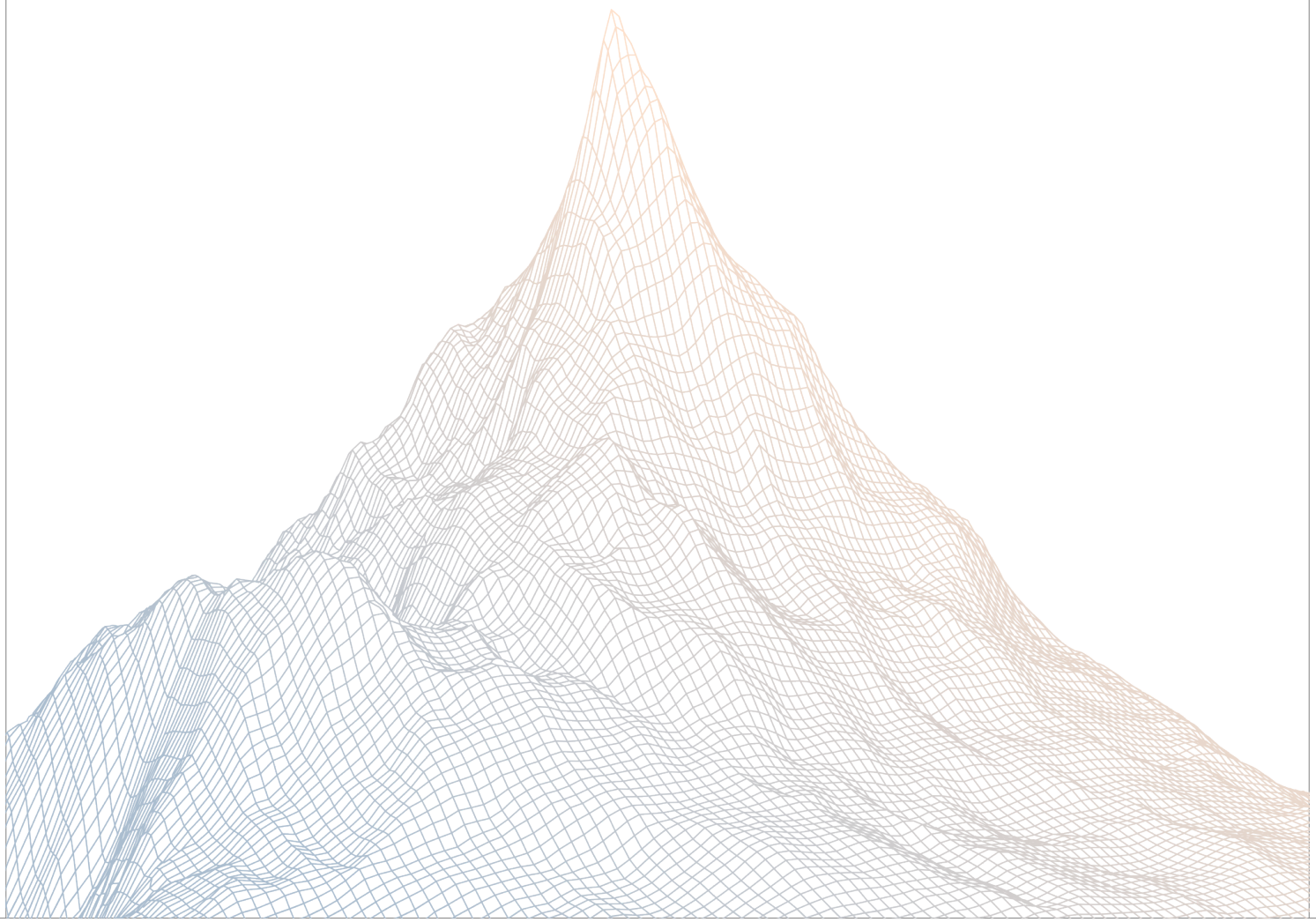
Zenith

Kinetiq

Smart Contract Security Assessment

VERSION 1.1



AUDIT DATES: November 13th to November 20th, 2025

AUDITED BY: adriro
ast3ros

Contents

1 Introduction

1.1 About Zenith

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

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

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

1.2 Disclaimer

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

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

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

1.3 Risk Classification

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

2

Executive Summary

2.1 About Kinetiq

Kinetiq is a liquid staking protocol built natively on Hyperliquid, enabling users to stake the native token of the Hyperliquid blockchain (HYPE) and receive Kinetiq Staked HYPE (kHYPE) in return.

kHYPE enables staking participation while retaining full liquidity and capital efficiency — earning staking rewards while staying active across DeFi.

