Kinetiq: PR 69

Security Review

Cantina Solo review by:
Optimum, Lead Security Researcher

June 11, 2026

Contents

1 Introduction **2**
- 1.1 About Cantina . . . 2
- 1.2 Disclaimer . . . 2
- 1.3 Risk assessment . . . 2
  - 1.3.1 Severity Classification . . . 2

2 Security Review Summary **3**
- 2.1 Scope . . . 3

3 Findings **4**
- 3.1 High Risk . . . 4
  - 3.1.1 `LaunchFeeSplitter.split()` Sends 100x Less Fees Than Intended Due to Decimal Mismatch . . . 4
- 3.2 Medium Risk . . . 5
  - 3.2.1 Address Collision and Front-running Risk in Predetermined Gate Initializations . . . 5
  - 3.2.2 `LaunchFeeSplitter.split()` May Silently Fail if Recipient Accounts Are Not Activated on HyperCore . . . 5
- 3.3 Low Risk . . . 6
  - 3.3.1 Cumulative Lock Requirements Encourage Sybil Exploitation . . . 6
- 3.4 Gas Optimization . . . 8
  - 3.4.1 Gas Optimization via `msg.sender` in `MarketBonded` Event . . . 8
  - 3.4.2 Redundant Storage of `exManager` Contract Address . . . 8
  - 3.4.3 Gas Optimization via Ordered Array Validation for Deduplication . . . 8
- 3.5 Informational . . . 9
  - 3.5.1 Inconsistent Storage Layouts in Upgradeable Contracts . . . 9
  - 3.5.2 Ambiguous Variable Naming for Tier Identifiers in `queueTierUpgrade` . . . 9
  - 3.5.3 Inaccurate Role Reference in `forceUnwindPhase` Comment . . . 10
  - 3.5.4 Lost Context in `Unwound` Event Due to Early State Reset . . . 10
  - 3.5.5 Hardcoded Token Decimal Equivalence Across Asset Tracking Configurations . . . 10
  - 3.5.6 Missing Logical Separation of LST and Launch Proxies in `deployMarketContracts` . . . 11
  - 3.5.7 Redundant and Dead Code Execution in `bond()` . . . 11
  - 3.5.8 `EXRouter.queueWithdrawalAvailability()`: Unassigned Return Variable `instantWithdrawalCapacity` . . . 12
  - 3.5.9 Missing Fields in Withdrawal Events . . . 12
  - 3.5.10 Inconsistent Handling of Already Confirmed Withdrawals . . . 12

1 Introduction

1.1 About Cantina

Cantina is a security services marketplace that connects top security researchers and solutions with clients. Learn more at cantina.xyz

1.2 Disclaimer

A security review is a detailed evaluation of the security posture of the code at a particular moment based on the information available at the time of the review. While the review endeavors to identify and disclose all potential security issues, it cannot guarantee that every vulnerability will be detected or that the code will be entirely secure against all possible attacks. The assessment is conducted based on the specific commit and version of the code provided. Any subsequent modifications to the code may introduce new vulnerabilities that were absent during the initial review. Therefore, any changes made to the code require a new security review to ensure that the code remains secure. Please be advised that a security review is not a replacement for continuous security measures such as penetration testing, vulnerability scanning, and regular code reviews.

1.3 Risk assessment

Severity level	Impact: High	Impact: Medium	Impact: Low
Likelihood: high	Critical	High	Medium
Likelihood: medium	High	Medium	Low
Likelihood: low	Medium	Low	Low

1.3.1 Severity Classification

The severity of security issues found during the security review is categorized based on the above table. Critical findings have a high likelihood of being exploited and must be addressed immediately. High findings are almost certain to occur, easy to perform, or not easy but highly incentivized thus must be fixed as soon as possible.

Medium findings are conditionally possible or incentivized but are still relatively likely to occur and should be addressed. Low findings are a rare combination of circumstances to exploit, or offer little to no incentive to exploit but are recommended to be addressed.

Lastly, some findings might represent objective improvements that should be addressed but do not impact the project's overall security (Gas and Informational findings).

