Kinetiq LST Security Review

Auditors

Hyh, Lead Security Researcher

Optimum, Lead Security Researcher

Rvierdiiev, Security Researcher

Slowfi, Security Researcher

Report prepared by: Lucas Goiriz

June 24, 2025

Contents

1 About Spearbit

Spearbit is a decentralized network of expert security engineers offering reviews and other security related services to Web3 projects with the goal of creating a stronger ecosystem. Our network has experience on every part of the blockchain technology stack, including but not limited to protocol design, smart contracts and the Solidity compiler. Spearbit brings in untapped security talent by enabling expert freelance auditors seeking flexibility to work on interesting projects together.

Learn more about us at spearbit.com

2 Introduction

Kinetiq empowers Liquid Staking on Hyperliquid.

Disclaimer: This security review does not guarantee against a hack. It is a snapshot in time of Kinetiq LST according to the specific commit. Any modifications to the code will require a new security review.

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

3.1 Impact

- High - leads to a loss of a significant portion (>10%) of assets in the protocol, or significant harm to a majority of users.
- Medium - global losses <10% or losses to only a subset of users, but still unacceptable.
- Low - losses will be annoying but bearable--applies to things like griefing attacks that can be easily repaired or even gas inefficiencies.

3.2 Likelihood

- High - almost certain to happen, easy to perform, or not easy but highly incentivized
- Medium - only conditionally possible or incentivized, but still relatively likely
- Low - requires stars to align, or little-to-no incentive

3.3 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

4 Executive Summary

Over the course of 10 days in total, Kinetiq engaged with Spearbit to review the kinetiq-lst protocol. In this period of time a total of **38** issues were found.

Summary

Project Name	Kinetiq
Repository	kinetiq-lst
Commit	48cfade5
Type of Project	DeFi, Liquid Staking
Audit Timeline	Mar 19th to Mar 29th

Issues Found

Severity	Count	Fixed	Acknowledged
Critical Risk	0	0	0
High Risk	2	2	0
Medium Risk	7	5	2
Low Risk	5	3	2
Gas Optimizations	8	7	1
Informational	16	10	6
Total	**38**	**27**	**11**

5 Findings

5.1 High Risk

5.1.1 Initialization upon declaration will not work for "*proxied*" contracts

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

Description: `OracleManager` and `StakingManager` are designed to operate behind a delegation proxy, meaning their code will be executed, but their storage will not be directly used. However, both contracts declare and initialize storage variables upon declaration, causing these values to be written to the implementation contract's storage instead of the proxy's. As a result, these variables will remain uninitialized in the proxy storage, defaulting to `0`. A critical consequence is that the unstake fee will not be applied.

```
// OracleManager.sol
uint256 public MIN_UPDATE_INTERVAL = 1 hours; // Minimum time between updates
...
uint256 public MAX_ORACLE_STALENESS = 1 hours;
```

```
// StakingManager.sol
uint256 public unstakeFeeRate = 10;
```

Recommendation: Initialize these variables inside the `initialize()` function to ensure they are properly set in the proxy's storage. For the long term, consider using a plugin for safe upgrades as well as use capital case letters only for constant/immutable variables.

Kinetiq: Fixed in e2bac5c.

Cantina Managed: Fixed in e2bac5c by implementing the reviewer's recommendation.

5.1.2 Precision Truncation on Stake May Lead to Improper Accounting and Protocol Insolvency

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

Description: In the `StakingManager` contract, `_distributeStake` sends HYPE to the L1 via:

```
(bool success,) = payable(L1_HYPE_CONTRACT).call{value: truncatedAmount}("");
```

The `truncatedAmount` is derived from user input but may not be divisible by `1e10`, which is required by HyperCore's 8-decimal format. Any residual amount (less than `1e10`) is lost during the transfer. This leads to a permanent accounting mismatch: more `kHYPE` is minted than the amount effectively deposited to L1, creating latent insolvency.

This situation often arises after a withdrawal operation leaves the internal `hypeBuffer` in a non-aligned state. A subsequent deposit will adjust for the buffer, leaving a remainder (`amountToStake`) that is no longer `1e10`-divisible — leading to precision loss and drift.

How it affects the system:

- Silent loss of `HYPE` on every unaligned stake.
- Incorrect `totalStaked` accounting in both `StakingManager` and `StakingAccountant`.
- Minted `kHYPE` cannot be fully redeemed, resulting in user withdrawals reverting.
- If unstake fees are enabled, protocol treasury may absorb the discrepancy since part of the `kHYPE` is not redeemed.

Although the problem may not surface immediately when fees are active, it ultimately dilutes protocol revenue by over-minting `kHYPE` relative to real funds.

The total protocol or user-facing loss in `HYPE` due to misaligned buffer truncation can be expressed as:

- n be the number of deposit operations following a buffer misalignment.
- a_i be the intended amount to stake for deposit i.
- t_i be the truncated amount actually sent to L1 for deposit i.
- $\Delta_i = a_i - t_i$ be the unrecoverable loss from deposit i due to precision truncation.

Then, the total accumulated loss L_{total} is:

$$L_{\text{total}} = \sum_{i=1}^{n} \Delta_i = \sum_{i=1}^{n} \left(a_i \bmod 10^{10} \right)$$

Each Δ_i represents the portion of HYPE (in wei) that is silently dropped due to the 8-decimal constraint on HyperCore, and cannot be recovered via unstake or withdrawal mechanisms. This value accumulates in the contract and cannot be redeemed via withdrawal or unstaking.

Proof Of Concept: The following test demonstrates the exact scenario:

```
function test_simple_bufferLeadsToInsolvency() public {
    vm.prank(manager);
    stakingManager.setUnstakeFeeRate(0);

    _stake(user, 1 ether / 2);
    _unstake(user, 997); // leaves non-aligned buffer
    _stake(user, 0.5 ether); // triggers truncated send
    deal(address(stakingManager), sentToL1 * 1e10 + address(stakingManager).balance);
    _unstake(user, kHYPE.balanceOf(user));
    _passTimeAndWithdraw(user, 0);
    _passTimeAndWithdraw(user, 1); // Reverts - balance not recoverable
}
```

Truncated amounts are **not** retained by the contract or recoverable from `HyperCore`.

Recommendation:

- Change `recordStake()` to operate on the actual `truncatedAmount` that was sent to L1, not the original input amount.
- Consider adding logic to enforce `amountToStake % 1e10 == 0` or normalize the buffer post-withdrawal.
- Track unredeemable "dust" explicitly and separate it from `_cancelledWithdrawalAmount` to allow intentional recovery.
- Warn integrators and users to always interact with a consistent `StakingManager` instance, as buffer state is isolated and loss is local to each instance.