2.2 Scope

The engagement involved a review of the following targets:

Target	launch
Repository	https://github.com/kinetiq-research/launch
Commit Hash	`c38b918dbe338ea69b7d8b905d72dcd11e389904`
Files	`src/*`

2.3 Audit Timeline

November 13, 2025	Audit start
November 20, 2025	Audit end
November 24, 2025	Report published

2.4 Issues Found

SEVERITY	COUNT
Critical Risk	0
High Risk	0
Medium Risk	2
Low Risk	13
Informational	8
Total Issues	**23**

3

Findings Summary

ID	Description	Status
M-1	Users cannot withdraw small amounts in WOUND_DOWN phase	Resolved
M-2	Withdrawals cannot be confirmed after slashing events due to strict amount validation	Acknowledged
L-1	Rounding dust accumulates in queue from per-user share conversions	Acknowledged
L-2	Unfair fee distribution via front-running	Acknowledged
L-3	queueWithdrawalAvailability returns zeros for WOUND_DOWN phase	Resolved
L-4	Excess HYPE not refunded in EXRouter deposits	Resolved
L-5	Requests canceled through the StakingManager would disrupt accounting	Acknowledged
L-6	Unbounded MAX_WHITELIST_TIERS constant	Resolved
L-7	Incorrect claimable amount in blockedWithdrawalInfo()	Resolved
L-8	Undefined attribute in blockedWithdrawalInfo()	Resolved
L-9	Missing LSTState initialization in BlockedWithdrawalQueue	Resolved
L-10	Recipient can manipulate amount received calculation	Resolved
L-11	Incorrect return value in _pay when recipient is address(this)	Resolved
L-12	Final staked amount can fall Below minHypeStake due to truncation	Resolved
L-13	Excess HYPE deposited and locked when bonding with HYPE	Resolved
I-1	No slippage protection for blocked withdrawals	Acknowledged

ID	Description	Status
I-2	Contract doesn't follow ERC-7201 standard for namespaced storage	Resolved
I-3	Unused internal function	Resolved
I-4	Wrong error message in phase validation	Resolved
I-5	Add storage gap to LSTState	Resolved
I-6	Misleading withdrawal delay in EXRouter	Resolved
I-7	Add setter for chunk size	Resolved
I-8	Unnecessary signature in updateWallet()	Acknowledged

4

Findings

4.1 Medium Risk

A total of 2 medium risk findings were identified.

[M-1] Users cannot withdraw small amounts in WOUND_DOWN phase

SEVERITY: Medium	IMPACT: Medium
STATUS: Resolved	LIKELIHOOD: Medium

Target

- EXManager.sol#L533

Description:

The `withdrawOnceLive` function incorrectly enforces the `minimumWithdrawWhenLive` check during the `WOUND_DOWN` phase, preventing users with small share balances from withdrawing their funds. This violates the intended design where all shares should be withdrawable after `windDown` is called: "If the manager calls windDown(), then all shares are available for withdrawal."

```
function withdrawOnceLive(uint256 shares, address recipient)
    external
    whenNotPaused
    returns (
        uint256 hypeAmount,
        uint256 hypeFee,
        uint256 withdrawalId,
        uint256 blockedShares,
        uint256 blockedWithdrawalId
    )
{
    ...
    // enforce minimum withdrawal amount when live
>>>     if (shares < minimumWithdrawWhenLive)
    revert Errors.WithdrawalLessThanMinimum();
    ...
}
```

Impact: Users with balances below `minimumWithdrawWhenLive` have their funds permanently locked during the WOUND_DOWN phase, unable to withdraw despite the manager's intent to allow full withdrawals.

Recommendations:

Exempt the WOUND_DOWN phase from the minimum withdrawal check:

```
if (exPhase == EXPhase.LIVE && shares < minimumWithdrawWhenLive) {
    revert Errors.WithdrawalLessThanMinimum();
}
```

Kinetiq: Resolved with `@29346bc2f8..`.

Zenith: Verified.

[M-2] Withdrawals cannot be confirmed after slashing events due to strict amount validation

SEVERITY: Medium	IMPACT: High
STATUS: Acknowledged	LIKELIHOOD: Low

Target

- LSTPayments.sol#L93-L96

Description:

From HIP-3:

```
In the event of malicious market operation, validators have the authority to
    slash the deployer's stake by conducting a stake-weighted vote. Even if
    the deployer has unstaked and initiated a staking withdrawal, the stake
    is still slashable during the 7-day unstaking queue.
```

Per HIP-3 specification, validators can slash a deployer's stake even during the 7-day unstaking period. However, the withdrawal mechanism locks the expected HYPE amount at queue time rather than settlement time, creating a critical mismatch when slashing occurs.