2 Security Review Summary

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

From May 18th to May 26th the security researchers conducted a review of launch on commit hash 6b6896e6. A total of **17** issues were identified:

Issues Found

Severity	Count	Fixed	Acknowledged
Critical Risk	0	0	0
High Risk	1	1	0
Medium Risk	2	2	0
Low Risk	1	0	1
Gas Optimizations	3	0	0
Informational	10	0	0
Total	**17**	**3**	**1**

2.1 Scope

The security review had the following components in scope for launch on commit hash 6b6896e6:

```
src
├── BlockedWithdrawalQueue.sol
├── EXDeployer.sol
├── EXFactory.sol
├── EXLST.sol
├── EXManager.sol
├── EXRouter.sol
├── GlobalConfig.sol
├── ProtocolRolesController.sol
├── base
│   ├── HyperCoreActivatable.sol
│   ├── HyperCoreAsset.sol
│   └── LSTPayments.sol
├── beacon
│   └── UpgradeableBeaconRegistry.sol
├── facets
│   ├── HIP3ConfigFacet.sol
│   └── SlashingAwareWithdrawalFacet.sol
├── fee-distribution
│   ├── LaunchFeeSplitter.sol
│   └── StakeFeesThrottle.sol
├── gate
│   ├── CompositeGate.sol
│   ├── MultiplexerGate.sol
│   ├── TieredMintGate.sol
│   └── WhitelistGate.sol
└── lib
    ├── Constants.sol
    ├── Errors.sol
    └── HIP3L1Write.sol
```

3 Findings

3.1 High Risk

3.1.1 `LaunchFeeSplitter.split()` Sends 100x Less Fees Than Intended Due to Decimal Mismatch

Severity: High Risk

Context: LaunchFeeSplitter.sol#L160

Description:

> **Note:** This issue was discovered by the client during the first day of the review.

`LaunchFeeSplitter::split()` distributes fees to `protocolFeeTreasury`, `deployerTreasury`, and `buybackWallet` via CoreWriter `sendAsset` actions. If any recipient address is not yet activated on HyperCore, the first transfer to that address requires a 1 quote token activation fee deducted from the splitter's own HyperCore spot balance.

If the splitter's spot balance is insufficient to cover activation fees for all unactivated recipients, the CoreWriter actions silently fail — the EVM transaction succeeds and emits no error, but the HyperCore-side transfers never occur and fees are not delivered.

`EXFactory::activateMarket` is expected to fund the splitter's own activation, but does not activate `deployerTreasury` or `buybackWallet`, which are deployer-supplied addresses of unknown activation status.

Recommendation:

1. **Require `deployerTreasury` to be activated at deploy time**: In `EXFactory::deployMarket`, add a check that `globalConfig.l1Read().coreUserExists(params.deployerTreasury) == true` and revert if not. This forces the deployer (`msg.sender`) to ensure their treasury is activated on HyperCore before the market can be deployed, without requiring the protocol to manage activation on their behalf.
2. **Guard configurable recipient setters**: Add `coreUserExists` activation checks in `GlobalConfig` wherever `protocolFeeTreasury` can be updated, and in `LaunchFeeSplitter` in `setBuybackWallet`. Since these addresses are configurable post-deployment, ensuring they are activated at the point of configuration prevents silent failures if they are ever changed to unactivated addresses. `viewSplits()` reads the contract's margin balance from the HIP-3 perp DEX via `accountMarginSummary().rawUsd`:

```
int64 bal = globalConfig.l1Read().accountMarginSummary(_asyncConfig.sourceDex,
↪  address(this)).rawUsd;
uint256 amount = uint256(uint64(bal));
```

`rawUsd` reports the margin balance in **6 decimals** (USDC native precision). This `amount` is then passed directly to `_send()` which forwards it as the `_wei` parameter to CoreWriter's `sendAsset`:

```
_wei: uint64(amount)
```

CoreWriter expects amounts in **HyperCore wei decimals (8 decimals)** for tokens such as USDC. Since no scaling is applied between the read and the send, every fee distribution transfers **100x less than the actual accumulated balance**.

Recommendation: Store the quote token's HyperCore `weiDecimals` in `_asyncConfig` alongside `feeToken`, and normalize `rawUsd` to the quote token's HyperCore `weiDecimals` before calling `sendAsset`. `feeTokenWeiDecimals` should be set in both `initializeAsyncConfig` and `setFeeToken` to ensure it stays in sync with the configured fee token.

