SPEARBIT

# Kinetiq LST Protocol Security Review

**Auditors**

0xRajeev, Lead Security Researcher

Optimum, Lead Security Researcher

Rvierdiiev, Security Researcher

Kamensec, Associate Security Researcher

**Report prepared by:** Lucas Goiriz

December 13, 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 is a protocol that scores validators and distributes staking allocations based on performance, utilizing HYPE tokens to manage delegation.

*Disclaimer*: This security review does not guarantee against a hack. It is a snapshot in time of Kinetiq LST Protocol 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 4 days in total, Kinetiq engaged with Spearbit to review the launch protocol. In this period of time a total of **27** issues were found.

**Summary**

| | |
|---|---|
| **Project Name** | Kinetiq |
| **Repository** | launch |
| **Commit** | 019c2a0e |
| **Type of Project** | Liquid Staking, Perpetuals |
| **Audit Timeline** | Nov 23rd to Nov 27th |

**Issues Found**

| **Severity** | **Count** | **Fixed** | **Acknowledged** |
|---|---|---|---|
| Critical Risk | 0 | 0 | 0 |
| High Risk | 1 | 0 | 0 |
| Medium Risk | 1 | 0 | 0 |
| Low Risk | 17 | 0 | 0 |
| Gas Optimizations | 0 | 0 | 0 |
| Informational | 8 | 0 | 0 |
| **Total** | **27** | **11** | **16** |

## 4.1 Scope

The security review had the following components in scope for launch on commit hash 019c2a0e:

```
src
├── base
│   ├── EIP712Verifier.sol
│   ├── LSTPayments.sol
│   └── LSTState.sol
├── BlockedWithdrawalQueue.sol
├── EXLST.sol
├── EXManager.sol
├── EXRouter.sol
├── GlobalConfig.sol
├── HIP3StakingManager.sol
└── lib
    ├── Constants.sol
    ├── Errors.sol
    └── HIP3L1Write.sol
```

# 5 Findings

## 5.1 High Risk

### 5.1.1 Operator bond can not be recovered and will be locked in `ExManager`

**Severity:** High Risk

**Context:** EXManager.sol#L257

**Description:** `ExManager` requires operators to deposit HYPE tokens equal to `opBond` as a ”skin-in-the-game” mechanism. When an operator calls `ExManager.bond()`, the contract deposits the HYPE bond and mints the corresponding amount of `exLST` tokens. These `exLST` tokens are minted to the `ExManager` contract itself, not to the operator, ensuring that the operator cannot withdraw or use the bonded funds until the contract is unwound.

In the current implementation, however, once the contract is unwound, the operator has no mechanism to redeem or claim back their HYPE bond. The `exLST` tokens minted to `ExManager` remain locked inside `HIP3StakingManager`, making the operator bond permanently inaccessible.

This results in the bonded assets being irrecoverably locked, which contradicts the intended lifecycle of the operator bond and creates a significant economic burden for operators.

While the impact is limited for the upcoming permissioned deployment-where `opBond` is relatively small-it becomes severe in the permissionless version, where `opBond` may be as high as 10% of 500,000 HYPE, potentially resulting in substantial permanent losses per operator.

**Recommendation:** Consider allowing the operator to burn the corresponding amount of `exLST` tokens and claim back the HYPE bond after unwinding.

**Kinetiq:** Fixed in PR 50.

**Spearbit:** Fix verified.

## 5.2 Medium Risk

### 5.2.1 Withdrawal Delay Can Be Bypassed When L1 Operations Processed More Than Once Per 24 Hours

**Severity:** Medium Risk

**Context:** HIP3StakingManager.sol#L12

**Description:** HyperCore enforces a 24-hour cooldown after staking before withdrawals from the same validator can be processed. The `StakingManager` processes L1 operations (withdrawals first, then deposits) via `processL1Operations()`, but there is no on-chain enforcement that this function is called at most once per 24 hours.

If `processL1Operations()` is called more frequently than every 24 hours, the following scenario can occur:

1. User queues a withdrawal at $T$ = 0.
2. Operator processes L1 operations at $T$ = 0 (withdrawal sent to L1).
3. L1 withdrawal fails because a deposit was made within last 24 hours.
4. Operator retries with `queueL1Operation()` at $T$ = 6 days (withdrawal now succeeds on L1).
5. User calls `confirmWithdrawal()` at $T$ = 7 days.

The issue is that the withdrawal delay check in `confirmWithdrawal()` uses the original `request.timestamp` from when the withdrawal was queued:

```solidity
// src/EXManager.sol (confirmWithdrawal)
if (block.timestamp < request.timestamp + withdrawalDelay) {
    revert Errors.WithdrawalDelayNotMet();
}
```

When the operator retries a failed L1 operation, the `request.timestamp` is NOT updated. This means:

- Original queue time: $T$ = 0.
- Retry succeeds: $T$ = 6 days.
- User confirms: $T$ = 7 days (only 1 day after actual L1 processing).