When user queues withdrawal (in both blocked and unblocked withdrawal queues), contract stores fixed `hypeAmount` based on current exchange rate. But if a slashing event happens during the delay window, the exchange rate worsens, but the stored `hypeAmount` doesn't change.

```
function queueWithdrawal(uint256 kHYPEAmount)
    external nonReentrant whenNotPaused whenWithdrawalNotPaused {
    ...
    // Third, create the WithdrawalRequest with the correct bufferUsed
    _withdrawalRequests[msg.sender][withdrawalId] = WithdrawalRequest({
>>>         hypeAmount: hypeAmount,
        kHYPEAmount: postFeeKHYPE,
        kHYPEFee: kHYPEFee,
        bufferUsed: bufferUsed,
        timestamp: block.timestamp
    });

    ...
```

```
}
```

Slashing event occurs during 7-day delay and the slash amount can be from 20% to 100%, it means L1 stake reduced and actual HYPE returned is less than queued amount. User attempts to confirm but transaction reverts due to strict equality checks:

- `StakingManager.confirmWithdrawal` requires sufficient balance:

```
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");

    stakingAccountant.recordClaim(amount);

    // Process withdrawal using call instead of transfer
    (bool success,) = payable(msg.sender).call{value: amount}("");
    require(success, "Transfer failed");
}
```

- `LSTPayments._confirmWithdrawal` reverts:

```
function _confirmWithdrawal(IStakingManager lstStakingManager, uint256 smid)
    internal returns (uint256 hypeAmount) {
    uint256 expectedWithdrawal
    = lstStakingManager.withdrawalRequests(address(this), smid).hypeAmount;
    uint256 balanceBefore = _reserves(HYPE);

>>>     lstStakingManager.confirmWithdrawal(smid);

    hypeAmount = _reserves(HYPE) - balanceBefore;
    if (hypeAmount ≠ expectedWithdrawal) revert Errors.InvalidAmount();
}
```

Impact: The withdrawals become unconfirmable and Users' funds are stuck after any slashing event unless new HYPE flows in. Fees are calculated on the stale queued amount rather than actual received amount.

Recommendations:

Proportionally adjust all pending withdrawals after slashing events.

Kinetiq: Acknowledged. Keeping as is given intended structure/flow of LST staking

manager coupled with HIP3 slashing risk requiring explicit bug manipulation in the HIP3 protocol implementation — which not worried about for our own markets launch.

4.2 Low Risk

A total of 13 low risk findings were identified.

[L-1] Rounding dust accumulates in queue from per-user share conversions

SEVERITY: Low	IMPACT: Low
STATUS: Acknowledged	LIKELIHOOD: Low

Target

- BlockedWithdrawalQueue.sol#L179-L180
- BlockedWithdrawalQueue.sol#L201-L202

Description:

In `processBlockedWithdrawals`, each user's HYPE amount is calculated individually via `_LSTToHYPE`, causing rounding down per conversion. However, the batch withdrawal from the staking manager converts the total shares in one operation.

This creates a discrepancy:

- Per-user sum: `sum(floor(userShares × rate))`
- Batch total: `floor(totalShares × rate)`

Since the batch total is always >= the sum of individual floors, the difference remains as unallocated HYPE in the queue contract after all users claim. This dust accumulates over time with each batch processed.

```
function processBlockedWithdrawals(uint256 items) external {
    ...
        // Calculate HYPE amount and fees for this portion
>>>         uint256 totalHypeAmount = _LSTToHYPE(exStakingAccountant,
    sharesToQueue);
>>>         (uint256 hypeAmount, uint256 fee) = _applyFee(totalHypeAmount,
    feeRate);

    ...
```

```
    // Queue the batch with the staking manager if we have shares to process
    if (batchSize > 0) {
>>>         (uint256 actualSmid, uint256 totalHypeAmount) =
            _processQueueWithdrawal(ghostLST, exStakingManager, batchSize);

        // Verify SMID matches expected value
        if (smid != actualSmid) revert Errors.InvalidSMID();

        uint256 hypeAmount = totalHypeAmount - batchFees;

        // @dev safe as uint248 is much larger than the total hype supply, so
    no overflow possible.
        queuedBatch[smid] = BatchInfo({fee: uint248(batchFees), status:
    BatchStatus.PENDING});
        emit BatchQueued(smid, batchSize, hypeAmount, batchFees);
    }
}
```

Recommendations:

Accumulate dust into the fee for that batch.