Kinetiq: Fixed in e2bac5c.

Cantina Managed: Fixed in e2bac5c. The fix calculates the amount that was lost previously and stores it in the `buffer` to add liquidity for withdrawals. This effectively solves the issue by expanding the buffer potentially over the maximum `target` defined on the global state variable. This fix works **if and only if** only one `StakingManager` is used per `Khype` token contract or if users withdraw from the same exact `StakingManager` they used for staking. Added event for the fix in `3c1ec31` commit.

5.2 Medium Risk

5.2.1 Sentinel can block core operations for any validator chosen due to missing input control in `requestEmergencyWithdrawal()`

Severity: Medium Risk

Context: ValidatorManager.sol#L326-L342

Description: `generatePerformance(targetValidator)`, `reactivateValidator(targetValidator)`, `requestEmergencyWithdrawal(..., targetValidator,...)` and `rebalanceWithdrawal(..., [..., targetValidator, ...], ...)` operations for any `targetValidator` chosen can be blocked permanently by any sentinel.

For that end a holder of `SENTINEL_ROLE` can call `requestEmergencyWithdrawal(fakeStakingManager, targetValidator, amount)` with `amount > 0` when it's `block.timestamp >= lastEmergencyTime + EMERGENCY_COOLDOWN` with `fakeStakingManager` being a tailor-made contract such that `IStakingManager(fakeStakingManager).processValidatorWithdrawals(validators, amounts)` is a noop, while `IStakingManager(fakeStakingManager).processValidatorRedelegation(...)` reverts. The latter will make `closeRebalanceRequests(fakeStakingManager, ...)` also reverting permanently, making the removal of `targetValidator` from `_validatorsWithPendingRebalance` impossible due to `request.staking == stakingManager` control there:

- ValidatorManager.sol#L270-L305:

```
function closeRebalanceRequests(address stakingManager, address[] calldata validators)
    external
    whenNotPaused
    nonReentrant
    onlyRole(MANAGER_ROLE)
{
    // ...

    uint256 totalAmount = 0;

    for (uint256 i = 0; i < validators.length;) {
        address validator = validators[i];
        require(_validatorsWithPendingRebalance.contains(validator), "No pending request");

        // Add amount to total for redelegation
        RebalanceRequest memory request = validatorRebalanceRequests[validator];
        require(request.staking == stakingManager, "Invalid staking manager for rebalance");  //
        ↪  @audit if `request.staking` is hoax

        totalAmount += request.amount;

        // Clear the rebalance request
        delete validatorRebalanceRequests[validator];
        _validatorsWithPendingRebalance.remove(validator);

        emit RebalanceRequestClosed(validator, request.amount);

        unchecked {
            ++i;
        }
    }

    // Trigger redelegation through StakingManager if there's an amount to delegate
    if (totalAmount > 0) {
        IStakingManager(stakingManager).processValidatorRedelegation(totalAmount);  // @audit
        ↪  then `closeRebalanceRequests()` can be blocked
    }
```

Then, `generatePerformance(targetValidator)` will be failing due to L204 `!validatorManager.hasPendingRebalance(targetValidator)` check as `validatorManager.hasPendingRebalance(targetValidator) == _validatorsWithPendingRebalance.contains(validator) == true`. `reactivateValidator(targetValidator)` will be failing due to `!_validatorsWithPendingRebalance.contains(validator)` L206 check.

`requestEmergencyWithdrawal(..., targetValidator,...)` and `rebalanceWithdrawal(..., [..., targetValidator, ...], ...)` will be failing due to `_addRebalanceRequest(...)` call reverting on the same `!_valida-`

`torsWithPendingRebalance.contains(validator)` check.

Recommendation: Consider adding `stakingManager` whitelist as an input control in `requestEmergencyWithdrawal()`.

Kinetiq: Fixed in PR 11.

Cantina Managed: Fix verified.

5.2.2 Any decrease in slashed or rewarded amounts reported will make `validatorManager` accounting permanently corrupted

Severity: Medium Risk

Context: OracleManager.sol#L304-L312

Description: If there be a reverse slashing or rewarding, i.e. oracle reported slashed or rewarded amounts decrease for any reason, then the corresponding `ValidatorManager` accounting corrupts permanently since it's ignored there (`reportRewardEvent()` and `reportSlashingEvent()` aren't invoked), but not by oracle.

This way when it is `avgSlashAmount < previousSlashing` or `avgRewardAmount < previousRewards` there will be no update in validatorManager accounting, which will become off: validatorManager's `totalRewards`, `validatorRewards` and `totalSlashing`, `validatorSlashing` will permanently diverge from Oracle readings:

- ValidatorManager.sol#L469-L503:

```
/// @notice Report a reward event for a validator
/// @param validator Address of the validator to be rewarded
/// @param amount Amount of rewards for the validator
function reportRewardEvent(address validator, uint256 amount)
    external
    onlyRole(ORACLE_ROLE)
    validatorActive(validator)
{
    require(amount > 0, "Invalid reward amount");

    // Update reward amounts
    totalRewards += amount;  // @audit has to follow Oracle updates
    validatorRewards[validator] += amount;

    emit RewardEventReported(validator, amount);
}

/* ========== SLASHING ========== */

/// @notice Report a slashing event for a validator
/// @param validator Address of the validator to be slashed
/// @param amount Amount to slash from the validator
function reportSlashingEvent(address validator, uint256 amount)
    external
    onlyRole(ORACLE_ROLE)
    validatorActive(validator)
{
    require(amount > 0, "Invalid slash amount");

    // Update slashing amounts
    totalSlashing += amount;  // @audit has to follow Oracle updates
    validatorSlashing[validator] += amount;

    emit SlashingEventReported(validator, amount);
}
```

In a simple scenario of going back and forth between some old value, e.g. `900`, and some new one, e.g. `1000`, only upticks to be recorded in validatorManager's accumulators, which can become unlimitedly bloated this way.

Recommendation: Since it's implicitly assumed that there will be no decrease in slashed or rewarded amounts reported by Oracles consider directly controlling for that.

Kinetiq: Fixed in commit b8bfbaee.

Cantina Managed: Fix verified.