Kinetiq: Fixed in commit a98810.

Cantina Managed: Fixed by injecting the adjusted `amount` as a parameter. Note that the OPERATOR role is in charge of adjusting the decimals correctly.

3.2 Medium Risk

3.2.1 Address Collision and Front-running Risk in Predetermined Gate Initializations

Severity: Medium Risk

Context: EXFactory.sol#L574

Description: The gate contracts are deployed independently outside of `EXFactory.deployMarket()`. This structural decoupling is likely driven by two core design requirements:

1. **Flexibility:** It allows markets to choose or customize different gate contracts (e.g., `WhitelistGate`, `TieredMintGate`, `CompositeGate`) depending on their specific access control needs.
2. **Circular Dependency:** A mutual dependency exists between the two components — the gate contract must explicitly store and track its parent `exManager` or `authorizedCaller`, while the `ExManager` must simultaneously store the address of its assigned gate.

To resolve this, the deployment architecture relies on pre-deploying the gate contracts configured with a pre-computed address for `ExManager`.

However, the current proxy deployment method is highly vulnerable to disruption. `EXDeployer._createProxy()` uses the standard `new` keyword rather than deterministically deploying proxies via `CREATE2`. Standard deployments rely strictly on the deployer contract's sequential address nonce. Because `deployMarket()` in `EXFactory` is a public function, any user or attacker can execute the transaction. An attacker can intentionally front-run or clear unrelated market deployments, forcing the `EXDeployer` nonce forward and altering the resulting address sequence.

Impact: As a result of address drift, the gate contracts will be initialized with and store the wrong address for `exManager` and/or `authorizedCaller`. This address mismatch will cause crucial permissioned hooks and validations inside the gate system to fail, leading to persistent reverts on vital market functions and causing at least a temporary denial of service (DoS) for the protocol.

Recommendation: Refactor the deployment logic in `EXDeployer` to utilize `CREATE2` when generating the proxy instances for `ExManager` and its companion market infrastructure. The `salt` for `CREATE2` should be constructed from the market id (`keccak256(abi.encode(msg.sender, nonce))`) combined with the `contractType`. This ensures that the `ExManager` address remains perfectly deterministic and independent of execution order or standard nonce mutations, neutralizing front-running risks and deployment drift.

Kinetiq: Fixed in commit a98810.

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

3.2.2 `LaunchFeeSplitter.split()` May Silently Fail if Recipient Accounts Are Not Activated on HyperCore

Severity: Medium Risk

Context: LaunchFeeSplitter.sol#L149

Description:

> **Note:** This issue was discovered by the client during the first day of the review.

`LaunchFeeSplitter::split()` distributes fees to `protocolFeeTreasury`, `deployerTreasury`, and `buybackWallet` via CoreWriter `sendAsset` actions. If any recipient address is not yet activated on HyperCore, the first transfer to that address requires a 1 quote token activation fee deducted from the splitter's own HyperCore spot balance.

If the splitter's spot balance is insufficient to cover activation fees for all unactivated recipients, the CoreWriter actions silently fail — the EVM transaction succeeds and emits no error, but the HyperCore-side transfers never occur and fees are not delivered.

`EXFactory::activateMarket` is expected to fund the splitter's own activation, but does not activate `deployerTreasury` or `buybackWallet`, which are deployer-supplied addresses of unknown activation status.

Recommendation:

1. **Require `deployerTreasury` to be activated at deploy time**: In `EXFactory::deployMarket`, add a check that `globalConfig.l1Read().coreUserExists(params.deployerTreasury) == true` and revert if not. This forces the deployer (`msg.sender`) to ensure their treasury is activated on HyperCore before the market can be deployed, without requiring the protocol to manage activation on their behalf.
2. **Guard configurable recipient setters**: Add `coreUserExists` activation checks in `GlobalConfig` wherever `protocolFeeTreasury` can be updated, and in `LaunchFeeSplitter` in `setBuybackWallet`. Since these addresses are configurable post-deployment, ensuring they are activated at the point of configuration prevents silent failures if they are ever changed to unactivated addresses.