Kinetiq: Acknowledged, but in practice accumulated dust will be very small so keeping as is.

[L-2] Unfair fee distribution via front-running

SEVERITY: Low	IMPACT: Medium
STATUS: Acknowledged	LIKELIHOOD: Low

Target

- EXManager.sol#L363-L367

Description:

The `stakeFees` function stakes trading fees from the HIP3 market into the staking manager, immediately increasing the value of all existing exLST shares proportionally. An attacker can exploit this by:

- Target pools with staked HYPE > minimum requirements
- Front-running the keeper bot's `stakeFees` transaction with a large deposit to mint exLST shares at the current exchange rate
- Benefiting from the instant value increase when fees are staked
- Back-running with a withdrawal request to capture the profit

This attack dilutes rewards intended for long-term stakers, allowing attackers to extract value without bearing risks over time. The attack is particularly profitable when deposit sizes are large relative to the pool and when substantial fees are being distributed.

```
function stakeFees() external payable onlyPhase(EXPhase.LIVE)
    whenNotPaused {
    uint256 hypeAmount = msg.value;
    _stake(_exGhostLST, exStakingManager, hypeAmount);
    emit FeesStaked(hypeAmount);
}
```

Recommendations:

- Distribute fees over a period of time OR
- Only stake fees in small amount so the profit is less than withdrawal fees.

Kinetiq: Acknowledged.This is a good point to ensure fee distributions are regular.

Although worth noting that exchanges that are successful or promising will likely be at the

maximum supply cap of the EXLST, and then dilution would not be possible.

In the version where we plan to allow permissionless deployments, fee compounding will be fully on-chain and automated.

The setup we will deploy with this version of contracts is planned to be 888,888 maximum EXLST shares for the 500k minimum hype.

[L-3] queueWithdrawalAvailability returns zeros for WOUND_DOWN phase

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

Target

- EXRouter.sol#L220-L247

Description:

The `queueWithdrawalAvailability` function does not handle the `WOUND_DOWN` phase, causing it to return all zeros. This misrepresents the actual withdrawal capacity, as all shares become available for withdrawal in this phase per the design documentation.

```
function queueWithdrawalAvailability(IEXManager exManager, address token)
    external
    view
    returns (
        uint256 instantWithdrawalCapacity, //  Denominated in EXLST shares.
        uint256 processableWithdrawalCapacity, //  Denominated in EXLST
    shares.
        uint256 currentBlockedWithdrawals, // Denominated in GHOST LST
    shares.
        uint256 delay, //Withdrawal delay for processable withdrawals.
        uint256 minWithdrawalAmount
    )
{
    IEXManager.EXPhase exPhase = phase(exManager);
    if (exPhase == IEXManager.EXPhase.FUNDING) {
        // ... handles FUNDING
    } else if (exPhase == IEXManager.EXPhase.LIVE) {
        if (token ≠ HYPE) revert Errors.InvalidToken();
        // ... handles LIVE
    }
    // WOUND_DOWN case missing - returns zeros
}
```

Recommendations:

Add handling for the `WOUND_DOWN` phase to return accurate withdrawal capacity and delay information

Kinetiq: Resolved with `@c772d6f733..`.

Zenith: Verified.

[L-4] Excess HYPE not refunded in EXRouter deposits

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

Target

- EXRouter.sol#L53

Description:

When users deposit HYPE through `EXRouter.deposit`, any excess `msg.value` beyond the specified `amountIn` becomes permanently stuck in the router contract.

- The `_pay` function only validates that `msg.value ⩾ amountIn`.
- The router forwards exactly `amountIn` to the `EXManager`.
- Any excess (`msg.value - amountIn`) remains in the router with no recovery mechanism.

```
function deposit(
    IEXManager exManager,
    address tokenIn,
    uint256 amountIn,
    address recipient,
    IEIP712Verifier.EIP712SignedData memory whitelistUserSignedData
) public payable returns (uint256 sharesOut) {
    // pay token in to router and increase allowance for ex manager to pull
    if not HYPE
>>>     _pay(tokenIn, msg.sender, address(this), amountIn);
    if (tokenIn ≠ HYPE) {
        IERC20(tokenIn).safeIncreaseAllowance(address(exManager), amountIn);
    }

    // msg.value to send in to ex manager if HYPE should be amountIn
    specified
    IEXManager.EXPhase exPhase = phase(exManager);
>>>     uint256 value = (tokenIn == HYPE) ? amountIn : 0;

    ...
}
```

While this requires user error, the EXRouter has no withdrawal function to recover stuck funds, making the loss permanent.