5.2.3 `_checkValidatorBehavior()` is mistakenly using the current validator balance

Severity: Medium Risk

Context: OracleManager.sol#L165

Description: `_checkValidatorBehavior()` is a function that implements checks to make sure the changes in slashed amount and rewards amount are reasonable. To achieve it, tolerance levels are defined for both variables. In practice, these tolerance levels are represented as percentages of the `avgBalance` parameter, which is the **current** balance of the validator (after the change). To better understand the issue let's consider a simple example:

- SlashingTolerence = 10%.
- Current balance of validator A is 100 (already stored inside the `ValidatorManager` contract).
- At this point, there is a slashing of 10.
- For simplicity, assuming all oracles submit the same values, in the call to `generatePerformance()` we will have:

```
avgBalance = 90.
avgSlashAmount = 10.
```

- The issue here is that the call to `_checkValidatorBehavior()` will return false (instead of true) since the computation will be:

```
slashingBps ((10/90)%) > SlashingTolerence (10%).
```

Recommendation: Consider fetching the balance before the change by calling `ValidatorManager.validatorPerformance()` and using it for `_checkValidatorBehavior()` instead.

Kinetiq: Acknowledged.

Cantina Managed: Acknowledged.

5.2.4 Non whitelisted caller can queue withdrawal

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

Description: There will be several `StakingManager` contracts in the system as some of them will be deployed for institutions. Only whitelisted addresses should be allowed to stake and withdraw from the institution's `StakingManager`. It's possible to configure whitelisted addresses through `_whitelist` variable. The problem is that it's only checked in the `stake()` function that caller is whitelisted, but there is no same check in the `queueWithdrawal()`. Thus a non-allowed actor can interact with `StakingManager` of the institution.

Recommendation: It's recommended to check that caller is whitelisted in the `queueWithdrawal()` function.

Kinetiq: Fixed in e306672.

Cantina Managed: Fixed in commit e306672 by implementing the reviewer's recommendation.

5.2.5 Lack of liquidity inside `StakingManager`

Severity: Medium Risk

Context: StakingAccountant.sol#L142-L147

Description: The exchange rate of `kHYPE` is calculated based on total amounts of all `StakingManagers` in the system.

```
uint256 rewardsAmount = validatorManager.totalRewards();
uint256 slashingAmount = validatorManager.totalSlashing();
uint256 totalHYPE = totalStaked + rewardsAmount - totalClaimed - slashingAmount; // <<<

// Calculate ratio with 18 decimals precision
return Math.mulDiv(totalHYPE, 1e18, kHYPESupply);
```

Those `StakingManager` contracts may delegate funds to different validators. As there is no way to move liquidity between the `StakingManagers`, there could be lack of liquidity in case users want to exit `StakingManager` as they don't have enough funds on their validator's balance.

To explain the problem an example is given:

- Suppose that `StakingManager1` staked 100 Hype to validator1 and earned 5 Hype in rewards, but `StakingManager2` staked 100 Hype to validator2 and earned 15 Hype as rewards, thus according to exchange ratio formula, both StakingManagers should have 110 Hype to withdraw, but `StakingManager1` won't be able to withdraw 110 from the validator as it has only 105 on balance of validator1. In case the stakers of `StakingManager1` want to withdraw all funds there would be a shortage of 5 Hype.
- The protocol team said that this liquidity should be withdrawn from `StakingManager2` and staked to `StakingManager1` by any user or protocol admin. While this is indeed possible, in case if `StakinManager2` is created for institution, it won't be possible to simply interact with it, as it has a whitelist enabled that limits actors who can participate.

Recommendation: Consider ensuring and documenting that institutional usage of the protocol takes place via separate instance of it, so all the `StakingManagers`' `MANAGER_ROLE` and `ValidatorManager`'s `MANAGER_ROLE` belongs to the same party, the institution, who was instructed to maintain enough stakes in each `StakingManager`, monitor the liquidity and perform the necessary rebalancing between `StakingManagers`.

Kinetiq: Acknowledged.

Cantina Managed: Acknowledged.

5.2.6 Withdrawal Cooldown Can Be Bypassed And Overly Restrictive

Severity: Medium Risk

Context: StakingManager.sol#L278

Description: In the `StakingManager` contract, the `queueWithdrawal` function enforces a cooldown using `lastStakeTimestamp[msg.sender]`. While this aims to prevent immediate withdrawals after staking, there are a couple of edge cases worth noting:

- The restriction can be bypassed by transferring `kHYPE` to another EOA controlled by the user. That account would not be subject to the cooldown and could proceed with withdrawal immediately.
- The cooldown is applied to the entire `kHYPE` balance, not just the portion staked most recently. For example, if a user stakes 100 HYPE, waits for the cooldown period, and then stakes an additional 10 HYPE, they would be blocked from withdrawing the full 110 HYPE until the cooldown resets — even though the original 100 HYPE is technically eligible.

These behaviors may unintentionally limit user flexibility or encourage workarounds via multiple addresses.

Recommendation: Consider refining the cooldown logic to track cooldowns per deposit or per `kHYPE` tranche, or revisiting whether the restriction provides meaningful protection in its current form. Also, be aware that users can bypass the cooldown entirely by redistributing their `kHYPE` to a fresh EOA.

Kinetiq: Fixed in commit e2bac5c.

Cantina Managed: Fixed in commit e2bac5c by removing the cooldown withdrawal mechanism from the protocol.

5.2.7 `ValidatorSanityChecker` storage variables can be set by anyone, causing the rejection of oracles reports

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

Description: This issue was found during the fixes review period in commit hash e2bac5c. As part of mitigating the issues found during the review, the Kinetiq team introduced a new contract named `ValidatorSanityChecker` which is used for validity checks on oracles reports. The contract has one main function named `checkValidatorSanity` which among other checks, validates that given values (such as `avgUptimeScore`, `avgSpeedScore`, etc) are inside bounds which are based on storage variables that can be set. The issue here is that these storage variables can be set by anyone which can be used by attackers to cause the rejection of oracles reports. The full list of unprotected functions:

1. `setSlashingTolerance()`.
2. `setRewardsTolerance()`.
3. `setScoreTolerance()`.
4. `setMaxScoreBound()`.

Recommendation: Consider restricting the access to these functions to privileged accounts only.

Kinetiq: Fixed in commit b739069.

Cantina Managed: Fix verified.

5.3 Low Risk

5.3.1 Admin will pay fee while withdrawing collected fees

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

Description: When kHYPE is unstaked through `queueWithdrawal()` function, then fee is collected by the protocol. It is sent in form of kHYPE token to the treasury. As there is no other function to redeem fees, in order to do that protocol will use `queueWithdrawal()` function as well. The problem is that this function will charge fee on collected fees, thus not all amount will be redeemed.