The 7-day withdrawal delay was intended to ensure funds are fully available on L1 before user confirmation, but if the L1 operation was delayed and retried, the actual L1 processing may have occurred much later than the original queue timestamp.

**Impact Explanation:** Users may be able to confirm withdrawals before the full intended delay has passed from the actual L1 operation.

**Likelihood Explanation:** Low likelihood because:

- Operator is a trusted role expected to follow operational guidelines.
- L1 operation failures that require retry are edge cases.
- The current implementation relies on off-chain operational discipline.

However, the scenario becomes more likely if:

- Multiple operators or automated systems process operations.
- L1 congestion causes frequent operation failures.
- Operational mistakes lead to processing more than once per day.

**Recommendation:** Add an on-chain cooldown enforcement for `processL1Operations()`.

**Kinetiq:** Acknowledged. Looking to minimize changes to the LST staking manager as much as possible for our markets initial version.

Given our operator on the `HIP3StakingManager` will only call to `processL1Operations()` every 36 hours, this should be ok for now.

**Spearbit:** Acknowledged.

## 5.3 Low Risk

### 5.3.1 Members of withdrawal queue do not receive HIP3 fee

**Severity:** Low Risk

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

**Description:** When fees are distributed using `stakeFees()` function, the `_exGhostLST` supply is increased based on the stake amount, while new `exLST` tokens are not minted. As a result, the `exLST : _exGhostLST` rate decreases, and the Hype share of active stakers increases in value.

When a withdrawal enters the blocking queue, some amount of `exLST` is burned and the corresponding `_exGhostLST` shares are transferred to the queue. Once this happens, subsequent `stakeFees()` calls do not increase the Hype value for users in the withdrawal queue, because their `_exGhostLST` balance is fixed and does not grow. Therefore, queued members do not receive fees earned by the deployed market during the time they are waiting.

**Recommendation:** As this is a known limitation, it is recommended to document the behavior clearly so users understand that fees accumulated while in the withdrawal queue will not be attributed to them.

**Kinetiq:** Fixed in PR 52.

**Spearbit:** Fix verified. The team has added a comment that documents the problem.

### 5.3.2 Withdrawal inside blocking queue may be processed with updated fee share

**Severity:** Low Risk

**Context:** BlockedWithdrawalQueue.sol#L186

**Description:** `BlockedWithdrawalQueue` calculates unstaking fees at withdrawal processing time rather than when the withdrawal is queued. If the fee rate changes while the withdrawal is pending, the user will be charged using the *updated* unstaking fee rate. This may result in the user paying a different fee than expected at the time they initiated the withdrawal.

**Recommendation:** Store the unstaking fee rate in the `BlockedWithdrawal` struct at the moment the withdrawal is queued, and use that stored value when calculating fees during withdrawal processing.

**Kinetiq:** Acknowledged. Will address in full permissionless roll out given don't expect significant changes to unstake fee rate for our markets version.

**Spearbit:** Acknowledged.

### 5.3.3 Slippage protection is missing for the `withdrawOnceLive`

**Severity:** Low Risk

**Context:** EXManager.sol#L550-L555

**Description:** When a user withdraws with `withdrawOnceLive()`, they intend to redeem a specific `shares` amount of `exLST`. However, depending on `availableWithdrawals()` amount, part-or even all-of the withdrawal may become blocked and placed into the `BlockedWithdrawalQueue` for an unknown period of time.

Currently, there is no slippage protection mechanism. In the worst-case scenario, the entire withdrawal request can be forced into the blocked queue, which may be highly unexpected and undesirable for the user.

**Recommendation:** Introduce an additional parameter (e.g., `maxBlockedShares`) to define the maximum amount of `shares` the user is willing to have queued. If more would be blocked, revert the transaction to protect the user from unintended outcomes.

**Kinetiq:** Fixed in PR 59.

**Spearbit:** Fix verified.

### 5.3.4 Tokens can't be rescued from `HIP3StakingManager` contract

**Severity:** Low Risk

**Context:** EXManager.sol#L207-L209

**Description:** `StakingManager` includes a `rescueToken()` function that sends accidentally stuck tokens to the treasury:

```
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), "Cannot withdraw kHYPE");

    // For ERC20 tokens - use safeTransfer instead of transfer
    IERC20(token).safeTransfer(treasury, amount);

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

For the `HIP3StakingManager`, the treasury is the `ExManager` contract. However, `ExManager` itself does not implement any rescue mechanism, so tokens sent to the `ExManager` contract cannot be recovered.

**Recommendation:** Implement a `rescueToken()` function in the `ExManager` contract to allow the treasury to recover rescued tokens. Also make sure, it's not possible to withdraw `exLST` and `kHype` tokens.

**Kinetiq:** Acknowledged.

**Spearbit:** Acknowledged.

### 5.3.5 Operator Bond Undercollateralized During Slashing Due to Share-Based Accounting

**Severity:** Low Risk

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

**Description:** The operator bond in `EXManager` is tracked in shares (exLST) rather than underlying HYPE amounts. When bonding occurs via `bond()`, the operator deposits KHYPE which is converted to exLST shares at the current exchange rate. The bond requirement check on line 259 validates that minted shares meet the `opBond` minimum:

```
// src/EXManager.sol:259
if (bondShares < opBond) revert Errors.InsufficientBond();
```

However, the HIP-3 self-bonding requirement and slashing mechanism operate on underlying HYPE amounts, not shares. If a slashing event occurs on the validator before or during the FUNDING phase, the HYPE:KHYPE exchange rate diverges from 1:1, meaning the operator's share-denominated bond represents less actual HYPE than intended.

The issue manifests in two scenarios:

1. Pre-bond slashing: If the validator experiences slashing before `bond()` is called, the KHYPE deposited converts to fewer HYPE equivalent, but shares still meet `opBond` minimum.
2. Post-bond slashing: After bonding, if slashing occurs, the operator's shares represent proportionally less HYPE, but their ”skin in the game” relative to user deposits decreases since slashing affects the underlying proportionally.

Since slashing is denominated in HYPE (the underlying asset), the operator's effective exposure to slashing risk is reduced compared to regular users when the bond was established during a period of exchange rate deviation.

**Impact Explanation:** Impact is limited because:

- The protocol uses permissioned/trusted validators.
- Slashing events are rare in practice.
- The bond percentage (~10%) provides some buffer.

**Likelihood Explanation:** Low likelihood because:

- Validator selection is permissioned and trusted.
- Slashing events on HyperLiquid are rare.
- The KHYPE:HYPE exchange rate is relatively stable during bonding phases.
- Bonding typically occurs at protocol initialization when exchange rate is close to 1:1.

However, the scenario is technically possible if:

- Slashing occurs on the linked validator before bonding.
- The StakingManager/StakingAccountant used for KHYPE has experienced slashing.

**Recommendation:** Consider tracking the operator bond in terms of underlying HYPE value rather than shares.

**Kinetiq:** Acknowledged. Slashing on kHYPE not really worried about.

**Spearbit:** Acknowledged.

### 5.3.6 Unbounded Chunk Size Can DoS Blocked Withdrawal Queue Processing

**Severity:** Low Risk

**Context:** BlockedWithdrawalQueue.sol#L367-L371

**Description:** The `processBlockedWithdrawals()` function in `BlockedWithdrawalQueue` uses a `chunkSize` parameter to limit size of blocked withdrawals are processed per call. However, there is no upper bound validation on this parameter:

```
if (withdrawableShares < Math.min(nextItem.remainingGhostShares, chunkSize)) {
    break;
}
```

The issue is on line 197 where the loop breaks if a single withdrawal's `withdrawableShares` is less than the minimum of either the `nextItem.remainingGhostShares` or `chunkSize`.

A whale user with a very large blocked withdrawal could:

1. Queue a massive withdrawal that is larger than typical `availableHype`.
2. This withdrawal sits at the head of the blocked queue.
3. Every call to `processBlockedWithdrawals()` hits the break condition immediately.
4. Smaller withdrawals behind the whale are stuck indefinitely.

The lack of an upper bound on `chunkSize` exacerbates this because:

- Operator might set very large `chunkSize` expecting to process many withdrawals.
- But if the first item is a whale, nothing gets processed regardless of `chunkSize`.
- No mechanism to skip or partially process the blocking withdrawal.

**Impact Explanation:** A single large withdrawal can block the entire withdrawal queue for other users.

The impact is constrained because;

- The operator can wait for sufficient HYPE to accumulate.
- The chunk size can be reduced to unblock the DOS.

**Likelihood Explanation:** Low likelihood because:

- Requires a user with holdings large enough to exceed typical available HYPE.
- Protocol would need to set unrealistically large chunk size value.

**Recommendation:** Add upper bound on chunk size.

**Kinetiq:** Acknowledged. Will address in fully permissionless roll out.

**Spearbit:** Acknowledged.

### 5.3.7 Compromised operator can bypass enforced whitelist to deposit in `FUNDING` phase

**Severity:** Low Risk

**Context:** EXManager.sol#L254-L257, EXManager.sol#L405-L407, EXManager.sol#L770

**Description:** Protocol implements an optional whitelist enforcement with tier-based user/global caps for deposits during the `FUNDING` phase. This logic is implemented in `_enforceWhitelistMintCapIfEnabled(...)` called from `depositWhileFunding(...)`. However, to allow the operator to deposit its bond, `_enforceWhitelistMintCapIfEnabled(...)` returns successfully if `hasRole(OPERATOR_ROLE, sender)`.

This operator exception for whitelisting assumes a trusted operator who does not deposit in the `FUNDING` phase. However, a compromised operator can exploit this exception to infinitely mint in `FUNDING` phase thereby breaking the expected whitelisting enforcement. While the likelihood of a compromised operator is low in this permissioned setup, this trust assumption will be invalid in any future permissionless version.

**Recommendation:** Consider refactoring logic to skip whitelist enforcement only during `UNBONDED` phase for operator bond payment instead of a permanent exception irrespective of the phase.

**Kinetiq:** Fixed in PR 58.

**Spearbit:** Fix verified. Reviewed that PR 58 mitigates the issue by replacing `hasRole(OPERATOR_ROLE, sender)` as one of the early exit conditions in `_enforceWhitelistMintCapIfEnabled(...)` with `_isBonding(sender)`, which additionally checks that `exPhase == EXPhase.FUNDING` and `_reserves(address(exLST)) < opBond`. This retrofits the recommendation into the ExManager bond call with the initial assumption on the values of `opBond` and `_reserves(address(exLST))`.

### 5.3.8 Unenforced staking requirement of 500K HYPE may lead to unexpected behavior

**Severity:** Low Risk

**Context:** GlobalConfig.sol#L62, GlobalConfig.sol#L92-L96

**Description:** Hyperliquid's HIP-3, which enables permissionless deployment of custom perpetual futures markets on the HyperCore layer, enforces a 500K HYPE staking requirement to serve as a bond for market deployers and deter any malicious behavior. This stake must be maintained at all times to operate a HIP-3 market. If the stake drops below 500K HYPE (e.g., due to attempted unstaking), the associated markets become inoperable forcing a wind-down, halting trading and other unexpected behavior.

To adhere to this staking requirement, the protocol enforces a minimum reserve requirement in the `LIVE` phase using `globalConfig.minHypeStake()`. However, the `minHypeStake` set in `GlobalConfig.initialize()` or in the setter `GlobalConfig.setMinHypeStake(...)` do not strictly enforce the 500K HYPE lower-bound requirement and instead assume that initialization or `CONFIG_ADMIN_ROLE` will set this appropriately. This is supposedly to allow flexibility given that HIP-3 specifies that: ”*The staking requirement for mainnet will be 500K HYPE. This requirement is expected to decrease over time as the infrastructure matures*”.

In the low likelihood that `minHypeStake` is set to a lower than 500K HYPE, this will cause the associated market to become inoperable leading to unexpected behavior for users including potential loss of fees/funds.

**Recommendation:** Consider strictly enforcing the current HIP-3 500K HYPE staking requirement with the flexibility to change this in future contract upgrades.

**Kinetiq:** Acknowledged, but prefer to not upgrade given using similar security setup with global config contract roles as in the user able to upgrade contract.

**Spearbit:** Acknowledged.

### 5.3.9 Missing check allows tokens to get stuck in `EXRouter`

**Severity:** Low Risk

**Context:** EXManager.sol#L427, EXManager.sol#L533, EXRouter.sol#L86-L112

**Description:** The withdraw functions in `EXManager` perform

```
if (recipient == address(this)) revert Errors.InvalidRecipient()
```

check on recipient to prevent tokens from accidentally getting stuck in the contract. However, `EXRouter`, which implements user convenience function wrappers is missing a similar check in `withdraw(...)`. This allows user tokens to accidentally get stuck in `EXRouter`.

**Recommendation:** Consider implementing a

```
if (recipient == address(this)) revert Errors.InvalidRecipient()
```

check in `EXRouter.withdraw(...)`.

**Kinetiq:** Fixed in PR 56.

**Spearbit:** Fix verified.

### 5.3.10 Incorrect value emitted in `Bonded` event may affect offchain tracking

**Severity:** Low Risk

**Context:** EXManager.sol#L257-L264, IEXManager.sol#L121-L124

**Description:** `ExManager.bond()` emits a `Bonded` event to capture the amount of `exLST` shares escrowed for operator bonded capital. This event uses the `shares` value instead of `opBond`. However, `shares` minted may be in excess of `opBond` in which case `bond()` returns any excess `exLST` shares back to the operator. Given this, the correct value to be emitted is `opBond` and not `shares`. This incorrect value emitted in `Bonded` event may affect any offchain tracking of operator bonded capital.

**Recommendation:** Consider emitting `opBond` value instead of `shares`.

**Kinetiq:** Fixed in PR 55.

**Spearbit:** Fix verified.

### 5.3.11 Missing upper bound check for `minimumWithdrawWhenLive` may prevent user withdrawals

**Severity:** Low Risk

**Context:** EXManager.sol#L537-L540, EXManager.sol#L841-L844

**Description:** Protocol implements a `minimumWithdrawWhenLive`, which is the minimum amount of shares required for withdrawal when live. The motivating reason is to prevent small dust withdrawals that may supposedly overwhelm the system. This `minimumWithdrawWhenLive` is set by `MANAGER_ROLE` using `setMinimumWithdrawWhenLive(...)` and enforced in `withdrawOnceLive(...)` by reverting when `shares < minimumWithdrawWhenLive`.

However, the `setMinimumWithdrawWhenLive(...)` setter is missing a reasonable upper bound check which allows the (compromised) manager to (intentionally) accidentally set `minimumWithdrawWhenLive` to a large enough value to prevent all/many withdrawals.

**Recommendation:** Consider implementing a a reasonable upper bound constant for `minimumWithdrawWhenLive`, which can be checked in `setMinimumWithdrawWhenLive(...)`.

**Kinetiq:** Acknowledged. Will address in the fully permissionless version.

**Spearbit:** Acknowledged.

### 5.3.12 `WOUND_DOWN` Phase Allows New Withdrawals to Bypass Users In Blocked Queue FIFO Ordering

**Severity:** Low Risk

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

**Description:** The `BlockedWithdrawalQueue` enforces FIFO (first-in-first-out) ordering during the LIVE phase to ensure fair withdrawal processing when liquidity is constrained. When `availableWithdrawals()` returns 0 due to an existing blocked queue, all new withdrawals are forced to join the blocked queue:

```
// src/EXManager.sol:859-863
function availableWithdrawals() public view returns (uint256 availableShares) {
    if (exPhase == EXPhase.LIVE) {
        if (blockedWithdrawalQueue.totalBlockedQueue() > 0) {
            return 0;  // Forces new withdrawals to join blocked queue
        }
        // ... calculate based on minHypeStake
    } else {
        return _totalSupply();  // WOUND_DOWN: ALL shares available
    }
}
```

However, when the protocol transitions to `WOUND_DOWN` phase, this FIFO enforcement is bypassed. New withdrawals via `withdrawOnceLive()` go directly to the StakingManager queue, while users already in the blocked queue must wait for `processBlockedWithdrawals()` to be called first.

The blocked queue still processes in FIFO order internally:

```
// src/BlockedWithdrawalQueue.sol:191-193
// Process blocked withdrawals in FIFO order
for (; i < n; i++) {
    BlockedWithdrawal storage nextItem = blockedWithdrawals[i];
    // ...
}
```

But new WOUND_DOWN withdrawals completely bypass this queue, allowing latecomers to jump ahead of users who have been waiting in the blocked queue.

**Impact Explanation:** Users who withdrew first and were placed in the blocked queue during LIVE phase can be jumped by users who withdraw after the `WOUND_DOWN` transition. Although intention is that block queue processing is allowed to complete in its entirety after unwind, the assumption is that both queues direct withdrawal and blocked queue can finalize immediately after `WOUND_DOWN`. However, there is a withdrawal limit on hyperliquid of 5 per address, and since each block is processed by the staking manager every 36 hours (a business limitation), so if more than 3 withdrawals are not finalized, its possible the direct withdrawal is placed in front within the 5 withdrawals, whilst the next withdrawal from the blocked queue is in the next batch. This violates the fairness that the blocked queue was designed to enforce forcing blocked withdrawals to be delayed by 36 hours (or block processing time).

**Likelihood Explanation:** Medium - requires the protocol to enter `WOUND_DOWN` phase while a blocked queue exists. The operator has a 7-day delay between `setUnwindPhase()` and `unwind()`, during which the blocked queue could theoretically be processed. However, if liquidity remains constrained during this period, the blocked queue may persist into `WOUND_DOWN` and users are in fact benefited by delaying withdrawal until unwinding period is activated.

**Recommendation:** Modify `availableWithdrawals()` to respect the blocked queue in `WOUND_DOWN` phase, or add a mechanism to ensure blocked queue users are processed before new `WOUND_DOWN` withdrawals.

**Kinetiq:** Fixed in PR 60.

**Spearbit:** Fix verified.

### 5.3.13 L1 Account Activation Not Verified Before CoreWriter Actions

**Severity:** Low Risk

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

**Description:** HyperCore accounts must be activated before they can execute CoreWriter actions such as staking deposits. According to HyperLiquid documentation, activation occurs when an account receives its first spot transfer from an already-activated sender, which charges a 1 USDC activation fee.

The `StakingManager.stake()` function calls `_distributeStake()` with `OperationType.UserDeposit`, which performs two L1 operations without verifying the StakingManager's L1 account is activated:

```
// lib/lst/src/StakingManager.sol:249
function stake() external payable nonReentrant whenNotPaused returns (uint256 kHYPEAmount) {
    // ... validation ...
    kHYPE.mint(msg.sender, kHYPEAmount);
    _distributeStake(msg.value, OperationType.UserDeposit);  // No activation check
    stakingAccountant.recordStake(msg.value);
}
```

Within `_distributeStake()` for `UserDeposit`:

```
// lib/lst/src/StakingManager.sol:482-490
// 1. Move HYPE from EVM to spot balance on L1
(bool success,) = payable(L1_HYPE_CONTRACT).call{value: amount}("");
require(success, "Failed to send HYPE to L1");

// 2. Move from spot balance to staking balance (CoreWriter action)
uint256 truncatedAmount = _convertTo8Decimals(amount, false);
L1Write.sendCDeposit(uint64(truncatedAmount));  // Requires activated account!