Recommendations:

Add a check to ensure exact payment or refund excess amounts.

Kinetiq: Resolved with `@6aeeaab64e..`.

Zenith: Verified.

[L-5] Requests canceled through the StakingManager would disrupt accounting

SEVERITY: Low	IMPACT: Medium
STATUS: Acknowledged	LIKELIHOOD: Low

Target

- EXManager.sol
- BlockedWithdrawalQueue.sol

Description:

Once requested, withdrawals in the StakingManager contract can be canceled by the admin role. When this happens, the request is deleted and the LST tokens are returned to the user.

These cancellations would conflict with the implementation of EXManager and BlockedWithdrawalQueue, as these contracts track requests expecting these to be eventually confirmed, and without having a way to deal with cancellations and refunded assets.

Recommendations:

Ensure not to cancel requests through the StakingManager, or implement additional support to handle these exceptional scenarios.

Kinetiq: Acknowledged. Will keep as is, and ensure not to cancel requests through staking manager.

[L-6] Unbounded `MAX_WHITELIST_TIERS` constant

`SEVERITY:` Low	`IMPACT:` Low
`STATUS:` Resolved	`LIKELIHOOD:` Low

Target

- Constants.sol

Description:

The `MAX_WHITELIST_TIERS` constant is intended to act as a maximum limit on the size of the whitelist tiers. However, its value of `type(uint128).max` provides no practical bound on the number of tiers.

Recommendations:

Use an appropriate value for the constant, or remove the check if not needed.

Kinetiq: Resolved with `@d71e9de610..`.

Zenith: Verified. The constant and the check have been removed.

[L-7] Incorrect claimable amount in `blockedWithdrawalInfo()`

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

Target

- EXRouter.sol

Description:

The implementation of `_extractBlockedWithdrawalInfo()` fills the `claimableAmount` attribute by aggregating the total HYPE pending in withdrawal requests to the StakingManager.

```
375:             IStakingManager.WithdrawalRequest memory wr =
376:                 exStakingMan-
    ager.withdrawalRequests(address(blockedWithdrawalQueue), qb.smid);
377:             // Check batch fee status
378:             uint256 deadline = wr.timestamp + delay;
379:             if (
380:                 batchStatus ==
    IBlockedWithdrawalQueue.BatchStatus.CONFIRMED
381:                     || _isWithdrawable(exStakingManager, deadline,
    wr.hypeAmount)
382:             ) {
383:                 //Batch confirmed, can claim
384:                 blockedWithdrawal.claimableBatchIndices[claimableIndex]
    = j;
385:                 blockedWithdrawal.claimableAmount += wr.hypeAmount;
386:                 claimableIndex++;
387:             } else {
```

This would include HYPE from all batches present in the request, not necessarily scoped to the user.

Recommendations:

Use the amount from the batch that corresponds to the user in context.

```
//Batch confirmed, can claim
blockedWithdrawal.claimableBatchIndices[claimableIndex] = j;
blockedWithdrawal.claimableAmount += wr.hypeAmount;
blockedWithdrawal.claimableAmount += qb.hypeOwed;
claimableIndex++;
```

Kinetiq: Resolved with @c482300397.. .

Zenith: Verified.

[L-8] Undefined attribute in `blockedWithdrawalInfo()`

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

Target

- EXRouter.sol

Description:

The list returned by `blockedWithdrawalInfo()` is missing the `blockedWithdrawalId` attribute, as this value is never filled by the implementation of `_extractBlockedWithdrawalInfo()`.

Recommendations:

Assign the `blockedWithdrawalId` attribute with the value of `withdrawalIndex`.

Kinetiq: Resolved with `@983e93b6c6...`.

Zenith: Verified.

[L-9] Missing LSTState initialization in BlockedWithdrawalQueue

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

Target

- BlockedWithdrawalQueue.sol

Description:

The BlockedWithdrawalQueue contract inherits from LSTState through LSTPayments, but it never initializes the base contract state, leading to undefined references.

Recommendations:

Call `__LSTState_init()` in the initializer of BlockedWithdrawalQueue.

Kinetiq: Resolved with `@26fac1e47f..`.