Recommendation: It's recommended to create additional function that allows redeeming fees without applying fee on them.

Kinetiq: Fixed in commit e2bac5c.

Cantina Managed: Fixed in commit e2bac5c by setting fee to 0 when caller is a treasury.

5.3.2 `StakingManager.confirmWithdrawal()/batchConfirmWithdrawals()` are using `.transfer()` limiting the call gas to 2300

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

Description: Using `.transfer()` to transfer native assets is compiled to a low level `CALL` opcode with a gas limitation of 2300 during compilation. In case the recipient is a smart contract with a fallback function that consumes more than 2300, the call will fail and the entire call to `confirmWithdrawal()/batchConfirmWithdrawals()` will revert, denying the service for these kind of recipients.

```
function confirmWithdrawal(uint256 withdrawalId) external nonReentrant whenNotPaused {
    uint256 amount = _processConfirmation(msg.sender, withdrawalId);
    require(amount > 0, "No valid withdrawal request");
    require(address(this).balance >= amount, "Insufficient contract balance");

    // Process withdrawal
    payable(msg.sender).transfer(amount);

    stakingAccountant.recordClaim(amount);
}
```

```
function batchConfirmWithdrawals(uint256[] calldata withdrawalIds) external nonReentrant whenNotPaused {
    uint256 totalAmount = 0;

    for (uint256 i = 0; i < withdrawalIds.length; i++) {
        totalAmount += _processConfirmation(msg.sender, withdrawalIds[i]);
    }

    // Process total withdrawal if any valid requests were found
    if (totalAmount > 0) {
        require(address(this).balance >= totalAmount, "Insufficient contract balance");
        payable(msg.sender).transfer(totalAmount);
    }

    stakingAccountant.recordClaim(totalAmount);
}
```

Recommendation: Consider replacing `.transfer()` with `.call()` instead, keep in mind that it will require placing the `.call()` after the call to `stakingAccountant.recordClaim(amount);` (a writing operation).

Kinetiq: Fixed in commit e2bac5c.

Cantina Managed: Fixed in commit e2bac5c by implementing the reviewer's recommendation.

5.3.3 `StakingManager.setWithdrawalDelay()`: `withdrawalDelay` should not be less than 7 days

Severity: Low Risk

Context: StakingManager.sol#L720

Description: `withdrawalDelay` is a storage variable used to set the minimum time from which withdrawals can be made by users. The withdrawal process for users includes the following steps:

1. `StakingManager.queueWithdrawal()` (called by the staker).
2. `StakingManager.processL1Operations()` (called by the operator).
3. `StakingManager.withdrawFromSpot()` (called by the operator, after 7 days - the withdrawal time of HyperLiquid).
4. `confirmWithdrawal()/batchConfirmWithdrawals()` (called by the staker).

`_processConfirmation()` is called as part of `confirmWithdrawal()/batchConfirmWithdrawals()` and is setting the minimum time which from the staker can finalize the withdrawal:

```
function _processConfirmation(address user, uint256 withdrawalId) internal returns (uint256) {
    WithdrawalRequest memory request = _withdrawalRequests[user][withdrawalId];

    // Skip if request doesn't exist or delay period not met
    if (request.hypeAmount == 0 || block.timestamp < request.timestamp + withdrawalDelay) {
        return 0;
    }
    // ...
```

In the current version of the code, `withdrawalDelay` can be set to any value as we can see in:

```
function setWithdrawalDelay(uint256 newDelay) external onlyRole(MANAGER_ROLE) {
    withdrawalDelay = newDelay;
    emit WithdrawalDelayUpdated(newDelay);
}
```

The issue here is that in case `withdrawalDelay` is less than 7 days, it allows stakers to launch the last step of the withdrawal before his funds are waiting in the contract. In this case stakers might be able to withdraw funds that belong to other stakers that asked for a withdrawal but have not claimed it yet, causing these users to wait additional 7 days in the worst case.

Recommendation: Consider reverting calls to `setWithdrawalDelay()` in case `newDelay` is less than 7 days.

Kinetiq: Acknowledged.

Cantina Managed: Acknowledged.

5.3.4 Staking limit calculation is not accurate

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

Description: Currently staking limit calculation depends on:

- total staked amount of StakingManager.
- total claimed amount of StakingManager.
- global rewards amount that was earned by all StakingManagers in the system.

```
uint256 rewardsAmount = validatorManager.totalRewards();
uint256 netStaked = totalStaked + rewardsAmount - totalClaimed;
require(netStaked + msg.value <= stakingLimit, "Staking limit reached");
```

There are several points to note here:

1. Slashed amount is not accounted (only slashed amount for current `StakingManager` should be used).
2. Rewards amount is a global value and not reflect amount that was earned by current `StakingManager`.

Because of that, the calculation is not accurate, thus the protocol can't limit staking precisely.

Also it's possible that when new `StakingManager` is added to the system then staking will be blocked, if total rewards amount is bigger than staking limit. E.g. for a very old system and fresh `StakingManager` its capacity can be filled by old rewards earned and claimed before the addition of this `StakingManager` as the currently used `totalStaked + rewardsAmount - totalClaimed = 0 + bigNumber - 0` figure isn't represent `netStaked`, but merely total rewards earned overall. So, if `stakingLimit` be naively set to some amount for net staking only for this manager disregarding overall rewards then no new staking be effectively allowed.

Recommendation: In order to do accurate calculations protocol needs to track rewards and slashing amounts per `StakingManager`, which is not easy to do with current implementation.

Kinetiq: Acknowledged.

Cantina Managed: Acknowledged.

5.3.5 Lack of Validator Address Validation in `queueL1Operations` function

Severity: Low Risk

Context: StakingManager.sol#L548-L575

Description: In the `StakingManager` contract, the `queueL1Operations` function allows a trusted role (`OPERATOR_ROLE`) to enqueue a list of validator-related operations. However, it does not validate that the provided `validators[i]` addresses are legitimate or currently active validators. This contrasts with other functions in the contract that explicitly check validator status using `ValidatorManager`, e.g., via `validatorActiveState()` or `getDelegation()`.

While `queueL1Operations` is restricted to operator-only access, the absence of a validation step leaves room for accidental queuing of invalid or inactive validator addresses. This may result in wasted queue entries, failed execution attempts, or the need for manual correction/reset of the queue.