// 3. Queue the delegation operation
_queueL1Operation(validator, truncatedAmount, operationType);
```

Then `L1_HYPE_CONTRACT.call{value: amount}` is executed, this triggers

```
CoreExecution.executeNativeTransfer()
```

which has the `initAccountWithToken` modifier. However, this modifier calls `_initializeAccount()` which checks if the account already exists on Core:

```
// hyper-evm-lib/test/simulation/hyper-core/CoreState.sol:183-190
RealL1Read.CoreUserExists memory coreUserExists = RealL1Read.coreUserExists(_account);
if (!coreUserExists.exists && !force) {
    return;  // Early return - does NOT activate!
}
_initializedAccounts[_account] = true;
account.activated = true;
```

If `coreUserExists` returns false (account never received a spot transfer), the function returns early without activating. The HYPE then goes to latent balance instead of usable spot balance:

```
// hyper-evm-lib/test/simulation/hyper-core/CoreExecution.sol:51-55
if (_accounts[from].activated) {
    _accounts[from].spot[HYPE_TOKEN_INDEX] += (value / 1e10).toUint64();
} else {
    _latentSpotBalance[from][HYPE_TOKEN_INDEX] += (value / 1e10).toUint64();  // Unusable!
}
```

The subsequent `L1Write.sendCDeposit()` call attempts to move funds from spot balance to staking balance. CoreWriter actions from unactivated accounts fail silently - the EVM transaction succeeds but the L1 operation is not executed (guarded by `whenActivated` modifier which returns early).

**Impact Explanation:** If the StakingManager's L1 account is not activated before the first staking operation:

1. Users call `stake()` and receive kHYPE tokens (EVM state updated).
2. HYPE is sent to L1 but lands in latent balance (not usable spot balance).
3. `sendCDeposit()` silently fails (account not activated).
4. `sendTokenDelegate()` silently fails (account not activated).
5. Result: Users hold kHYPE backed by HYPE that is stuck in latent balance and never actually staked.

This creates unbacked LST shares and in some cases underlying reverts during market launch if minimum hype balances are insufficient.

**Likelihood Explanation:** Low - the deployment script provides an option to activate the account by sending USDC, and operators are expected to follow proper initialization procedures. However, this is an optional step with no on-chain enforcement, relying entirely on off-chain operational discipline.

**Recommendation:** Add the `coreUserExists` precompile to `L1Read.sol` and verify account activation before the first CoreWriter action.

**Kinetiq:** Acknowledged. Will add into deploy scripts for our markets launch.

**Spearbit:** Acknowledged. Be cautious in permissionless case we might need something in a factory contract that checks these things if they use some non standard staking manager contract.

### 5.3.14 `EXManager` and `BlockedWithdrawalQueue` lack support of cancelled withdrawal requests

**Severity:** Low Risk

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

**Description:** `StakingManager` allows users to queue withdrawal requests via `queueWithdrawal()`. This function transfers LST tokens from the caller and creates a corresponding withdrawal request object. A privileged account with `MANAGER_ROLE` may later cancel a withdrawal request by calling `cancelWithdrawal()`, which reverses the effects of `queueWithdrawal()` by deleting the request object and refunding the LST tokens to the original requester.

However, both `EXManager` and `BlockedWithdrawalQueue` do not implement any logic to handle cancelled withdrawal requests. As a result, if a withdrawal initiated by these contracts is cancelled, the refunded LST tokens will be sent back to them without any mechanism to correctly account for, forward, or recover these funds. This leads to a loss of funds or stuck tokens for users interacting through these contracts.

While the impact is high due to the potential for permanent fund loss, the likelihood is assumed to be low because cancellations performed by the manager on requests originating from these contracts are not expected operationally.

**Recommendation:** Add explicit support in both `EXManager` and `BlockedWithdrawalQueue` to properly handle LST refunds triggered by `cancelWithdrawal()`.

**Kinetiq:** Acknowledged. Won't cancel withdraws given exLST accounting.

**Spearbit:** Acknowledged.

### 5.3.15 Compounding precision loss in `_HYPEToEXLST` increasingly delays withdrawals as protocol yields grow

**Severity:** Low Risk

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

**Description:** In `EXManager.availableWithdrawals()`, the calculation of available shares for immediate withdrawal undergoes two sequential rounding-down operations in `_HYPEToEXLST()`:

```
// LSTPayments.sol:194-203
function _HYPEToEXLST(
    uint256 hypeAmount,
    IStakingAccountant reserveStakingAccountant,
    address reserveLST,
    uint256 exLstTotalSupply
) internal view returns (uint256 shares) {
    uint256 lstShares = _HYPEToLST(reserveStakingAccountant, hypeAmount);  // Rounds down
    uint256 lstReserves = _reserves(reserveLST);
    shares = _calcShares(lstShares, lstReserves, exLstTotalSupply);        // Rounds down
}
```

This causes `availableWithdrawals()` to return a value slightly lower than the theoretical maximum. As the protocol's exchange rate increases due to yield accrual, the HYPE value represented by each lost share grows, meaning users lose access to increasingly valuable immediate withdrawal capacity.

If the delta available hype is 100,000 (600k - 500k minimum), exchange rate is `100e18`, so `HYPEAMOUNT` < 100 will round down. As the exchange rate grows this rounding becomes more pronounces.

**Recommendation:** Acknowledge this behaviour, this can be limited with continuous reward claiming to reduce exchange rate growth having an impact on withdrawing the excess hype amounts above minimum.

**Kinetiq:** Acknowledged. `minHypeStake` likely set to slightly > 500k in case we miss rounding issues elsewhere

**Spearbit:** Acknowledged.

### 5.3.16 `initEXManager()` might be front ran to disrupt deployment

**Severity:** Low Risk

**Context:** HIP3StakingManager.sol#L23

**Description:** `HIP3StakingManager` is extending `StakingManager` and is meant to be used as an upgradeable contract under a proxy. In the current version of the code there are no deployment scripts but we were able to find the following function in `HIP3StakingManagerTest`:

```
function _initHIP3StakingManager(address proxyAdmin, address proxy, address implementation)
↪   internal {
    uint256 minStakeAmount = 1 ether;
    uint256 maxStakeAmount = 10 ether;
    uint256 stakingLimit = 1000 ether;
    uint64 hypeTokenId = 1105;