Kinetiq: Fixed in commit a98810.

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

3.3 Low Risk

3.3.1 Cumulative Lock Requirements Encourage Sybil Exploitation

Severity: Low Risk

Context: TieredMintGate.sol#L193

Description: The `TieredMintGate` contract calculates a user's minting allocation based on the total cumulative amount of `lockTokens` currently held in their address. Because thresholds are evaluated cumulatively per account rather than on a per-transaction basis, a user attempting to execute multiple sequential mints is forced to deposit exponentially more lock collateral to climb into higher tier brackets.

An attacker can easily bypass this capital penalty by splitting their funds across multiple unique addresses (a Sybil attack). Instead of locking a massive cumulative sum on a single account to increase an allowance, the user can deploy multiple wallets to repeatedly exploit the lower, highly capital-efficient tiers. Because the minted shares are standard transferable tokens, they can be consolidated back into a single address post-mint, fully circumventing the intended economic constraints.

Concrete Examples:

Example 1: Single Tier Barrier: Assume the contract is configured with only one active tier configuration:

- **Tier 1:** Lock `1,000` tokens -> Cumulative Mint Allowance: `10,000` shares.
- **Scenario A: Honest User (Single Address)**.
  1. Alice locks `1,000` tokens using her primary address and successfully mints `10,000` shares.
  2. Later, Alice decides she wants to mint an additional `10,000` shares.
  3. Because her single address has already completely exhausted its maximum Tier 1 allocation allowance, her transaction is blocked unless she can access a higher tier or deposit exponentially more lock collateral.
- **Scenario B: Sybil Attacker (Multiple Addresses)**.
  1. Bob locks `1,000` tokens using *Address 1* and mints `10,000` shares.
  2. To mint another `10,000` shares, Bob simply generates *Address 2*, locks another **`1,000` tokens**, and successfully triggers the Tier 1 minting allowance a second time.
  3. Bob transfers the `10,000` shares from *Address 2* back to *Address 1*.
  - **Total capital locked by Bob:** `2,000` tokens for `20,000` shares.

Example 2: Multiple Tiers Scaling Efficiency: Assume the contract is configured with two active tier configurations:

- **Tier 1:** Lock `1,000` tokens -> Cumulative Mint Allowance: `10,000` shares.
- **Tier 2:** Lock `3,000` tokens -> Cumulative Mint Allowance: `20,000` shares.
- **Scenario A: Honest User (Single Address)**.
  1. Alice locks `1,000` tokens and mints her first batch of `10,000` shares.
  2. Later, Alice wants to mint another `10,000` shares from the same address. Because her account's cumulative requirement scales up, she must lock an **additional `2,000` tokens** (bringing her total lock to `3,000` tokens) to reach Tier 2 and obtain the cumulative `20,000` share allowance headroom.
  - **Total capital locked by Alice:** `3,000` tokens for `20,000` shares.
- **Scenario B: Sybil Attacker (Multiple Addresses)**.
  1. Bob locks `1,000` tokens using *Address 1* and mints `10,000` shares.
  2. Instead of upgrading to Tier 2 on the same account, Bob generates *Address 2*, locks only **`1,000` tokens**, and mints an additional `10,000` shares.
  3. Bob transfers the `10,000` shares from *Address 2* back to *Address 1*.
  - **Total capital locked by Bob:** `2,000` tokens for `20,000` shares.

Bob achieves the exact same mint allocation outcome as Alice while risking 33% less lock collateral, penalizing legitimate single-address users.

Recommendation: Redesign the allocation tracking mechanism to remove the single-address cumulative penalty:

1. **Per-Transaction Pricing:** Transition from a cumulative tier system to a uniform, linear pricing model where every unit of minted allocation always requires a standard, fixed quantity of locked assets.

Kinetiq: Acknowledged.

Cantina Managed: Acknowledged.

3.4 Gas Optimization

3.4.1 Gas Optimization via `msg.sender` in `MarketBonded` Event

Severity: Gas Optimization

Context: EXFactory.sol#L257

Description: The `bondMarket` function uses `info.deployer` to specify the operator address when emitting the `MarketBonded` event.