Recommendation: Consider validating each validator address against `ValidatorManager` during `queueL1Operations`, such as by checking:

```
require(validatorManager.validatorActiveState(validators[i]), "Invalid validator");
```

This would bring consistency with other validator-related operations and help reduce the potential for operational mistakes, even under trusted roles.

Kinetiq: Fixed in commit e2bac5c.

Cantina Managed: Fixed in commit e2bac5c by implementing the validator check. Moreover the Kinetiq team add a flow control to perform the check just for deposit operations. Thus making the function robust consistent and secure.

5.4 Gas Optimization

5.4.1 `redelegateWithdrawnHYPE`: `_cancelledWithdrawalAmount` can be cached earlier in the function

Severity: Gas Optimization

Context: StakingManager.sol#L845

Description:

```
function redelegateWithdrawnHYPE() external onlyRole(MANAGER_ROLE) whenNotPaused {
    require(_cancelledWithdrawalAmount > 0, "No cancelled withdrawals");
    require(address(this).balance >= _cancelledWithdrawalAmount, "Insufficient HYPE balance");

    uint256 amount = _cancelledWithdrawalAmount;
    _cancelledWithdrawalAmount = 0;

    // Delegate to current validator
    _distributeStake(amount, OperationType.RebalanceDeposit);

    emit WithdrawalRedelegated(amount);
}
```

As we can see, `_cancelledWithdrawalAmount` is stored locally in `amount`, but this should be the first line of the function.

Kinetiq: Acknowledged.

Cantina Managed: Acknowledged.

5.4.2 `ValidatorManager.requestEmergencyWithdrawal()`: validator existence check is executed twice

Severity: Gas Optimization

Context: ValidatorManager.sol#L338

Description:

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

// Create rebalance request
_addRebalanceRequest(stakingManager, validator, amount);
```

```
function _addRebalanceRequest(address staking, 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");
    // ...
```

As we can see, during the execution of `requestEmergencyWithdrawal` the existence check happens twice.

Recommendation: Consider removing the first check while keeping the existence check that's inside `_addRebalanceRequest()`.

Kinetiq: Fixed in commit ff58fa25.

Cantina Managed: Fix verified.

5.4.3 Remove redundant modifiers from `StakingManager.processL1Operations()` function

Severity: Gas Optimization

Context: StakingManager.sol#L669

Description: Modifiers `onlyRole(OPERATOR_ROLE)` and `whenNotPaused` from `StakingManager.processL1Operations()` function are redundant as they are executed in overloaded version of function, in the next call in the stack.

Recommendation: It's recommended to remove those modifiers to save gas.

Kinetiq: Fixed in commit 821c985f.

Cantina Managed: Fixed in commit 821c985f by applying the recommendation.

5.4.4 Remove redundant operations inside `StakingManager.initialize()` function

Severity: Gas Optimization

Context: StakingManager.sol#L199-L201

Description: As `DEFAULT_ADMIN_ROLE` is set as admin role for each new role by default, you can remove operations to manually set the role admin.

```
_setRoleAdmin(OPERATOR_ROLE, DEFAULT_ADMIN_ROLE);
_setRoleAdmin(MANAGER_ROLE, DEFAULT_ADMIN_ROLE);
_setRoleAdmin(TREASURY_ROLE, DEFAULT_ADMIN_ROLE);
```

Recommendation: It's recommended to remove those operations to save gas.

Kinetiq: Fixed in commit e2bac5c5.

Cantina Managed: Fixed in commit e2bac5c5 by applying the recommendation.

5.4.5 Redundant Storage Assignment in `authorizeOracleAdapter`

Severity: Gas Optimization

Context: OracleManager.sol#L98

Description: In the `OracleManager` contract, the `authorizeOracleAdapter` function sets `activeOracles[adapter] = false;` when revoking an adapter. Since `false` is the default value for a `bool` mapping in Solidity, this storage write is redundant and results in unnecessary gas usage.

Recommendation: Consider removing the assignment to avoid redundant storage operations and reduce gas costs.

Kinetiq: Fixed in commit e2bac5c.

Cantina Managed: Fixed in commit e2bac5c by removing the redundant assingment.

5.4.6 Unnecessary Use of `memory` for Immutable String Parameters

Severity: Gas Optimization

Context: KHYPE.sol#L51-L52

Description: In the `KHYPE` contract, the `initialize` function accepts `name` and `symbol` parameters as `string memory`. Since the function is `external` and these parameters are not modified within the function, they can be declared as `calldata` instead of `memory`. Using `calldata` avoids an unnecessary copy operation from calldata to memory, resulting in lower gas usage.

Recommendation: Consider changing the parameter declarations from `string memory` to `string calldata` in the `initialize` function to reduce gas costs:

```
function initialize(
    string calldata name,
    string calldata symbol,
    // ...
)
```

This is a minor but effective optimization for external functions that do not modify their string parameters.

Kinetiq: Fixed in commit e2bac5c.

Cantina Managed: Fixed in commit e2bac5c by implementing the reviewer's recommendation.

5.4.7 Redundant `validator != address(0)` Check in `_distributeStake` function

Severity: Gas Optimization

Context: StakingManager.sol#L450

Description: In the `StakingManager` contract, the `_distributeStake` function performs an explicit `require(validator != address(0))` check after calling `validatorManager.getDelegation(address(this))`. However, the `getDelegation` function in `ValidatorManager` already includes this check internally, along with an additional check for validator activity. Rechecking `validator != address(0)` in `_distributeStake` is therefore redundant and introduces unnecessary bytecode and gas consumption.

Recommendation: Consider removing validation line from `_distributeStake` in `StakingManager`. This check is already enforced by the `ValidatorManager.getDelegation` function and can be safely omitted to reduce gas usage.

Kinetiq: Fixed in commit e2bac5c.

Cantina Managed: Fixed in commit e2bac5c by implementing the reviewer's recommendation.

5.4.8 Single Operation Queue Limit Privileged Granular Control And Efficiency

Severity: Gas Optimization

Context: StakingManager.sol#L664

Description: In the `StakingManager` contract, `queueL1Operations` allows an operator to queue various types of operations — including user withdrawals, rebalances, and deposits — using a single unified array. This array is later processed in `_distributeStake`, which iterates through all entries and performs actions based on their `OperationType`. This structure introduces potential inefficiencies and reduces flexibility:

- All operation types are stored in the same queue, so even when only a specific type (e.g., operator-driven rebalances) needs to be executed, the system must iterate through unrelated entries.
- There is no built-in mechanism to prioritize or isolate operator-managed actions from user-initiated ones.
- Since any user can call `stake()` or `queueWithdrawal()`, they may continuously append to the shared queue. This can interfere with time-sensitive or critical operator-driven actions, especially if the protocol depends on central roles to stabilize or rebalance validator allocations.
- The presence of `queueL1Operations` (restricted to `OPERATOR_ROLE`) suggests that operator-level management is an expected control flow, but the shared queue design makes it difficult to guarantee clean separation from public activity.

These constraints could cause delays or require repeated queue resets, undermining the goal of centralized administrative control in contingency or exception scenarios.

Recommendation: Consider splitting the operation queue into multiple dedicated queues — e.g., one for user-triggered deposits, another for user withdrawals, and a separate queue for operator-initiated rebalances or administrative flows. This would allow:

- More efficient execution by minimizing unnecessary branching.
- Clearer operational separation between user activity and protocol control actions.
- Greater reliability for admin-managed situations, where unexpected user actions would otherwise interfere with intended validator reconfiguration or fund movement.

Kinetiq: Fixed in commit 833f07c6.

Cantina Managed: Fixed in commit 833f07c6 by implementing the reviewer's recommendation.

5.5 Informational

5.5.1 Notes about `knowledge-bank/staking_flow`

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: During the review, the team has provided us a file named staking_flow which is part of the documentation. This file is inaccurate and inconsistent with the current implementation. Below a list of things that should be fixed:

1. The diagram should include an explanation of which steps are direct EVM calls and which are cross-chain communication. For example, "Step 1: Send to L1" is a direct call while "Step 2: Move to staking account" relies on cross-chain communication.
2. The functionality that is described to be part of the kHYPE contract mostly resides in `StakingManager` instead.
3. The "Unstaking Workflow Implementation" is not updated:
    - `StakingManager` contract does not have an `unstake` function but rather has `queueWithdrawal()` instead.
    - `initiateTransferToSpot`, `initiateWithdrawal` does not appear in the repo at all.

Kinetiq: Fixed in commit b739069.

Cantina Managed: Fixed verified.

5.5.2 Trust assumptions on oracles in the system

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description:

1. `generatePerformance()` implements a few input checks, some are per oracle and some are per the average values of all reporting oracles. In case the checks of an oracle fail, it is skipped and not being accounted as part of the calculated average values. However, in case the checks of `_checkValidatorBehavior()` are invalid (`valid = false`), then the entire call to `generatePerformance()` will return false which will keep the system outdated. Malicious oracles can use it to report values (such as slashing or rewards amounts above the accepted threshold) to cause the entire call to `generatePerformance()` to fail, potentially causing a denial of service.
2. `generatePerformance()` loops through all authorized oracles to get average values. In case the oracle data is stale or call to the oracle reverts, then such oracle is just skipped. Because of that, only `1` oracle out of X may provide data and such report will be considered valid. It's recommended to introduce valid oracle threshold and in case if it's not met then report should be considered invalid. There is a trade off between making this threshold too low, e.g. it is now being effectively equal to `1`, which makes the reported value prone to manipulations, and making it too high, e.g. `10` out of `15` active Oracles, when some honest updates can be missed because of threshold not being reached. For reference, protocols like Lido and Kiln which implement LSTs as well have chosen a different approach to handle oracles consensus, not relying on averaging the numbers provided but rather require the exact same report from a majority of oracles with minimum participation required.
3. Since an Oracle running by a third party cannot be fully trusted, e.g. even it was honest for a long time it can collide with an attacker for the each next report. When dealing with such setups, a smoothing approach can be recommended, a some way to remove the extremes, e.g. require at least `5` distinct entries, remove min and max entries by some aggregate score, say the by the average of the validator performance scores (assuming that they are equally scaled), then proceed with the remaining ones.

Kinetiq: Acknowledged.

Cantina Managed: Acknowledged.

5.5.3 `withdrawalDelay` is not set during initialization

Severity: Informational

Context: StakingManager.sol#L66

Description: The `withdrawalDelay` storage variable is used in both `_processConfirmation()` and `cancelWithdrawal()`, but it is not initialized in the `initialize()` function. Instead, it is set via `setWithdrawalDelay()`. If the manager role forgets to call this function, `withdrawalDelay` remains `0`, potentially affecting the execution of dependent functions.

Recommendation: Consider initializing `withdrawalDelay` in the `initialize()` function to ensure a default value or removing it if it is unnecessary.

Kinetiq: Fixed in commit e2bac5c.

Cantina Managed: Fixed in commit e2bac5c by implementing the reviewer's recommendation.

5.5.4 `removeFromWhitelist()` is missing a return value check for `EnumerableSet.remove()`

Severity: Informational

Context: StakingManager.sol#L758

Description: The `removeFromWhitelist()` function allows the `MANAGER_ROLE` to remove accounts from the whitelist and emits the `AddressRemovedFromWhitelist` event to reflect the removed accounts. However, since the return value of `.remove()` is not checked, an account may fail to be removed while still being emitted as removed, leading to potential inconsistencies.

```
function removeFromWhitelist(address[] calldata accounts) external onlyRole(MANAGER_ROLE) {
    for (uint256 i = 0; i < accounts.length; i++) {
        _whitelist.remove(accounts[i]);
        emit AddressRemovedFromWhitelist(accounts[i]);
    }
}
```

Recommendation: Validate the return value of `.remove()`, and make sure the `WhitelistRemoved` event is emitted only in case the return value of `.remove()` is `true`.

Kinetiq: Fixed in commit e2bac5c.

Cantina Managed: Fixed in commit e2bac5c by implementing the reviewer's recommendation.

5.5.5 `StakingManager.rescueToken()`: The non-zero address check for `token` is redundant

Severity: Informational

Context: StakingManager.sol#L937

Description: The `rescueToken()` function allows the `TREASURY_ROLE` to recover tokens sent to `StakingManager`. It includes a check to prevent withdrawing `kHYPE` tokens and tokens with `address(0)`, aiming to disallow the withdrawal of `HYPE`. However, this restriction is unnecessary because `HYPE` is the native currency of HyperLiquid, meaning `rescueToken()` cannot be used to withdraw it.

