 avgBalance	Low	Resolved
[L-08]	Stakers may fail to withdraw their staking	Low	Acknowledged

8. Findings

8.1. High Findings

[H-01] `_AddRebalanceRequest` may use outdated balance for delegate withdrawal request

Severity

Impact: High

Likelihood: Medium

Description

`ValidatorManager._addRebalanceRequest()` checks validator's balance and records a withdrawal request:

```
function _addRebalanceRequest
      (address validator, uint256 withdrawalAmount) internal {
        require(!_validatorsWithPendingRebalance.contains
          (validator), "Validator has pending rebalance");
        require(withdrawalAmount > 0, "Invalid withdrawal amount");

        (bool exists, uint256 index) = _validatorIndexes.tryGet(validator);
        require(exists, "Validator does not exist");
        require(
          _validators[index].balance>=withdrawalAmount,
          "Insufficientbalance"
        );

        validatorRebalanceRequests[validator] = RebalanceRequest
          ({validator: validator, amount: withdrawalAmount});
        _validatorsWithPendingRebalance.add(validator);

        emit RebalanceRequestAdded(validator, withdrawalAmount);
    }
```

However, a validator's balance can change at any time due to rewards or slashing. This creates two potential problems when manager calls `closeRebalanceRequest()`:

1. Balance Decreases: If the validator's balance decreases after the request is made, the delegation removal or withdrawal will fail.
2. Balance Increases: If the balance increases, some funds may remain in the validator's account.

This is especially critical during emergency withdrawals or when deactivating a validator, as the manager needs to retrieve all funds.

Recommendations

1. Ensure that the `closeRebalanceRequest()` function checks the validator's balance to reflect the most up-to-date amount before processing the withdrawal request.
2. Add a mechanism to `closeRebalanceRequest()` so that manager can withdraw all available funds at the moment in validator balance

[H-02] ValidatorManager: missing fund withdrawal from validator in `deactivatevalidator` function

Severity

Impact: Medium

Likelihood: High

Description

`StakingManager.deactivateValidator()` creates a withdrawal request but does not undelegate/withdraw funds from validator by calling `processValidatorWithdrawals()`:

```
function deactivateValidator
      (address validator) external whenNotPaused nonReentrant validatorExists(validator)
        // Oracle manager will also call this, so limit the msg.sender to both
        // MANAGER_ROLE and ORACLE_ROLE
        require(hasRole(MANAGER_ROLE, msg.sender) || hasRole
          (ORACLE_ROLE, msg.sender), "Not authorized");

        (bool exists, uint256 index) = _validatorIndexes.tryGet(validator);
        require(exists, "Validator does not exist");

        Validator storage validatorData = _validators[index];
        require(validatorData.active, "Validator already inactive");

        // Create withdrawal request before state changes
        if (validatorData.balance > 0) {
            _addRebalanceRequest(validator, validatorData.balance);
        }

        // Update state after withdrawal request
        validatorData.active = false;

        emit ValidatorDeactivated(validator);
    }
```

Because of this issue, funds will stuck in the validator's account:

1. Manager call to `closeRebalanceRequests()` to remove the rebalance request will revert because there is not enough balance in staking manager.
2. Manager call to `rebalanceWithdrawal()` will revert because the validator is added to the pending list.
3. Sentinel call to `requestEmergencyWithdrawal()` will revert too for the same reason

Recommendations

In `deactivateValidator()` function, withdraw funds from validator by calling: `IStakingManager(stakingManager).processValidatorWithdrawals();`

[H-03] Deactivated validator retains old balance after reactivation

Severity

Impact: High

Likelihood: Medium

Description

When a validator is deactivated via `ValidatorManager::deactivateValidator`, the validator's balance is not updated. However, since a rebalance withdrawal request is created for this validator, their **actual** balance becomes 0.

```
function deactivateValidator
      (address validator) external whenNotPaused nonReentrant validatorExists(validator)

        // ...

        Validator storage validatorData = _validators[index];

        require(validatorData.active, "Validator already inactive");

        // Create withdrawal request before state changes

        if (validatorData.balance > 0) {

            _addRebalanceRequest(validator, validatorData.balance);

        }

        // Update state after withdrawal request

        validatorData.active = false;

        emit ValidatorDeactivated(validator);

    }
```

The problem occurs from the fact that even if his actual balance is 0, it will neverben updated since `OracleManager::updatePerformance()` passes the inactive validators without update. This could result to an inactive validator with his old non-zerobalance saved and this can be a problem because when this validator gets reactivate again through `ValidatorManager::reactivateValidator()` he will maintain in his old balance for a while.

