yAudit

October 2026

Prepared for
Lynxify

Audited by
fedebianu
devtooligan
yBorg fellows

LYNX LP Range Manager

Smart Contract Security Assessment

Contents

1 Review Summary **3**
1.1 Protocol Overview . . . 3
1.2 Audit Scope . . . 3
1.3 Risk Assessment Framework . . . 3
1.3.1 Severity Classification . . . 4
1.4 Key Findings . . . 4
1.5 Overall Assessment . . . 4

2 Audit Overview **4**
2.1 Project Information . . . 5
2.2 Audit Timeline . . . 5
2.3 Audit Resources . . . 5
2.4 Critical Findings . . . 5
2.5 High Findings . . . 5
2.6 Medium Findings . . . 5
2.6.1 LP minimums are priced at the TWAP while the NPM fills at spot, so rotates and deposits on concentrated ranges revert after a few ticks of drift 5
2.6.2 `_onChainLiquidity()` reads the wrong word of the production `positions()` return, so rotates and both emergency exits always revert . . . 7
2.6.3 One persistently reverting target starves range management of every later target . . . 9
2.7 Low Findings . . . 11
2.7.1 A reverting collect blocks the paged harvest for every later position . . . 11
2.7.2 The fixed LP mint fee overpays SaucerSwap, and the excess is refunded to the next NPM caller . . . 12
2.7.3 Stale oorStreak lets a target rotate early . . . 13
2.7.4 Phase 2 overflows the 50 child-record cap . . . 15

3 6 covers the same phase-2 coupling for catchable reverts. **16**
3.0.1 `deployLpIdle()` swaps a fixed share of each target's budget regardless of its range, so concentrated and out-of-range targets leave much of every deploy idle . . . 16
3.0.2 The playbook's stop/start recovery replays phase 3 and can confirm a rotate on a single out-of-range observation . . . 17
3.0.3 Drain proceeds are not pulled back once the LP strategy's principal reaches zero . . . 19
3.0.4 A failed drain rolls back the same phase's buffer refill . . . 20
3.0.5 A pair-token donation can permanently block the drain of a disabled LP position . . . 21
3.0.6 The allocator's governance powers cannot be exercised through `LynxGovernor` . . . 22
3.0.7 `emergencyWiden()` still depends on a ready TWAP close to spot, so it fails in the conditions it is advertised for . . . 23
3.0.8 Permissionless `retrySchedule()` on phase 0 mid-pass postpones the cycle by a week and keeps the harvest lock . . . 25
3.1 Gas Savings Findings . . . 26
3.2 Informational Findings . . . 26
3.2.1 Range policies near the maximum width cannot be built when the TWAP is far from tick 0 . . . 26

3.2.2 `_clearPendingSchedule()` ignores the response code of `deleteSchedule()` . . . 27
3.2.3 A replacement mint deploys credited pair-token fees and erases their credit, so the fees skip the profit lock . . . 28
3.2.4 Harvest swap backlog can outgrow one swap . . . 29
3.2.5 Rotates keep deploying idle capital while the vault is paused . . . 30
3.2.6 Cardinality is treated as TWAP readiness, so a pool with short observation history can block vault settlement . . . 31
3.2.7 The post-unwind recovery in the NatSpec and keeper playbook omits the required strategy ownership handover . . . 32
3.2.8 Settlement valuation relies on a raw pool TWAP and counts unsolicited pair-token balances . . . 33
3.2.9 Swap `minOut` quotes ignore the pool fee, so swaps on 1% pools fail in practice . . . 34
3.2.10 A partially filled harvest swap erases the fee credit of tokens that were not sold . . . 35
3.2.11 `emergencyWiden()` is not payable and depends on the allocator's leftover HBAR for the mint fee . . . 37
3.2.12 OOR divert flag is computed from pre-rotate state, so a successful rotate can still divert the week's surplus . . . 37
3.2.13 Owner-side harvest-lock hatches on the LP strategy are unreachable once the allocator owns it . . . 38

1 Review Summary

1.1 Protocol Overview

Lynxify runs LYNX, a yield aggregator on Hedera. Users deposit HBAR into an ERC-4626 vault, and a strategy manager invests it in SaucerSwap V3 liquidity positions and HBAR staking. This review covers the code added since the previous yAudit review: the paged LP harvest and the LP Range Manager. `SaucerSwapLiquidityStrategy` holds the vault's concentrated-liquidity positions. Its harvest now runs one page of collects or swaps per call, so each transaction stays under Hedera's limit of 50 child records. `LynxLpAllocator` gives each LP target a range policy. When a position stays out of range for a set number of weekly cycles, the allocator closes it and mints a new one around the pool's TWAP, and the owner and governance can widen or unwind a position in an emergency. `LynxKeeperRouter` runs the weekly loop through the Hedera Schedule Service in six phases, one per transaction: paged harvest, staking harvest, buffer refill and drain, range operations, surplus allocation and idle LP deployment. `LpRangeMath` and `TickMath` hold the range and price math. The live mainnet stack still runs the earlier four-phase keeper, and neither the paged harvest nor the LP Range Manager is deployed.