Zenith: Verified.

[L-10] Recipient can manipulate amount received calculation

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

Target

- LSTPayments.sol#L158-L187

Description:

The `_pay` function calculates `amountReceived` by comparing balances before and after transfer. In `tokenReceived = HYPE` and `payer == address(this)`, when the `recipient` is a contract, the `payable(recipient).call{value: amountPaid}("")` grants control flow to the `recipient`, allowing it to manipulate its own balance during the callback.

A malicious recipient could artificially inflate `amountReceived` by receiving additional tokens during the callback, or deflate it by transferring tokens away, potentially breaking accounting assumptions in the calling contract.

```
function _pay(address tokenReceived, address payer, address recipient,
    uint256 amountPaid)
    internal
    returns (uint256 amountReceived)
{
    if (amountPaid == 0) return 0;
    uint256 balanceBefore = _balance(tokenReceived, recipient);
    ...
            if (address(this).balance < amountPaid) {
                revert Errors.InsufficientAmount(address(this).balance,
    amountPaid);
            }
>>>             (bool success,) = payable(recipient).call{value:
    amountPaid}("");
            require(success, "Native transfer failed");
        ...
    }
>>>     amountReceived = _balance(tokenReceived, recipient) - balanceBefore;
}
```

Recommendations:

Return `amountPaid` directly for native HYPE paths instead of recalculating from balance differences.

Kinetiq: Resolved with `@7ebba852f3..`.

Zenith: Verified.

[L-11] Incorrect return value in _pay when recipient is address(this)

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

Target

- LSTPayments.sol#L158-L187

Description:

The `_pay` function incorrectly returns `0` for `amountReceived` when the recipient is `address(this)` and native HYPE is being transferred.

The root cause is that `balanceBefore` captures the contract's balance after `msg.value` has already been added (since `msg.value` is credited before function execution). When calculating `amountReceived = _balance(tokenReceived, recipient) - balanceBefore`, the result is `0` because no additional balance change occurs.

```
function _pay(address tokenReceived, address payer, address recipient,
    uint256 amountPaid)
    internal
    returns (uint256 amountReceived)
{
    if (amountPaid == 0) return 0;
    uint256 balanceBefore = _balance(tokenReceived, recipient);
    if (tokenReceived == HYPE) {
        ...
>>>         } else if (recipient == address(this) && msg.value < amountPaid) {
            revert Errors.InsufficientAmount(msg.value, amountPaid);
        }
    ...
    amountReceived = _balance(tokenReceived, recipient) - balanceBefore;
}
```

Recommendations:

When `recipient == address(this)` and native HYPE is paid, return `msg.value` directly

Kinetiq: Resolved with `@754985ace0..`.

Zenith: Verified.

[L-12] Final staked amount can fall Below minHypeStake due to truncation

SEVERITY: Low	IMPACT: Medium
STATUS: Resolved	LIKELIHOOD: Low

Target

- EXManager.sol#L275
- EXManager.sol#L300

Description:

The `fund` function enforces that the HYPE amount meets the minimum stake requirement:

```
function fund()
    external
    onlyRole(OPERATOR_ROLE)
    onlyPhase(EXPhase.FUNDING)
    whenNotPaused
    returns (uint256 hypeAmount)
{
    ...
>>>     if (hypeAmount < globalConfig.minHypeStake()) {
        revert Errors.InvalidAmount();
    }

    exPhase = EXPhase.LAUNCHING;
}
```

However, `launch` truncates the stake amount to make it divisible by 1e10, removing any remainder as "dust":

```
function launch(EIP712SignedData memory walletSignedData)
    external
    onlyRole(OPERATOR_ROLE)
    onlyPhase(EXPhase.LAUNCHING)
    whenNotPaused
    returns (address wallet, uint256 hypeAmount, uint256 hypeDust)
{
```

```
    ...
    hypeAmount = _reserves(HYPE);

    // check hypeAmount divisible by 1e10, sweep remainder out to operator
    after staking
    hypeDust = hypeAmount % 1e10;
>>>     hypeAmount -= hypeDust;

    ...
}
```

If `minHypeStake` is not a multiple of 1e10, the final staked amount will be up to 1e10-1 wei below `minHypeStake`, violating the variant: The hype staked in the `HIP3StakingManager` must stay above the `minHypeStake` set in the global config during the LIVE phase.

Example: If `minHypeStake = 1e18 + 5e9`, the check passes in `fund`, but `launch` stakes only `1e18`, which is below the configured minimum.