The problem occurs from the fact that even though the validator's actual balance is 0, it is never updated because `OracleManager::updatePerformance()` skips inactive validators. As a result, an inactive validator will retain its old non-zero balance in storage. This becomes problematic because when the validator is reactivated via `ValidatorManager::reactivateValidator()`, it will temporarily retain its outdated balance.

```
function reactivateValidator(address validator)

        // ...

    {

        // ...

        Validator storage validatorData = _validators[index];

        require(!validatorData.active, “Validator already active”);

        require(!_validatorsWithPendingRebalance.contains
          (validator), “Validator has pending rebalance”);

        // Reactivate the validator

        validatorData.active = true;

        emit ValidatorReactivated(validator);

    }
```

What is the impact of this?

Firstly, `totalBalance` will be incorrectly inflated until the deactivated validator is reactivated and its balance is updated. This is because the validator receiving the redelegated amount will report an increased balance, and this increase will be included in `totalBalance` during `generatePerformance`.

Consider the following scenario:

- `valA:100, valB:200, totalBalance:300`
- `valA` is deactivated, and its balance is redelegated to `valB`.
- During the next update, `valB` will report a balance of `300`, causing totalBalance to increase to `400`. `valA`, which now has a balance of 0, will not report its balance because it is deactivated.

This is incorrect because no new funds entered the system and only a balance transfer occurred. The system incorrectly tracks more funds than actually exist.

Secondly, in the above scenario, if `valA` is later reactivated, it will still have its old balance until `OracleManager::generatePerformance` updates it. If `valA`'s old balance is larger than the current `totalBalance`, a revert due to underflow will occur when `ValidatorManager::updateValidatorPerformance()` attempts to update `totalBalance`.

For example :

- `valA` has a balance of 100 when deactivated.
- Some time later, `totalBalance` is reduced to 80.
- When `valA` is reactivated, `updateValidatorPerformance()` will attempt to compute :

```
totalBalance = totalBalance - oldBalance + balance;
```

This translates to:

```
totalBalance = 80 - 100 + 0
```

This underflow causes a revert, resulting in a DoS on the update

```
function updateValidatorPerformance(
        address validator,
        uint256 balance,
        // ...
    ) external whenNotPaused onlyRole(ORACLE_ROLE) validatorExists
      (validator) validatorActive(validator) {
        // ...
        // Cache old balance for total balance update
        uint256 oldBalance = val.balance;
        // ...
        // Update total balance
@>        totalBalance = totalBalance - oldBalance + balance;
        // ...
    }
```

Recommendations

Consider zeroing out the validator's balance upon deactivation and subtracting it from `totalBalance` when the rebalance request is created. This will ensures that accounting inconsistencies and the potential underflows upon reactivation, will be avoided.

[H-04] `ReportSlashingEvent` reverts if outdated balance is below slashing amount

Severity

Impact: High

Likelihood: Medium

Description

`OracleManager::generatePerformance` is supposed to be called once every hour (or at any other interval specified by `MIN_UPDATE_INTERVAL`). It updates the stats of every validator and also updates the `totalBalance`. If there are any rewards or slashing amounts, they are supposed to be accounted for and `ValidatorManager` should be updated accordingly.

```
function generatePerformance() external whenNotPaused onlyRole
      (OPERATOR_ROLE) returns (bool) {
        // ..

        // Update validators with averaged values
        for (uint256 i = 0; i < validatorCount; i++) {
            // ...

            // Handle slashing
            if (avgSlashAmount > previousSlashing) {
                uint256 newSlashAmount = avgSlashAmount - previousSlashing;
@>                validatorManager.reportSlashingEvent
  (validator, newSlashAmount);
            }

            // ...
        }

        // ...

        return true;
    }
```

As we can see, if the accumulated slashing amount for this particular validator has increased, `reportSlashingEvent` will be triggered with the increase passed as a parameter.

```
function reportSlashingEvent(address validator, uint256 amount)
        // ...
    {
        require(amount > 0, "Invalid slash amount");

        Validator storage val = _validators[_validatorIndexes.get(validator)];
@>        require(val.balance >= amount, "Insufficient stake for slashing");

        // Update balances
        unchecked {
            // These operations cannot overflow:
            // - val.balance >= amount (checked above)
            // - totalBalance >= val.balance
            //(invariant maintained by the contract)
            val.balance -= amount;
            totalBalance -= amount;
        }

        // ...
    }
```

As shown in the highlighted section, the entire `reportSlashingEvent` (and therefore `generatePerformance` as well) will revert if the new slashing amount is greater than the validator's previously reported balance. However, this is actually a very likely scenario and could easily happen due to the intervals between updates.