Because the function explicitly validates that the caller is the deployer using an authorization guard (`if (info.deployer != msg.sender) revert Errors.NotAuthorized();`), `msg.sender` and `info.deployer` are guaranteed to hold identical values at execution time. Reading `info.deployer` forces the EVM to perform an expensive storage read (`SLOAD`), whereas reading `msg.sender` fetches the value from execution context memory, resulting in unnecessary gas costs for every invocation.

Recommendation: Update the event emission parameters to utilize `msg.sender` directly instead of reading `info.deployer` from the storage struct to save gas.

3.4.2 Redundant Storage of `exManager` Contract Address

Severity: Gas Optimization

Context: EXFactory.sol#L199

Description: The `deployMarket` function stores the `c.exManager` address redundantly inside two distinct storage mappings mapping to the same `marketId_` :

1. It is stored as the `exManager` field inside the `MarketInfo` struct within the `_markets` mapping.
2. It is stored implicitly as part of the full `MarketContracts memory c` struct within the `_marketContracts` mapping.

Because `_marketContracts[marketId_]` already contains a complete copy of the `MarketContracts` struct (which includes `exManager`), explicitly persisting it again inside the `MarketInfo` struct incurs an unnecessary `SSTORE` operation. This duplicates storage slots and increases deployment gas costs for every market creation.

Recommendation: Remove the `exManager` field from the `MarketInfo` struct definition entirely. Any internal or external execution flows requiring the `exManager` address for a specific `marketId_` should read it directly from the existing `_marketContracts` mapping instead.

3.4.3 Gas Optimization via Ordered Array Validation for Deduplication

Severity: Gas Optimization

Context: CompositeGate.sol#L173

Description: The `_setGates` function employs a nested loop to check for duplicates in the `gateAddresses` array, leading to an expensive `O(N^2)` time complexity.

This can be optimized to linear `O(N)` time complexity by requiring the input array to be strictly ordered off-chain (sorted in ascending order). With this requirement, the contract only needs to verify that each element is strictly greater than the preceding one (`gateAddresses[i] > gateAddresses[i - 1]`). This single condition guarantees both uniqueness and sorting order, eliminating the nested loop entirely and saving gas.

Recommendation: Remove the nested loop and enforce ascending order verification. Loop through the array starting from index 1, validating that each entry is strictly greater than the previous entry.

3.5 Informational

3.5.1 Inconsistent Storage Layouts in Upgradeable Contracts

Severity: Informational

Context: BlockedWithdrawalQueue.sol#L31

Description: Multiple upgradeable contracts in the repository utilize a linear storage layout architecture. Across different code versions, these layouts are being modified by removing state variables or inserting new ones into the middle of variable sequences.

Even if these code versions are intended for fresh deployments rather than upgrades to active contracts, this practice violates smart contract development best practices. In upgradeable architectures, linear storage maps sequentially to slots. Modifying the middle of a layout or deleting variables shifts the slots of all subsequent variables. Maintaining inconsistent layouts across versions destroys predictability. If any future upgrade path, off-chain indexing service, or development tool assumes storage continuity based on the repository's historical design, it will break or cause critical state corruption.

Consider the example of `BlockedWithdrawalQueue` in which `chunkSize` was a storage variable that was removed — it will corrupt the contract storage in case of an upgrade.

Recommendation:

1. **Adopt Namespaced Storage:** For contracts requiring entirely different storage requirements across versions, transition to a namespaced storage pattern (such as ERC-7201) to safely isolate variables via slot hashing.
2. **Implement Automated Storage Layout Validation in CI/CD:** To prevent human error from introducing storage regressions into production, integrate an automated storage collision check directly into the repository's CI/CD pipeline. Use established industry tools such as `@openzeppelin/hardhat-upgrades` (validation task) or Foundry's `forge inspect <Contract> storage-layout` to generate and compare storage layouts between the target branch and the main/production branch. Configure the pipeline to automatically fail the build if any unauthorized storage mutations, slot shifting, or layout collisions are detected on upgradeable contracts.