Recommendations:

Enforce in `GlobalConfig` that `minHypeStake` must be a multiple of 1e10.

Kinetiq: Resolved with `@60af7510af..`

Zenith: Verified.

[L-13] Excess HYPE deposited and locked when bonding with HYPE

SEVERITY: Low	IMPACT: Medium
STATUS: Resolved	LIKELIHOOD: Low

Target

- EXManager.sol#L256
- EXManager.sol#L388

Description:

The `bond` function shows inconsistent behavior depending on whether HYPE or KHYPE is used for bonding:

- KHYPE path: Only the `opBond` amount is deposited and locked
- HYPE path: The entire `msg.value` is deposited and locked, regardless of the `opBond` amount

When `bond` calls `depositWhileFunding(opBond, ...)`, it passes `opBond` as the `amountPaid` parameter. However, if `msg.value > 0`, the function ignores this parameter and uses `msg.value` instead:

```
function bond()
    external
    payable
    onlyRole(OPERATOR_ROLE)
    onlyPhase(EXPhase.UNBONDED)
    whenNotPaused
    returns (uint256 shares)
{
    // flip to funding before deposit opBond to avoid revert on onlyPhase
    modifier
    exPhase = EXPhase.FUNDING;
    // deposit op bond
>>>     shares = depositWhileFunding(opBond, address(this),
    EIP712SignedData({data: bytes(""), signature: bytes("")}));
    // 1:1 ratio between shares and kHYPE added for first commit
    if (shares < opBond) revert Errors.InvalidShares();
    emit Bonded(msg.sender, shares);
```

```
}
```

```
function depositWhileFunding(uint256 amountPaid, address recipient,
    EIP712SignedData memory whitelistUserSignedData)
    public
    payable
    onlyPhase(EXPhase.FUNDING)
    whenNotPaused
    returns (uint256 shares)
{
    ...
    address tokenPaid;
    uint256 kHYPEAmount;
    if (msg.value > 0) {
>>>         tokenPaid = HYPE;
        amountPaid = msg.value;
        kHYPEAmount = _stake(KHYPE, KHYPE_STAKING_MANAGER, amountPaid);
    } else {
        tokenPaid = KHYPE;
        kHYPEAmount = _pay(KHYPE, msg.sender, address(this), amountPaid);
    }

    ...
}
```

If an operator sends `msg.value > opBond`, excess funds are permanently locked in the contract beyond the required bond amount.

Recommendations:

Consider:

- Require exact value on HYPE path OR
- Refund excess amounts to the operator OR
- Refund locked shares to the operator in WOUND_DOWN phase OR
- Mint Excess Shares to the Operator

Kinetiq: Resolved with `@885907bc7a..`.

Zenith: Verified.

4.3 Informational

A total of 8 informational findings were identified.

[I-1] No slippage protection for blocked withdrawals

SEVERITY: Informational	IMPACT: Informational
STATUS: Acknowledged	LIKELIHOOD: Low

Target

- BlockedWithdrawalQueue.sol#L157

Description:

When users queue a blocked withdrawal, they lock in the amount of ghost LST shares but not the final HYPE output amount or the fee rate. Since blocked withdrawals may remain in the queue for extended periods before processing, users face two sources of unexpected value loss:

- If governance increases `unstakeFeeRate` while a withdrawal is pending, users will pay higher fees than when they initiated the withdrawal.
- Slashing events will reduce the output amount.

Users have no mechanism to specify minimum acceptable output amounts or to cancel withdrawals if conditions become unfavorable.

```
function processBlockedWithdrawals(uint256 items) external {
    // Calculate available shares
    uint256 withdrawableShares = _withdrawableShares();

    // Early return if nothing to process
    if (withdrawableShares == 0) {
        return;
    }
    ...
    uint256 n = items == 0 ? _blockedWithdrawalCtr + 1 : Math.min(i + items,
    _blockedWithdrawalCtr + 1);
>>>     uint96 feeRate = uint96(globalConfig.unstakeFeeRate());
    ...
```

```
}
```

Recommendations:

Allow users to specify a minimum HYPE amount when queueing.

Kinetiq: Acknowledged, but keeping as is.