For example, at time `T`, the validator's balance could be `100`. At `T + 1 hour`, the validator's balance could grow to `500`, and the `slashingAmount` could be `110`. Due to this bug, the code will require `oldBalance > newSlashingAmount`, which would revert since `slashingAmount` is `110` and `oldBalance` is only `100`. Situations like this are especially likely to occur in the first few days after a validator is activated.

Recommendations

Consider implementing a mechanism for handling slashing amounts similar to how rewards are handled, or compare the new slashing amount with the validator's actual up to-date-balance.

[H-05] Funds can be permanently locked due to unsafe type casting

Severity

Impact: High

Likelihood: Medium

Description

The `StakingManager` contract manages staking operations for HYPE tokens, allowing users to stake their tokens and receive `kHYPE` tokens in return. When users stake tokens through the `stake()` function, the contract delegates these tokens to validators using the `_distributeStake()` internal function, which in turn calls `l1Write.sendTokenDelegate()`.

A critical issue exists in the type casting of the `amount` parameter in both `_distributeStake()` and `_withdrawFromValidator()` functions. The `l1Write.sendTokenDelegate()` function expects a `uint64` parameter, but the contract performs an unsafe cast from `uint256` to `uint64`. If the amount exceeds `type(uint64).max` (18,446,744,073,709,551,615), the value will silently overflow to 0.

```
l1Write.sendTokenDelegate(delegateTo, uint64(amount), false);
```

This becomes particularly dangerous when:

1. `maxStakeAmount` is set to 0 (unlimited) or to a value greater than `type(uint64).max`
2. A user stakes an amount larger than `type(uint64).max`

In such cases, the funds will be accepted by the contract and `kHYPE` tokens will be minted, but the delegation to the validator will be carried out with 0 due to the silent overflow. The tokens will become permanently locked in the contract as there is no mechanism to recover from this situation.

Proof of Concept

1. `maxStakeAmount` is set to 0 (unlimited)
2. User calls `stake()` with 19e18 HYPE tokens (> `type(uint64).max`)
3. Contract accepts the tokens and mints corresponding `kHYPE`
4. In `_distributeStake()`, `amount` is cast to `uint64`, resulting in 0
5. `l1Write.sendTokenDelegate()` is called with 0 tokens
6. The original HYPE tokens remain locked in the contract with no way to recover them

Recommendations

Implement OpenZeppelin's SafeCast library to ensure safe type conversions.

[H-06] Some stakers may fail to withdraw staking HYPE

Severity

Impact: Medium

Likelihood: High

Description

The `StakingManager` contract implements a buffer system intended to maintain a reserve of HYPE tokens for processing withdrawals. However, the current implementation fails to utilize this buffer effectively, potentially forcing unnecessary validator exits and creating systemic risks during periods of high withdrawal demand.

The contract maintains two buffer-related state variables:

- `hypeBuffer`: The current amount of HYPE tokens held in reserve
- `targetBuffer`: The desired buffer size

When users stake HYPE, the contract attempts to maintain this buffer through the `_distributeStake()` function, which prioritizes filling the buffer before delegating remaining tokens to validators. However, the critical flaw lies in the withdrawal logic.

In `queueWithdrawal()`, the contract immediately initiates a validator withdrawal through `_withdrawFromValidator()` without first attempting to service the withdrawal from the buffer. This defeats the purpose of maintaining a buffer and can lead to:

1. Unnecessary validator exits even when sufficient funds are available in the buffer
2. Reduced staking efficiency as funds may be pulled from productive validation

```
_withdrawFromValidator(currentDelegation, amount);
```

This contrasts with more robust implementations like Lido's buffered ETH system, where withdrawals are first serviced from the buffer, and validator exits are only triggered when withdrawal demands exceed available reserves.

Proof of Concept

1. User A stakes 100 HYPE, with `targetBuffer` set to 50 HYPE
    - 50 HYPE goes to buffer
    - 50 HYPE is delegated to validator

2. User B requests withdrawal of 40 HYPE
    - Despite having 50 HYPE in buffer, contract calls `_withdrawFromValidator()`

Recommendations

Modify the `queueWithdrawal()` function to prioritize using the buffer before forcing validator exits.

[H-07] Exchange rate implementation not used in token operations

Severity

Impact: Medium

Likelihood: High

Description

The `StakingManager` contract implements an exchange rate mechanism between HYPE and kHYPE tokens, but fails to utilize it in critical token operations. This discrepancy between the implemented exchange rate logic and its actual usage in token minting and burning operations will lead to incorrect token accounting and value loss for users.