    // Act - Initialize the contract
    vm.startPrank(admin);
    ProxyAdmin(proxyAdmin)
        .upgradeAndCall(
            ITransparentUpgradeableProxy(proxy),
            implementation,
            abi.encodeCall(
                StakingManager.initialize,
                (
                    admin,
                    operator,
                    manager,
                    pauserRegistry,
                    kHYPE,
                    validatorManager,
                    stakingAccountant,
                    treasury,
                    minStakeAmount,
                    maxStakeAmount,
                    stakingLimit,
                    hypeTokenId
                )
            )
        );
    HIP3StakingManager(payable(proxy)).initEXManager(exchangeManager);
    vm.stopPrank();
}
```

As we can see, `StakingManager.initialize()` is followed by a call to `initEXManager()`, although this is only a test script, a similar deployment scripts might be used, which will not ensure no transactions in between the two calls.

Front runners might be able to call `initEXManager()` right before the deployment script, and set the `exManager` to their desired address. note that it will call the script call to `initEXManager()` to revert which might be spotted by the deployers which will try to deploy again.

**Recommendation:** Consider either restricting `initEXManager()` to the manager or operator defined inside `StakingManager.initialize()` only, or alternatively, declare it as the main `initialize()` function, which will call `StakingManager.initialize()`.

**Kinetiq:** Acknowledged.

**Spearbit:** Acknowledged.

#### 5.3.17 `ExManager.confirmWithdrawal()` can be front ran causing a revert for the original caller

**Severity:** Low Risk

**Context:** EXManager.sol#L582

**Description:** `confirmWithdrawal()` is permissionless and can be called by anyone for any `recipient`. This design introduces a griefing vector: a frontrunner can pre-emptively confirm another user's withdrawal. While the attacker's confirmation succeeds, the original user's transaction attempting to confirm the same withdrawal will then revert.

This issue also affects `EXRouter.confirmAll()`. Since a single failing withdrawal confirmation causes the entire loop to revert, an attacker can repeatedly trigger failures and prevent `confirmAll()` from completing successfully.

**Recommendation:** Modify `ExManager.confirmWithdrawal()` to avoid reverting when the withdrawal is already confirmed or otherwise invalid. Instead, follow a non-reverting pattern similar to `BlockedWithdrawalQueue.confirmBlockedWithdrawal()`, allowing the function to return gracefully. This prevents the griefing vector and ensures `confirmAll()` can complete even when some confirmations are no-ops.

**Kinetiq:** Fixed in PR 51.

**Spearbit:** Fix verified.

## 5.4 Informational

### 5.4.1 Missing zero-address checks

**Severity:** Informational

**Context:** GlobalConfig.sol#L66-L67

**Description:** While the protocol implements the best-practice of performing zero-address checks in most places, there are two missing checks for `_globalAdmin` and `_configAdmin`.

**Recommendation:** Consider adding the missing zero-address checks.

**Kinetiq:** Fixed in PR 54.

**Spearbit:** Fix verified. Reviewed that PR 54 adds the missing zero-address checks.

### 5.4.2 Missing checks for user-provided addresses are risky

**Severity:** Informational

**Context:** EXRouter.sol#L55, EXRouter.sol#L64, EXRouter.sol#L104, EXRouter.sol#L119, EXRouter.sol#L128, EXRouter.sol#L142

**Description:** There are functions that accept user-provided addresses to later make external calls on them. While they do not seem to pose an immediate risk given the current context of the calls, avoiding arbitrary external calls by enforcing checks on user-provided addresses is a security best-practice.

**Recommendation:** Consider adding the missing checks for user-provided addresses.

**Kinetiq:** Acknowledged. In full permissionless version will have factory registry to check against in ex router.

**Spearbit:** Acknowledged.

### 5.4.3 Missing upper bound check for `unwindDelay` may prevent markets from being wound down

**Severity:** Informational

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

**Description:** Protocol implements a `unwindDelay` to enforce a delay from the time an operator calls `setUnwindPhase(...)` to when it can successfully call `unwind()` for winding down a market. The motivating reason is to allow a window of opportunity for users to react to the market being wound down.

However, the `setUnwindDelay(...)` setter is missing a reasonable upper bound check, which allows the (compromised) `CONFIG_ADMIN_ROLE` to (intentionally) accidentally set `unwindDelay` to a large enough value to prevent market wind downs.

**Recommendation:** Consider implementing a a reasonable upper bound constant for `unwindDelay`, which can be checked in `setUnwindDelay(...)`.

**Kinetiq:** Fixed in PR 53.

**Spearbit:** Fix verified. Reviewed that PR 53 adds `MAX_UNWIND_DELAY = 90 days` and enforces it in `initialize(...)` and `setUnwindDelay(...)`.

### 5.4.4 Compromised/Malfunctional privileged roles/addresses can cause unexpected behavior

**Severity:** Informational

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

**Description:** The protocol has several privileged roles such as `MANAGER_ROLE`, `OPERATOR_ROLE` and `CONFIG_ADMIN_ROLE` along with other privileged addresses such as `whitelister` and `exWalletAdmin`, which are responsible for critical administrative/operational actions.

Any compromised/malfunctional privileged role/address can cause unexpected behavior if it calls its authorized functions with incorrect values or in an incorrect manner accidentally/intentionally. While there are some safeguard checks to prevent such behavior, these nevertheless pose a centralization risk in this permissioned setup.

**Recommendation:** Consider:

1. Implementing the highest levels of operational security for privileged role management.
2. Documenting and highlighting the assumptions and risks for protocol users.

**Kinetiq:** Acknowledged.

**Spearbit:** Acknowledged.

### 5.4.5 Potential temporary denial of service of withdrawals due to reverts from `exTreasury`

**Severity:** Informational

**Context:** BlockedWithdrawalQueue.sol#L268, EXManager.sol#L595

**Description:** Both `BlockedWithdrawalQueue.confirmBlockedWithdrawal()` and `ExManager.confirmWithdrawal()` transfers HYPE to two different addresses, the first is the recipient of the withdrawal and the second is `exTreasury` that receives the fees.

However, there is a potential scenario in which `exTreasury` will revert receiving the fees. In this case withdrawals will be blocked until `exTreasury` will be replaced.

**Recommendation:** Consider changing the code so that fee payments won't revert the entire transaction.

**Kinetiq:** Acknowledged. Should be ok given that global config ex treasury is set by us.

**Spearbit:** Acknowledged.

### 5.4.6 Withdrawal of dust shares is not supported

**Severity:** Informational

**Context:** EXManager.sol#L538-L540

**Description:** `withdrawOnceLive(...)` reverts if `shares < minimumWithdrawWhenLive`. While `minimumWithdrawWhenLive` is expected to be zero during normal operations, it may be set to a non-zero value in the presence of a blocked withdrawal queue and small withdrawal spam. In such cases, the protocol expects dust shares to be redeemed via secondary market exits. However, that may not always be feasible.

**Recommendation:** Consider if/when small withdrawals need to be prevented and if in-protocol mechanisms can be provided as an alternative.

**Kinetiq:** Acknowledged, but expect secondary market liquidity to be greater than dust.

**Spearbit:** Acknowledged.

### 5.4.7 `_exGhostLST` should be initialized with the LST of `exStakingManager`

**Severity:** Informational

**Context:** EXManager.sol#L112

**Description:** During `initialize()`, `_exGhostLST` is set to the parameter of `exGhostLST_`. Under the hood, this value should be equal to `exStakingManager.KHYPE` but this is never verified, which opens the unnecessary possibility for a mistake during deployment.

**Recommendation:** Consider adding the verification check that `exStakingManager.KHYPE == exGhostLST_`.

**Kinetiq:** Acknowledged, but given our instant withdraw update changes `exStakingManager.KHYPE` naming, gonna pass for now.

**Spearbit:** Acknowledged.

### 5.4.8 Missing Reentrancy Guards on State-Changing Functions

**Severity:** Informational

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

**Description:** Several functions across the codebase perform state updates, external calls, or token transfers without the use of a reentrancy guard. While some of these functions may not currently appear reentrant in their intended usage, the absence of explicit protection leaves the contracts more fragile and increases the risk that a future code change, integration, or unforeseen callback path could introduce a reentrancy vulnerability.

**Recommendation:** Add `nonReentrant` modifiers to functions that modify state, interact with external contracts, or transfer tokens, unless reentrancy is explicitly intended and safely handled.

**Kinetiq:** Fixed in PR 57.

**Spearbit:** Fix verified.