3.5.2 Ambiguous Variable Naming for Tier Identifiers in `queueTierUpgrade`

Severity: Informational

Context: EXManager.sol#L422

Description: The `queueTierUpgrade` function accepts a `uint256 newTier` parameter which acts as a lookup key for the `globalConfig.marketTiers` mapping. Naming a numerical identifier `newTier` creates code ambiguity, as it implies the variable holds the actual configuration data object rather than its index or identifier. This semantic inconsistency extends to the internal state variables (like `pendingTier` and `marketTier`) and events, which should explicitly denote that they store or emit IDs rather than full structs.

Recommendation: Rename the function parameter `newTier` to `newTierId` and systematically update all related variables, storage slots, and event parameters within the function scope to consistently use `Id` terminology (e.g., `marketTierId`, `pendingTierId`, and `newTierId`). This ensures clear distinction between the reference pointer and the fetched data object, improving readability and safeguarding against future refactoring errors.

3.5.3 Inaccurate Role Reference in `forceUnwindPhase` Comment

Severity: Informational

Context: EXManager.sol#L377

Description: The `forceUnwindPhase` function restricts access exclusively to accounts possessing the `RECOVERY_ROLE` via the `onlyRole(RECOVERY_ROLE)` modifier. However, an inline comment inside the function state validation logic mistakenly refers to the actor as the "protocol admin" instead of the "recovery role":

```
// protocol admin can only cancel their own forced unwinds, not operator-initiated ones
```

This naming discrepancy creates architectural ambiguity, as "protocol admin" typically implies a different privilege level (such as `DEFAULT_ADMIN_ROLE` or `CONFIG_ADMIN`), potentially confusing developers or auditors regarding which role handles emergency phase unwinding.

Recommendation: Update the inline comment to precisely reflect the authorized role (`RECOVERY_ROLE`).

3.5.4 Lost Context in `Unwound` Event Due to Early State Reset

Severity: Informational

Context: EXManager.sol#L414

Description: The `unwind` function clears the queue tracking state before emitting the `Unwound` event. Specifically, `protocolInitiatedUnwind` is set to `false` prior to the execution of the `emit` statement.

Because events capture the exact state at their execution point, resetting `protocolInitiatedUnwind` to `false` before emitting the event prevents downstream indexers, subgraphs, and off-chain monitoring tools from distinguishing whether the finalized unwind was originally initiated by an operator or via a forced recovery action. Including this origin flag directly inside the event parameters is essential for historical auditability and accurate state tracking.

Recommendation: Add a boolean flag (such as `isProtocolInitiated`) to the `Unwound` event definition. Within the `unwind` function, cache the value of `protocolInitiatedUnwind` into a local memory variable before resetting the state, and pass that cached value into the emitted event to preserve the initiation origin context.

3.5.5 Hardcoded Token Decimal Equivalence Across Asset Tracking Configurations

Severity: Informational

Context: WhitelistGate.sol#L342

Description: In `EXFactory.sol`, the contract manages the operator bond via `opBondEscrowed`, which represents an amount denominated in native `HYPE` tokens sent via `msg.value` during market deployment. This exact raw `HYPE` amount is subsequently passed to `EXManager.bond()` when finalizing the setup.

However, inside `EXManager.sol`, the `bond()` function assigns this numerical amount directly to `opBond`, where it is treated and utilized as a denomination of minted `exLST` shares rather than underlying `HYPE` tokens. For instance, during the contract termination phase in `unwind()`, the contract refunds the operator by sweeping `exLST` shares based directly on this value: `_pay(address(exLST), address(this), deployer, opBond);`

This cross-denominated design functions correctly only under the assumption that `HYPE` and `exLST` always maintain identical decimal configurations (18 decimals). If a future codebase version introduces an `exLST` variant with different decimals or incorporates dynamic exchange scaling between the underlying

asset and the shares, the accounting will suffer critical numerical errors, causing either an extreme under/over-payment or locked funds.

Recommendation: Standardize token accounting metrics by applying decimal normalization at all boundary points where raw values are moved between `HYPE` tracking and `exLST` share systems:

1. **Apply Explicit Scaling:** When translating the escrowed value to internal `exLST` configurations inside `EXManager`, dynamically scale the amount by comparing the decimals of `exLST` against `HYPE` (or native formatting constraints).
2. **Convert Values at Execution Points:** Instead of a direct raw assignment, adjust values at transfer and sweep boundaries to match the precision requirements of the specific token being handled by that code path.