The contract maintains an exchange rate calculation through `getExchangeRatio()` which considers:

- Total staked HYPE
- Total rewards
- Total claimed HYPE
- Total slashing
- Current kHYPE supply

This ratio is meant to reflect the actual value relationship between HYPE and kHYPE tokens. However, the contract's token operations (`stake()`, `queueWithdrawal()`, etc.) assume a 1:1 relationship between HYPE and kHYPE, completely ignoring the implemented exchange rate mechanism.

```
kHYPE.mint(msg.sender, msg.value);
```

The discrepancy between the implemented exchange rate and its actual usage can lead to:

1. Incorrect token minting during staking operations
2. Incorrect token burning during withdrawals
3. Value loss for users when the actual exchange rate deviates from 1:1

Proof of Concept

Consider the following scenario:

1. User stakes 100 HYPE through `stake()`
2. Contract mints 100 kHYPE (1:1 ratio) instead of using `HYPEToKHYPE()`
3. Later, due to slashing or rewards, the exchange rate becomes 0.9 (1 HYPE = 0.9 kHYPE)
4. User requests withdrawal of 100 kHYPE
5. Contract burns 100 kHYPE and sends 100 HYPE (1:1 ratio) instead of using `kHYPEToHYPE()`
6. User receives more HYPE than they should (100 instead of 90)

This creates a direct value loss for the protocol and incorrect token accounting.

Recommendations

The exchange rate should be properly integrated into all token operations. Here's the fix for the `stake()` function:

```
function stake() external payable nonReentrant whenNotPaused {
    // ... existing validation code ...

    uint256 kHYPEAmount = HYPEToKHYPE(msg.value);
    totalStaked += msg.value;
    kHYPE.mint(msg.sender, kHYPEAmount);
}
```

[H-08] Exchange rate calculation is incorrect

Severity

Impact: Medium

Likelihood: High

Description

The protocol implements a staking system where multiple `StakingManager` contracts can coexist, sharing profits and losses through a common `ValidatorManager`. Each `StakingManager` maintains its own accounting of staked and claimed amounts via `totalStaked` and `totalClaimed` state variables, while sharing rewards and slashing through the `ValidatorManager`.

The critical issue lies in the `getExchangeRatio()` function, which calculates the exchange rate between HYPE and kHYPE tokens. The current implementation uses local accounting values but global PnL figures, leading to incorrect exchange rate calculations across different `StakingManager` instances.

The exchange rate calculation follows this formula:

```
exchangeRate =
  (totalStaked + totalRewards - totalClaimed - totalSlashing) / kHYPESupply
```

Where:

- `totalStaked` and `totalClaimed` are local to each `StakingManager`
- `totalRewards` and `totalSlashing` are global values from `ValidatorManager`
- `kHYPESupply` is the total supply of kHYPE tokens

This creates a significant discrepancy as each `StakingManager` will calculate different exchange rates based on its local stake/claim values, while sharing the same rewards and slashing.

Proof of Concept

1. Two `StakingManager` contracts (SM1 and SM2) are deployed
2. User A stakes 100 HYPE through SM1
    - SM1.totalStaked = 100
    - SM2.totalStaked = 0
3. User B stakes 100 HYPE through SM2
    - SM1.totalStaked = 100
    - SM2.totalStaked = 100
4. `ValidatorManager` records 20 HYPE in rewards
5. Exchange rate calculation:
    - SM1: (100 + 20 - 0 - 0) / 100 = 1.2
    - SM2: (100 + 20 - 0 - 0) / 100 = 1.2

    The rates appear equal but are incorrect because each manager only sees half of the total staked amount.

6. Correct rate should be: (200 + 20 - 0 - 0) / 200 = 1.1

    The current design creates arbitrage opportunities and incorrect valuation of kHYPE tokens depending on which `StakingManager` users interact with.

Recommendations

The `ValidatorManager` should track global staking totals, and `StakingManager` instances should query these global values for exchange rate calculations.

8.2. Medium Findings

[M-01] StakingManager `getExchangeRatio` does not handle first mint

Severity

Impact: Medium

Likelihood: Medium

Description

`getExchangeRatio()` is supposed to be used for calculating exchange rate between HYPE and kHYPE. It checks kHYPESupply to be bigger than zero:

```
function getExchangeRatio() public view returns (uint256) {
        uint256 totalSlashing = validatorManager.totalSlashing();
        uint256 totalRewards = validatorManager.totalRewards();
        uint256 kHYPESupply = kHYPE.totalSupply();

@>      require(kHYPESupply > 0, "No kHYPE supply");

        // Calculate total HYPE: totalStaked + totalRewards - totalClaimed -
        // totalSlashing

                uint256 totalHYPE = totalStaked + totalRewards - totalClaimed - totalSl

        // Return ratio = totalHYPE / kHYPE.totalSupply (scaled by 1e18)
        return (totalHYPE * 1e18) / kHYPESupply;
    }
```

As a result, the first user call to `stake` will revert since `kHYPE.totalSupply()` is zero.

Recommendations

Remove the check for `kHYPESupply > 0`.