```
require(token != address(kHYPE) && token != address(0), "Cannot withdraw kHYPE or HYPE");

// For ERC20 tokens
IERC20(token).transfer(treasury, amount);
```

Recommendation: Consider removing the `&& token != address(0)` condition, as it does not impact the security of the function.

Kinetiq: Fixed in commit e2bac5c.

Cantina Managed: Fixed in commit e2bac5c by implementing the reviewer's recommendation.

5.5.6 `OracleManager._checkValidatorBehavior()` can be simplified

Severity: Informational

Context: OracleManager.sol#L169

Description: `_checkValidatorBehavior()` implements threshold checks against `SlashingTolerence` and `PerformanceTolerence`. However, it uses redundant conditions around `previousSlashing` and `previousRewards` as we can see:

```
// Check for slashing anomalies
if (previousSlashing > 0) {
    uint256 slashingDiff = avgSlashAmount > previousSlashing
        ? avgSlashAmount - previousSlashing
        : previousSlashing - avgSlashAmount;

    uint256 slashingBps = Math.mulDiv(slashingDiff, 10000, avgBalance);
    if (slashingBps > SlashingTolerence) {
        return (false, "Slashing change exceeds tolerance");
    }
} else if (avgSlashAmount > 0) {
    uint256 slashingBps = Math.mulDiv(avgSlashAmount, 10000, avgBalance);
    if (slashingBps > SlashingTolerence) {
        return (false, "New slashing exceeds tolerance");
    }
}

// Check for performance anomalies
if (previousRewards > 0) {
    uint256 rewardDiff = avgRewardAmount > previousRewards
        ? avgRewardAmount - previousRewards
        : previousRewards - avgRewardAmount;

    uint256 rewardBps = Math.mulDiv(rewardDiff, 10000, avgBalance);
    if (rewardBps > PerformanceTolerence) {
        return (false, "Performance change exceeds tolerance");
    }
} else if (avgRewardAmount > 0) {
    uint256 rewardBps = Math.mulDiv(avgRewardAmount, 10000, avgBalance);
    if (rewardBps > PerformanceTolerence) {
        return (false, "Initial rewards exceed tolerance");
    }
}
```

Recommendation: Consider simplifying the function by removing these redundant conditions:

```
function _checkValidatorBehavior(
    address, /* validator */
    uint256 previousSlashing,
    uint256 previousRewards,
    uint256 avgSlashAmount,
    uint256 avgRewardAmount,
    uint256 avgBalance
) internal view returns (bool isValid, string memory reason) {
    // Ensure we have a balance to compare against
    if (avgBalance == 0) {
        return (false, "Zero balance");
    }

    uint256 slashingDiff = avgSlashAmount > previousSlashing
        ? avgSlashAmount - previousSlashing
        : previousSlashing - avgSlashAmount;

    uint256 slashingBps = Math.mulDiv(slashingDiff, 10000, avgBalance);
    if (slashingBps > SlashingTolerence) {
        return (false, "Slashing change exceeds tolerance");
    }

    uint256 rewardDiff = avgRewardAmount > previousRewards
        ? avgRewardAmount - previousRewards
        : previousRewards - avgRewardAmount;

    uint256 rewardBps = Math.mulDiv(rewardDiff, 10000, avgBalance);
    if (rewardBps > PerformanceTolerence) {
        return (false, "Performance change exceeds tolerance");
    }

    return (true, "Valid behavior");
}
```

Kinetiq: Fixed in commit e2bac5c.

Cantina Managed: Fixed in commit e2bac5c by introducing a new contract named `ValidatorSanityChecker` used for validation of the input provided by validators.

5.5.7 Rewards reporting can be frontrun

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: When oracle reports new rewards, it increases exchange rate of kHYPE. It is possible to frontrun this action in order to earn part of rewards without staking to much.

1. User stakes right before oracle reports new rewards.
2. User sends kHYPE to another account to pass last stake cooldown.
3. User queues withdrawal and waits 7 days.

This attack can only be profitable if rewards are reported rarely with big amounts of new rewards coming and also if user controls a huge amount of HYPE tokens. Even though frontrunning is currently impossible on HL, this attack can be executed without it by staking funds ahead of reporting.

Recommendation: Make sure you do oracle updates often, so the attack is not attractive.

Kinetiq: Acknowledged.

Cantina Managed: Acknowledged.

5.5.8 `ValidatorManager.requestEmergencyWithdrawal()`: The emergency cool down check is inaccurate

Severity: Informational

Context: ValidatorManager.sol#L331

Description:

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

As we can see, the require statement above is inaccurate since it will be executed even in case `lastEmergencyTime == 0`.

Recommendation: Consider only executing this `require` statement in case `lastEmergencyTime > 0`.

Kinetiq: Fixed in commit ff58fa25.

Cantina Managed: The issue is no longer relevant since `requestEmergencyWithdrawal()` was removed from `ValidatorManager`.

5.5.9 `ValidatorManager.deactivateValidator()`: Discrepancy between documentation and implementation

Severity: Informational

Context: ValidatorManager.sol#L171

Description: In the current version of the code, `ValidatorManager.deactivateValidator()` only deactivates a validator, but the natspec documentation states it is also withdrawing all stake.

Recommendation: Consider fixing this code comment.

Kinetiq: Fixed in commit e2bac5c.

Cantina Managed: Fixed in commit e2bac5c by implementing the reviewer's recommendation.

5.5.10 `ValidatorManager`: `ORACLE_ROLE` should be renamed to `ORACLE_MANAGER_ROLE`

Severity: Informational

Context: ValidatorManager.sol#L53

Description: `ORACLE_ROLE` is an access control role meant to be assigned to the `OracleManager` contract but is currently called `ORACLE_ROLE` which might be misleading.

Recommendation: Consider renaming this variable to `ORACLE_MANAGER_ROLE` instead.

Kinetiq: Fixed in commit e2bac5c.

Cantina Managed: Fixed in commit e2bac5c by implementing the reviewer's recommendation.

5.5.11 `StakingManager.rescueToken()` function won't work with tokens that don't fully support ERC20 tokens

Severity: Informational

Context: StakingManager.sol#L940

Description: `StakingManager.rescueToken()` function assumes that all tokens that are passed into it are fully compatible with erc20 standard. However, some tokens don't fully support erc20, so they don't return boolean on transfer. Because token address is converted to `IERC20`, rescuing of such tokens would fail during the transfer call:

```
function rescueToken(address token, uint256 amount) external onlyRole(TREASURY_ROLE) whenNotPaused {
    require(amount > 0, "Invalid amount");

    // Prevent withdrawing HYPE & kHYPE tokens which are needed for the protocol

    require(token != address(kHYPE) && token != address(0), "Cannot withdraw kHYPE or HYPE");