3.5.6 Missing Logical Separation of LST and Launch Proxies in `deployMarketContracts`

Severity: Informational

Context: EXDeployer.sol#L49

Description: The `deployMarketContracts` function deploys 11 interconnected proxy and infrastructure instances in a continuous sequence. While these deployments are structurally sound, the underlying components belong to two entirely distinct functional layers of the protocol: the Liquid Staking Token (LST) infrastructure layer and the Launch infrastructure layer.

The current lack of code comments or physical partitioning between these two distinct categories degrades readability and makes it harder for reviewers, developers, and auditors to verify that all necessary components for each domain are completely and correctly initialized.

Recommendation: Add clear, structural inline comments to partition the deployment block into explicit, documented sub-sections separating the LST core components (such as `router`, `stakingAccountant`, `validatorManager`, `oracleManager`, `rewardShareTracker`, `ghostLST`, `defaultOracle`, and `oracleAdapter`) from the Launch-specific contracts (such as `exManager`, `exLST`, `bwq`, `launchFeeSplitter`, and `stakeFeesThrottle`).

3.5.7 Redundant and Dead Code Execution in `bond()`

Severity: Informational

Context: EXManager.sol#L296

Description: The `bond` function executes a conditional share sweeping mechanism to return excess token shares to the market deployer:

```
_pay(address(exLST), address(this), deployer, shares - opBond);
```

This internal transfer relies on the assumption that `shares` generated during initial staking can exceed the required operator bond value (`opBond`). However, at the very first protocol stake (when the initial pool state transitions from `UNBONDED`), the conversion ratio between the incoming asset and the minted `exLST` shares is mathematically locked at a fixed 1:1 parity baseline.

As a result, `shares` will always equal `opBond` at this exact phase transition. Because `shares - opBond` inevitably evaluates to zero, this transfer operation will never process an actual token payload. Retaining this operational block forces unnecessary mathematical operations and variable evaluation at runtime, wasting transaction gas during initialization.

Recommendation: Remove the dead parameter calculation and the redundant `_pay` function execution from the activation execution flow.

3.5.8 `EXRouter.queueWithdrawalAvailability()` : Unassigned Return Variable `instantWithdrawalCapacity`

Severity: Informational

Context: EXRouter.sol#L187

Description: The function `queueWithdrawalAvailability()` declares `instantWithdrawalCapacity` as a named return variable but never assigns it a value. This causes the function to implicitly return a default value of `0`, which can break external integrations or internal logic expecting an actual capacity value.

Recommendation: Either calculate and assign the correct value to `instantWithdrawalCapacity` before the function returns, or remove it from the return signature entirely if it is dead code.

3.5.9 Missing Fields in Withdrawal Events

Severity: Informational

Context: BlockedWithdrawalQueue.sol#L268

Description: The `BlockedWithdrawalConfirmed` event is missing both the `feesToPay` and `exTreasury` parameters. Additionally, the `exTreasury` parameter is missing from the standard `WithdrawalConfirmed` event. Excluding these fields limits off-chain tracking and transparency for treasury actions and fee distributions.

Recommendation: Add both `feesToPay` and `exTreasury` to the `BlockedWithdrawalConfirmed` event, and add `exTreasury` to the `WithdrawalConfirmed` event. Ensure these values are properly emitted alongside the other event data.

3.5.10 Inconsistent Handling of Already Confirmed Withdrawals

Severity: Informational

Context: BlockedWithdrawalQueue.sol#L267

Description: There is a logic inconsistency between `BlockedWithdrawalQueue.confirmBlockedWithdrawal()` and `ExManager.confirmWithdrawal()` regarding how already-confirmed withdrawals are handled. When `amountToPay` is `0` (indicating the withdrawal has already been processed or confirmed), `confirmBlockedWithdrawal()` does not return early. Instead, it proceeds through the execution flow and erroneously emits a `BlockedWithdrawalConfirmed` event.

Recommendation: Modify `BlockedWithdrawalQueue.confirmBlockedWithdrawal()` to include an early return for already confirmed withdrawals to align its behavior with `ExManager.confirmWithdrawal()` and prevent the emission of redundant events.