[M-02] `OracleManage::generatePerformance` reverts if deactivating validator with ongoing

rebalance request

Severity

Impact: High

Likelihood: Low

Description

`OracleManager::generatePerformance` is intended to update the stats of every validator and the `totalBalance` in `ValidatorManager`. This function is called once an hour (or at whatever interval is set by `MIN_UPDATE_INTERVAL`). It is crucial that this call does not revert, as a revert would cause the `ValidatorManager` to remain in a stale state. Additionally, `OracleManager::generatePerformance` performs an abnormal behaviour check, and if necessary, deactivates a validator.

```
// Check for anomalies
        if (
            !_checkValidatorBehavior(

                                validator, previousSlashing, previousRewards, avgSlashA
            )
        ) {
            // Deactivate validator if behavior is unexpected
@>            validatorManager.deactivateValidator(validator);
            continue;
        }
```

However, this deactivation call can easily revert (and thus cause the entire update to fail) if the given validator has an active rebalance request.

```
function deactivateValidator
      (address validator) external whenNotPaused nonReentrant validatorExists(validator)
        // ...

        // Create withdrawal request before state changes
        if (validatorData.balance > 0) {
@>            _addRebalanceRequest(validator, validatorData.balance);
        }

        // ...
    }

    function _addRebalanceRequest
      (address validator, uint256 withdrawalAmount) internal {
@>        require(!_validatorsWithPendingRebalance.contains
  (validator), "Validator has pending rebalance");
        // ...
    }
```

The result of this issue is a temporary DoS that prevents the performance update for validators from proceeding until the rebalance withdrawal request for the affected validator is resolved. In a large-scale system with many validators, such scenarios are likely to occur.

Recommendations

One way to mitigate this issue is to zero out the validator's balance (and subtract it from `totalBalance`) when a rebalance request is created. This way, upon deactivation, the validator would not attempt to open a new rebalance request.

[M-03] Invalid max stake amount validation prevents setting when no staking limit exists

Severity

Impact: Medium

Likelihood: Medium

Description

The `StakingManager` contract implements staking limits through three key parameters:

- `stakingLimit`: Maximum total stake allowed (0 = unlimited)
- `minStakeAmount`: Minimum stake per transaction
- `maxStakeAmount`: Maximum stake per transaction (0 = unlimited)

These parameters can be configured by accounts with the `MANAGER_ROLE` through dedicated setter functions. However, there is a logical error in the `setMaxStakeAmount()` function that prevents setting a valid max stake amount when there is no staking limit configured.

```
if (newMaxStakeAmount > 0) {
            require(
              newMaxStakeAmount>minStakeAmount,
              "Maxstakemustbegreaterthanmin"
            );
            require(
              newMaxStakeAmount<stakingLimit,
              "Maxstakemustbelessthanlimit"
            );
        }
```

The issue occurs in the validation logic of `setMaxStakeAmount()`, where it unconditionally requires that any non-zero `newMaxStakeAmount` must be less than `stakingLimit`. This check fails to account for the case where `stakingLimit` is 0 (unlimited), making it impossible to set a max stake amount when no total limit exists.

Proof of Concept

1. Initially `stakingLimit = 0` (unlimited total staking)
2. Manager attempts to set `maxStakeAmount = 100 ether` to limit individual transactions
3. The call to `setMaxStakeAmount(100 ether)` reverts because:
   - `newMaxStakeAmount > 0` triggers the validation checks
   - `require(newMaxStakeAmount < stakingLimit)` fails as `100 ether < 0` is false

Recommendations

Modify the `setMaxStakeAmount()` function to only perform the staking limit validation when a limit is actually set:

```
function setMaxStakeAmount(uint256 newMaxStakeAmount) external onlyRole
  (MANAGER_ROLE) {
    if (newMaxStakeAmount > 0) {
        require(
          newMaxStakeAmount>minStakeAmount,
          "Maxstakemustbegreaterthanmin"
        );
        if (stakingLimit > 0) {
            require(
              newMaxStakeAmount<stakingLimit,
              "Maxstakemustbelessthanlimit"
            );
        }
    }
    maxStakeAmount = newMaxStakeAmount;
    emit MaxStakeAmountUpdated(newMaxStakeAmount);
}
```

[M-04] New stakes delegated even when validator is inactive

Severity

Impact: Medium

Likelihood: Medium

Description

The `StakingManager` contract manages staking operations and validator delegation in the HYPE staking system. When users stake HYPE tokens through the `stake()` function, the funds are distributed via the internal `_distributeStake()` function, which handles both buffer management and delegation to validators.