1.2 Audit Scope

This review covers 7 Solidity files, totaling approximately 3078 lines of code. It focuses on the changes since commit `ca33ab4`, the baseline after the previous yAudit review. It was an AI-driven review: AI agents searched the code, and the auditors triaged and verified every reported issue before including it in this report. The auditors also manually reviewed the most critical parts of the code. Some findings come from the yAudit Fellows' review of the same commit.

```
hardhat/contracts
├── LynxKeeperRouter.sol
├── LynxLpAllocator.sol
├── SaucerSwapLiquidityStrategy.sol
├── interfaces
│   ├── ILpHarvestPaged.sol
│   └── ILynxLpAllocator.sol
└── libraries
    ├── LpRangeMath.sol
    └── TickMath.sol
```

1.3 Risk Assessment Framework

1.3.1 Severity Classification

Severity	Description	Potential Impact
Critical	Immediate threat to user funds or protocol integrity	Direct loss of funds, protocol compromise
High	Significant security risk requiring urgent attention	Potential fund loss, major functionality disruption
Medium	Important issue that should be addressed	Limited fund risk, functionality concerns
Low	Minor issue with minimal impact	Best practice violations, minor inefficiencies
Undetermined	Findings whose impact could not be fully assessed within the time constraints of the engagement. These issues may range from low to critical severity, and although their exact consequences remain uncertain, they present a sufficient potential risk to warrant attention and remediation.	Varies based on actual severity
Gas	Findings that can improve the gas efficiency of the contracts.	Increased transaction costs
Informational	Code quality and best practice recommendations	Reduced maintainability and readability

Table 1: severity classification

1.4 Key Findings

Breakdown of Finding Impacts

Impact Level	Count
Critical	0
High	0
Medium	3
Low	12
Informational	13

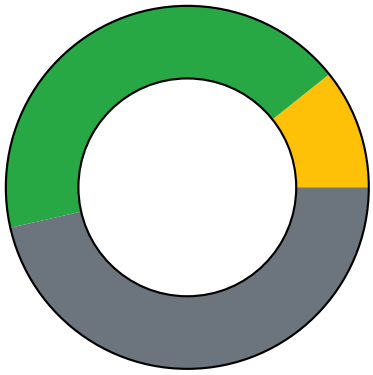


Figure 1: Distribution of security findings by impact level

1.5 Overall Assessment

2 Audit Overview

2.1 Project Information

Protocol Name: Lynxify
Repository: https://github.com/Lynxify-xyz/lynx-contracts
Commit Hash: 00827435615f3e81f3ed81e39e7b4a84005fd92d
Final Commit Hash: 3470f5fa3df95eaa393815494f06483ae98577b4
Commit URL: https://github.com/Lynxify-xyz/lynx-contracts/tree/00827435615f3e81f3ed81e39e7b4a84005fd92d

2.2 Audit Timeline

The audit was conducted from October 6 to 7, 2026.

2.3 Audit Resources

- Code repository, the audit delta one-pager and the team's internal deep audit of the LP Range Manager

2.4 Critical Findings

None.

2.5 High Findings

None.

2.6 Medium Findings

2.6.1 LP minimums are priced at the TWAP while the NPM fills at spot, so rotates and deposits on concentrated ranges revert after a few ticks of drift

Technical Details

`LpRangeMath.planDeploy()` sizes the liquidity and both minimums at the price it receives:

```
uint128 liq = LiquidityAmounts.getLiquidityForAmounts(sqrtPriceX96, sqrtA, sqrtB,
amount0Desired, amount1Desired);
(uint256 amount0, uint256 amount1) = LiquidityAmounts.getAmountsForLiquidity(
sqrtPriceX96, sqrtA, sqrtB, liq);
return (_applyMinBps(amount0, lpSlippageBps), _applyMinBps(amount1, lpSlippageBps));
```

The allocator always passes the 30-minute TWAP:

- **Rotate mint:** `_mintPlanned()` uses the TWAP captured once per rotate (L1413-L1416).
- **Weekly deposit:** `_increaseLpPosition()` uses `_trustedSqrtPriceX96()`, which returns the TWAP once the spot-deviation check passes.
- **Drain of a disabled position:** `decreaseMins()` prices the minimums at the same TWAP.