[I-2] Contract doesn't follow ERC-7201 standard for namespaced storage

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

Target

- HIP3StakingManager.sol#L14

Description:

The `HIP3StakingManager` uses a custom storage slot but deviates from the ERC-7201 standard in three ways:

- The formula omits `& ~bytes32(uint256(0xff))` required by ERC-7201
- Storage should be defined as a struct type rather than individual variables
- Lacks the `@custom:storage-location` tag

```
contract HIP3StakingManager is StakingManager, IHIP3StakingManager {
    // Storage
    bytes32 private constant EXMANAGER_SLOT
    = bytes32(uint256(keccak256("kinetiq.hip3.exManager")) - 1);
```

Recommendations:

Implement ERC-7201 compliant storage.

Kinetiq: Resolved with `@3540d0112f..`.

Zenith: Verified.

[I-3] Unused internal function

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

Target

- BlockedWithdrawalQueue.sol#L304-L308

Description:

The internal function `_totalGhostReserves` is never called within the contract or any derived contracts, making it dead code that unnecessarily increases deployment costs.

```
function _totalGhostReserves()
    internal view returns (uint256 totalGhostReserves) {
    totalGhostReserves = IERC20(ghostLST).balanceOf(address(exManager))
    + IERC20(ghostLST).balanceOf(address(this));
}
```

Recommendations:

Remove the unused function.

Kinetiq: Resolved with `@508749efdce..`.

Zenith: Verified.

[I-4] Wrong error message in phase validation

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

Target

- EXRouter.sol#L78
- EXRouter.sol#L107

Description:

The `deposit` and `withdraw` functions always revert with FUNDING as the required phase, regardless of which phases are actually valid for the operation. This provides incorrect feedback to users about which phase transitions are needed.

```
error InvalidPhase(uint256 got, uint256 needed);

function deposit(
    IEXManager exManager,
    address tokenIn,
    uint256 amountIn,
    address recipient,
    IEIP712Verifier.EIP712SignedData memory whitelistUserSignedData
) public payable returns (uint256 sharesOut) {
    ...
    } else {
>>>         revert Errors.InvalidPhase(uint256(exPhase),
    uint256(IEXManager.EXPhase.FUNDING));
    }
}

function withdraw(IEXManager exManager, address tokenOut, uint256 sharesIn,
    address recipient)
    public
    returns (
        uint256 amountOut,
        uint256 withdrawalFee,
        uint256 withdrawalId,
        uint256 blockedShares,
        uint256 blockedWithdrawalId
```

```
    )
{
    ...
    } else {
>>>         revert Errors.InvalidPhase(uint256(exPhase),
    uint256(IEXManager.EXPhase.FUNDING));
    }
}
```

Recommendations:

Pass a meaningful phase value representing the valid options (e.g., the earliest valid phase).

Kinetiq: Resolved with `@7d04a3fe92..`.

Zenith: Verified.

[I-5] Add storage gap to LSTState

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

Target

- LSTState.sol

Description:

Given that the LSTState contract acts as a base contract in an upgradeable hierarchy, consider introducing a storage gap to allow for future state variable additions without causing storage collisions in derived contracts.

Recommendations:

Add a storage gap (e.g., `uint256[50] private __gap`) in the LSTState contract.

Kinetiq: Resolved with `@c10868e35f..`.

Zenith: Verified.

[I-6] Misleading withdrawal delay in EXRouter

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

Target

- EXRouter.sol

Description:

The implementation of `withdrawalDelay()` returns the EX StakingManager processing delay during the LIVE or WOUND_DOWN phases.

However, this is still subject to available shares and the eventual block queue processing, which may delay withdrawals beyond the specified delay during the LIVE phase.

Recommendations:

Document these assumptions and inform of a potential unbounded delay in the NatSpec of `withdrawalDelay()`.

Kinetiq: Resolved with `@ce781ce8f7..`.

Zenith: Verified.

[I-7] Add setter for chunk size

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

Target

- BlockedWithdrawalQueue.sol

Description:

The `chunkSize` configuration in BlockedWithdrawalQueue cannot be modified once initialized.

Recommendations:

Consider adding a setter to modify the chunk size if needed.

Kinetiq: Resolved with `@123ed82b68..`.

Zenith: Verified.

[I-8] Unnecessary signature in `updateWallet()`

SEVERITY: Informational	IMPACT: Informational
STATUS: Acknowledged	LIKELIHOOD: Low

Target

- EXManager.sol

Description:

The `updateWallet()` function requires its payload to be signed by the `exWalletAdmin`, but also requires the caller to be the `exWalletAdmin`. Since the caller is the same authority, this signature can be skipped.

Recommendations:

The implementation of `updateWallet()` can be simplified to remove the signature check and directly apply the configuration changes.

Kinetiq: Acknowledged, but will keep as is to use internal `_updateWallet()` in current form.