Currently, `_distributeStake()` delegates funds to the validator address returned by `validatorManager.getDelegation()` without verifying if that validator is still active. This creates a risk where user funds could be delegated to validators that have been deactivated due to slashing or poor performance.

```
if (amount > 0) {
            address delegateTo = validatorManager.getDelegation(address(this));
            require(delegateTo != address(0), "No delegation set");

            // Send tokens to delegation
            l1Write.sendTokenDelegate(delegateTo, uint64(amount), false);

            emit Delegate(delegateTo, amount);
        }
```

This could lead to:

1. User funds being delegated to inactive/slashed validators
2. Reduced returns for stakers as inactive validators won't generate rewards

Proof of Concept

1. A validator is active and set as the current delegation target
2. The validator gets slashed and deactivated via `OracleManager.generatePerformance()`
3. A user calls `stake()` with 10 HYPE
4. `_distributeStake()` delegates the funds to the deactivated validator
5. The user's stake is now delegated to an inactive validator that won't generate rewards

Recommendations

Add a validation check in `_distributeStake()` to verify the validator's active status before delegation.

[M-05] Stake function does not account for totalClaimed when checking stakingLimit

Severity

Impact: Medium

Likelihood: Medium

Description

The `StakingManager` contract implements a staking limit mechanism through the `stakingLimit` parameter to control the maximum amount of HYPE tokens that can be staked in the system. However, the current implementation fails to accurately track the actual amount of tokens under management due to not accounting for withdrawn tokens.

The `stake()` function performs a limit check using:

```
require(totalStaked + msg.value <= stakingLimit, "Staking limit reached");
```

However, `totalStaked` is only incremented when users stake and never decremented, while a separate `totalClaimed` tracks withdrawals. This means the actual amount of tokens under management is `totalStaked - totalClaimed`, but the limit check uses the raw `totalStaked` value.

This creates a situation where the contract could reject new stakes even when the actual amount under management is well below the limit, effectively reducing the protocol's capacity unnecessarily.

Proof of Concept

- Admin sets stakingLimit to 1000 HYPE
- Alice stakes 1000 HYPE (totalStaked = 1000)
- Alice withdraws 500 HYPE (totalClaimed = 500, totalStaked = 1000)
- Bob tries to stake 100 HYPE
- Transaction reverts due to "Staking limit reached" even though only 500 HYPE is actually staked

Recommendations

Update the staking limit check in `stake()` to account for claimed tokens:

```
require(((
) + msg.value
```

[M-06] Improper execution order in `generatePerformance`

Severity

Impact: Medium

Likelihood: Medium

Description

In OracleManager, the operator role will generate validators' performance result periodically. When we get the performance result, we will check the validator's behavior. If this validator's behavior is below expectation, we will deactivate this validator.

The problem is that we will trigger to deactivate the validator directly, missing updating the latest slash/reward/balance amount.

There are some possible impacts:

1. If there are more rewards than slash in the latest period, we will withdraw less HYPE than we expect.
2. If there are more slash than rewards in the latest period, we will withdraw more HYPE than we delegate in the validator. This will cause the un-delegate failure.
3. In StakingManager, we will make use of `slash` and `reward` amount to calculate the exchange ratio. The calculation will be inaccurate.

```
function generatePerformance() external whenNotPaused onlyRole
      (OPERATOR_ROLE) returns (bool) {
        if (
            !_checkValidatorBehavior(

                                validator, previousSlashing, previousRewards, avgSlashA
            )
        ) {
            validatorManager.deactivateValidator(validator);
            continue;
        }

        if (avgSlashAmount > previousSlashing) {
            uint256 newSlashAmount = avgSlashAmount - previousSlashing;
            validatorManager.reportSlashingEvent(validator, newSlashAmount);
        }

        if (avgRewardAmount > previousRewards) {
            uint256 newRewardAmount = avgRewardAmount - previousRewards;
            validatorManager.reportRewardEvent(validator, newRewardAmount);
        }

        validatorManager.updateValidatorPerformance(

                        validator, avgBalance, avgUptimeScore, avgSpeedScore, avgIntegr
        );
    }
```

Recommendations

Update the balance/reward/slash to the latest amount, then deactivate this validator.