    // For ERC20 tokens
    IERC20(token).transfer(treasury, amount);

    emit TokenRescued(token, amount, treasury);
}
```

Recommendation: It's recommended to use `SafeERC20` library by OpenZeppelin.

Kinetiq: Fixed in commit 2c53a32.

Cantina Managed: Fixed in commit 2c53a32 by applying the recommendation.

5.5.12 `TODO` Comments Pending

Severity: Informational

Context: KHYPE.sol#L90

Description: In the `KHYPE` contract, there are two TODO comments left in the `_mint` and `_burn` operations indicating that additional logic related to a "mirror token" is yet to be implemented:

```
_mint(to, amount); // TODO update the logic with mirror token
_burn(from, amount); // TODO update the logic with mirror token
```

These serve as internal reminders, but it's helpful to track them as informational flags to ensure they are either resolved or intentionally deferred before deployment.

Recommendation: Consider reviewing and either implementing the intended mirror token logic or removing the TODOs if no further changes are needed.

Kinetiq: Acknowledged.

Cantina Managed: Acknowledged by Kinetiq team.

5.5.13 Unverified potential risks in HyperLiquid interactions

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: During our review, we identified dependencies on HyperLiquid L1 and HyperEVM. However, the full implementation of these components in HyperLiquid's code was not within the official scope of this review. Despite this, incorrect interactions could still lead to issues in the Kinetiq smart contracts.

To assess these dependencies, we referred to HyperLiquid's documentation, but it did not fully address some critical questions. We believe these should be highlighted to raise awareness. While we were able to partially verify some aspects, we recommend further investigation:.

1. Interactions with the `L1Write` Contract:
    - How is the `L1Write` contract configured for HyperEVM?
    - How does HyperEVM dispatch events from `L1Write`?

- Is event execution order guaranteed?
- Since `L1Write` does not return values, how does HyperEVM handle errors triggered by its events?

2. HyperLiquid Staking System:
   - Kinetiq verifies oracle inputs to prevent extreme variable changes. Is this behavior aligned with the actual mechanics of the HyperLiquid staking system?
   - Unlike Ethereum's deposit contract, where the depositor explicitly sets the withdrawal address (as seen in LST platforms), HyperLiquid abstracts this logic. During our review, we found the access control mechanism for staked funds unclear.

Kinetiq: Acknowledged.

Cantina Managed: Acknowledged.

5.5.14 Zero-Fee Withdrawals May Be Possible via Small Amounts

Severity: Informational

Context: StakingManager.sol#L284-L302

Description: In the `StakingManager` contract, the unstake fee is calculated as:

```
uint256 kHYPEFee = Math.mulDiv(kHYPEAmount, unstakeFeeRate, 10000);
```

Due to integer division, very small `kHYPEAmount` values may result in a computed fee of zero. This allows a user to withdraw without paying any protocol fee, effectively bypassing the intended fee logic.

This behavior was demonstrated in a proof of concept where a user withdraws `3` wei of `kHYPE` and receives the full amount without any fee being transferred to the treasury. The protocol treats this as valid and completes the withdrawal. While this may not be economically beneficial under typical conditions, there are considerations worth highlighting:

- The profitability depends on the `kHYPE` to `HYPE` exchange ratio, which in extreme scenarios could make the bypass worthwhile.
- Fee-free transactions may be more attractive in environments where users do not pay gas fees directly (e.g., meta-transactions).
- This is currently the only fee collected by the protocol, so even minor bypasses affect its intended revenue stream.

Proof Of Concept: The following test simulates a user staking and then withdrawing a small amount of `kHYPE` (3 wei). It verifies that the user receives the full amount, no fee is deducted, and the treasury receives zero tokens:

```
function test_wrongMultipleBuffer_feeBypass() public {
    _stake(user, 1 ether / 2);
    _unstake(user, 3);
    IStakingManager.WithdrawalRequest memory wr = stakingManager.withdrawalRequests(user, 0);
    assertEq(wr.kHYPEAmount + wr.kHYPEFee, kHYPE.balanceOf(address(stakingManager)));
    _passTimeAndWithdraw(user, 0);

    assertEq(stakingManager.unstakeFeeRate(), 10);
    assertEq(address(user).balance, 3);
    assertEq(kHYPE.balanceOf(treasury), 0); // Treasury didn't get any fees
}
```

Recommendation: Consider enforcing a minimum fee or disallowing withdrawals that result in a zero fee. This could be done by clamping the fee to a minimum of 1 wei if `kHYPEAmount` is nonzero, or by introducing a minimum withdrawal threshold.

Kinetiq: Acknowledged.

Cantina Managed: Acknowledged by Kinetiq team, as gas cost is bigger than benefit obtained from this transaction.

5.5.15 Consider Setting an Upper Bound for `bufferTarget`

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: In the `StakingManager` contract, the `bufferTarget` value determines how much HYPE is kept unallocated in the staking buffer. This value is configurable and governs how much is withheld from delegation to validators.

While flexibility is useful, there is currently no upper bound enforced on `bufferTarget`. Setting this value excessively high could result in the protocol holding all HYPE in the buffer and staking nothing. In that case, the protocol would not generate any staking yield, which may not be desirable behavior.

Recommendation: Consider introducing a reasonable upper limit to `bufferTarget` to ensure that the protocol always maintains some level of active staking. This can help avoid accidental misconfiguration that might prevent validator delegation entirely.

Kinetiq: Currently, we don't know how many will be set by the institution. I suggest not setting an upper bound and leaving space for customization.

Cantina Managed: Acknowledged by Kinetiq team.

5.5.16 `minStakeAmount` Should Align With Divisibility Constraint

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: In the `StakingManager` contract, line 455 enforces that all stake amounts must be divisible by `1e10`:

```
require(amount % 1e10 == 0, "Amount must be divisible by 1e10");
```

However, the `minStakeAmount` variable — which governs the minimum amount a user can stake — is only checked to be greater than zero during initialization. This allows `minStakeAmount` to be set to a value that violates the divisibility constraint, which could cause revert when users attempt to stake exactly the minimum.

Recommendation: Consider enforcing `minStakeAmount % 1e10 == 0` in the initialization logic to ensure consistency with the runtime check applied during staking. This can help avoid misconfiguration.

Kinetiq: Fixed in commit 5d9e97b.

Cantina Managed: Partially fixed in commit 5d9e97b by enforcing the restriction on `setMinStakeAmount` function. However `initialize` function does not enforce it.