8.3. Low Findings

[L-01] Missing validator active check in `rebalanceWithdrawal()`

In `ValidatorManager` contract: `rebalanceWithdrawal()` does not check if a validator is active before processing withdrawals, while `requestEmergencyWithdrawal()` does.

```
function rebalanceWithdrawal(
        address stakingManager,
        address[] calldata validators,
        uint256[] calldata withdrawalAmounts
    ) external whenNotPaused nonReentrant onlyRole(MANAGER_ROLE) {
        require
          (validators.length == withdrawalAmounts.length, "Length mismatch");
        require(validators.length > 0, "Empty arrays");

        for (uint256 i = 0; i < validators.length;) {
            require(validators[i] != address(0), "Invalid validator address");

            // Add rebalance request (this will check for duplicates)
            _addRebalanceRequest(validators[i], withdrawalAmounts[i]);

            unchecked {
                ++i;
            }
        }

        // Trigger withdrawals through StakingManager
        IStakingManager(stakingManager).processValidatorWithdrawals
          (validators, withdrawalAmounts);
    }
```

```
function requestEmergencyWithdrawal
      (address stakingManager, address validator, uint256 amount)
        external
        onlyRole(SENTINEL_ROLE)
        whenNotPaused
    {
        require(
          block.timestamp>=lastEmergencyTime+EMERGENCY_COOLDOWN,
          "Cooldownperiod"
        );
        require(amount > 0, "Invalid amount");

        if (emergencyWithdrawalLimit > 0) {
            require
              (amount <= emergencyWithdrawalLimit, "Exceeds emergency limit");
        }

        (bool exists, uint256 index) = _validatorIndexes.tryGet(validator);
        require(exists, "Validator does not exist");

        Validator storage validatorData = _validators[index];
@>      require(validatorData.active, "Validator not active");
```

[L-02] `OracleManager::lastUpdateTimestamp` updated even if no update performed

During `OracleManager::generatePerformance()`, the `lastUpdateTimestamp` is supposed to be updated in order for the `MIN_UPDATE_INTERVAL` delay to be enforced between calls. However, if all authorised oracles are inactive, no update on validators will be performed, but the `lastUpdateTimestamp` will still be set to `block.timestamp`. This could possibly block and delay later legitimate and valid calls to `generatePerformance()`, as real updates would have to wait for the `MIN_UPDATE_INTERVAL` to pass.

```
function generatePerformance() external whenNotPaused onlyRole
      (OPERATOR_ROLE) returns (bool) {
        // Check minimum interval between updates
        require(
          block.timestamp>=lastUpdateTime+MIN_UPDATE_INTERVAL,
          "Updatetoofrequent"
        );

        // ...

        // Update lastUpdateTime at the end of successful execution
@>        lastUpdateTime = block.timestamp;
        emit PerformanceUpdated(block.timestamp);

        return true;
    }
```

[L-03] Missing `_disableInitializers()` call in implementations' constructor

The following smart contracts, which are implementations that the proxies will point to, should call `_disableInitializers()` in their `constructor` but they are not doing it : `StakingManager.sol`, `KHYPE.sol`, `PauserRegistry.sol`, `ValidatorManager.sol` and `OracleManager.sol`. This could allow unintended initializations if the contracts are mistakenly deployed without proxies. Adding `_disableInitializers()` would help mitigate this risk by locking the `initializer` functions.

```
KHYPE.sol
    function initialize(
        // ...
    ) public initializer {
        // ...
    }
```

[L-04] OracleManager performance update may revert due to unbounded loops

The `generatePerformance()` function contains 4 nested loops with unbounded iterations over validators and oracles. Since there are no caps on the number of validators that can be added by `ValidatorManager` or oracles that can be authorized by `OracleManager`, this creates a risk of the function hitting the block gas limit and reverting.

The function makes multiple external calls and storage reads/writes within these loops, further increasing gas costs as the number of validators and oracles grows. This could prevent critical performance updates from being processed.

Recommendation: Allow the `OPERATOR_ROLE` to pass a bounded list of validators to process in each call to `generatePerformance()`, rather than attempting to process all validators at once. This would allow performance updates to be processed in batches that fit within gas limits.

[L-05] OracleManager does not check for oracle stale values

`OracleManager.generatePerformance()` gets all data from oracles but it does not check if oracle data is stale based on block number, timestamp or any other method:

```
function generatePerformance() external whenNotPaused onlyRole
  (OPERATOR_ROLE) returns (bool) {
              --snip--

            // Check each validator for this oracle
            for (uint256 i = 0; i < validatorCount; i++) {
                address validator = validators[i];
                if (!validatorManager.validatorActiveState(validator)) continue;

                try IOracleAdapter(oracle).getPerformance(validator) returns (
                    uint256 balance,
                    uint256 uptimeScore,
                    uint256 speedScore,
                    uint256 integrityScore,
                    uint256 selfStakeScore,
                    uint256 rewardAmount,
                    uint256 slashAmount
                ) {
                    // Sanity checks
                    if (

                                                uptimeScore <= maxPerformanceBound && s

                                                        && integrityScore <= maxPerform
                    ) {
                        // Aggregate scores
                        totalBalances[i] += balance;
                        totalUptimeScores[i] += uptimeScore;
                        totalSpeedScores[i] += speedScore;
                        totalIntegrityScores[i] += integrityScore;
                        totalSelfStakeScores[i] += selfStakeScore;
                        totalRewardAmounts[i] += rewardAmount;
                        totalSlashAmounts[i] += slashAmount;
                        validOracleCounts[i]++;
                    }
                } catch {
                    // Skip failed oracle calls
                    continue;
                }
            }
            --snip--
}
```

Record Oracle data timestamp and ignore stale oracle data in `generatePerformance()` function.

[L-06] Front-running `generatePerformance` to gain rewards or avoid slashing

`OracleManager.generatePerformance()` calls `reportRewardEvent()` and `reportSlashingEvent()` functions in `ValidatorManager` to distribute reward/slash among stakers:

```
function generatePerformance() external whenNotPaused onlyRole
  (OPERATOR_ROLE) returns (bool) {
      --snipp--

       // Handle slashing
      if (avgSlashAmount > previousSlashing) {
          uint256 newSlashAmount = avgSlashAmount - previousSlashing;
          validatorManager.reportSlashingEvent(validator, newSlashAmount);
      }

      // Handle rewards
      if (avgRewardAmount > previousRewards) {
          uint256 newRewardAmount = avgRewardAmount - previousRewards;
          validatorManager.reportRewardEvent(validator, newRewardAmount);
      }
      --snipp--
}
```

A user can frontrun `generatePerformance()` and :

1. call `stake()` to unfairly participate in rewards meant for existing stakers.
2. call `confirmWithdrawal()` to escape slashing penalties

To decrease the probability of this attack happening, consider the following:

1. Set a time window for staking and unstaking. Requests should only be processed within a timeframe; otherwise, they should be canceled.
2. Let Others Finalize: Allow other users to finalize a user's stake or unstake request as they have incentives to prevent unfair gains by attackers.

[L-07] `OracleManager::generatePerformance()` will revert if active validator has 0 `avgBalance`

During `OracleManager::generatePerformance()`, it is possible for an active validator to have zero balance for this performance report. However, the current implementation, instead of surpassing it or even deactivating him instantly, will revert the whole transaction due to a `require()` in `_checkValidatorBehavior()`. This revert can cause a temporary DoS of `OracleManager::generatePerformance()`, since if only one of the active validators has 0 `avgBalance`, the whole transaction call will revert and no update will be performed.

```
function _checkValidatorBehavior(
        // ...
    ) internal returns (bool isValid) {
        // Ensure we have a balance to compare against
@>        require(avgBalance > 0, "Zero balance");

        // ...
    }
```

Consider not reverting in the case where `avgBalance = 0`, and instead continue.

```
// Check for anomalies
            if (
+               avgBalance == 0 ||
                !_checkValidatorBehavior(

                                        validator, previousSlashing, previousRewards, a
                )
            ) {
                // Deactivate validator if behavior is unexpected
                validatorManager.deactivateValidator(validator);
                continue;
            }
```

[L-08] Stakers may fail to withdraw their staking

In StakingManager, stakers can request withdraw their staking HYPE via `queueWithdrawal`.

When we queue withdrawal, we will un-delegate the related amount from the current validator. Based on current implementation, one staking manager can support multiple validators, and the owner can rebalance different validators' delegate amount according to actual conditions to earn the maximum profits.

The problem is that when users want to queue withdraw HYPE, they have to un-delegate from the current validator. It's possible that there is not enough delegated HYPE amount in this current validator.

For example:

1. The owner sets the validatorA as current validator via `setDelegation`.
2. Alice stakes 2000 HYPE and these 2000 HYPE will be delegated to the validator A.
3. The owner sets the validatorB to the current validator via `setDelegation`.

4. Alice wants to queue withdraw, we want to un-delegate 2000 HYPE from validatorB. This request will fail.

```
function queueWithdrawal
      (uint256 amount) external nonReentrant whenNotPaused {
        require(amount > 0, "Invalid amount");
        require(kHYPE.balanceOf
          (msg.sender) >= amount, "Insufficient kHYPE balance");

        uint256 withdrawalId = nextWithdrawalId[msg.sender];

        // Lock kHYPE tokens
        kHYPE.transferFrom(msg.sender, address(this), amount);
        _withdrawalRequests[msg.sender][withdrawalId] = WithdrawalRequest
          ({amount: amount, timestamp: block.timestamp});

        nextWithdrawalId[msg.sender]++;
        totalQueuedWithdrawals += amount;
        address currentDelegation = validatorManager.getDelegation(address
          (this));
        require(currentDelegation != address(0), "No delegation set");

        _withdrawFromValidator(currentDelegation, amount);

        emit WithdrawalQueued(msg.sender, withdrawalId, amount);
    }
```

Recommendations: Allow uses to assign the validator that he want to queue withdraw.