The NPM sizes the liquidity at the pool's current `slot0()` price, then rejects the call if either amount used is below its minimum. On a full-range position, the token split barely moves with price. On a concentrated range it moves quickly, so a small gap between spot and the TWAP leaves one token short by more than `lpSlippageBps`. The default is 100 bps and the maximum is 500.

Rotate mint. The surplus swap targets the TWAP ratio, so the post-swap balances match the TWAP split and each minimum is about 99% of the balance. At any other spot price, one token is under-used, so the mint fails in both directions. The table gives the gap in ticks at which the mint fails, for a range centred on the TWAP, computed with Uniswap V3 liquidity math:

Half-width	`lpSlippageBps` = 100	`lpSlippageBps` = 500
±240	2 ticks	7 ticks
±600	3 ticks	16 ticks
±6000	27 ticks	133 ticks

Other paths.

- **Weekly deposit:** its desired amounts come from a fixed-share swap at spot (see #17), so it fails in one direction only.
- **Drain decrease:** a ±600 position centred on the TWAP fails at 6 ticks.
- **`emergencyWiden()`:** it prices its mint at spot, but it reads spot (L642) before its own surplus swap (L650). When the new range contains spot, the mint at L654 executes at the price the swap moved, so a concentrated widen can fail on the swap's own price impact, as the rotate does. A full-range widen, or one whose new range does not contain spot (no swap), is not affected.

A gap of a few ticks between spot and the 30-minute TWAP is the normal state of an active pool. On 2026-10-12 at 13:50 UTC, USDC/WHBAR spot was 4.4 ticks from its TWAP and SAUCE/WHBAR spot was 19.1 ticks away. At those gaps, a ±600 rotate would revert on USDC at the default setting, and on SAUCE even at the 500 bps maximum.

The current positions are full range and unaffected. #22 currently makes every rotate revert earlier, so this issue appears once #22 is fixed.

Impact

Medium. Concentrated ranges are what the range manager is for, and once they are in use:

- **Rotates** fail in most weeks. Each failed rotate consumes the week's attempt and blocks later targets (see #1). The out-of-range position earns nothing, and the divert keeps sending the surplus to staking.
- **Weekly deposits** often fail, and the target's WHBAR stays idle on the strategy.

No principal is lost.

Recommendation

Do not just move the minimums to spot. The TWAP-priced minimums also protect the mint against a manipulated spot price. Either:

- compute the minimums at spot, and bound the spot/TWAP gap for LP calls with a separate, tight limit (for example 50–100 bps). In the rotate and in `emergencyWiden()`, read spot after the surplus swap, not before it; or
- pass loose per-token minimums, and after `mint()` or `increaseLiquidity()` require the value received, at the TWAP, to be at least `1 - lpSlippageBps` of the value spent, also at the TWAP. Apply the same check to decreases.

Developer Response

Fixed in commits ced1133, 3623754, ecf1cff, a2dcd18, 2354f6d, 9ebaf30.

2.6.2 `_onChainLiquidity()` reads the wrong word of the production `positions()` return, so rotates and both emergency exits always revert

Technical Details

`_onChainLiquidity()` reads a position's liquidity from the raw return data of the NPM's `positions()`:

```
if (!ok || ret.length < 0x100) { /* bubble up */ }
assembly {
    liquidity := mload(add(ret, 0x100))
}
```

`mload(add(ret, 0x100))` loads return word 7, counting from 0. That is `liquidity` in the 12-word Uniswap V3 layout. The repository declares that layout in its `ISaucerSwapV2NonfungiblePositionManager` interface and implements it in `MockSaucerSwapV2NftManager.positions()`.

The production SaucerSwap V2 NPM (`0.0.4053945`) returns 10 words, without `nonce` and `operator`. See `NonfungiblePositionManager.positions()` in its verified source:

```
token0, token1, fee, tickLower, tickUpper, liquidity,
feeGrowthInside0LastX128, feeGrowthInside1LastX128, tokensOwed0, tokensOwed1
```

In production, `liquidity` is word 5 and word 7 is `feeGrowthInside1LastX128`. The length check still passes, because the return is 320 bytes. A read of the live launch position on 2026-10-12 shows the mismatch:

`positions(76608)`	Value
Word 5 (`liquidity`)	43,684,291,439. This equals the liquidity that mainnet strategy
Word 7 (`feeGrowthInside1LastX128`)	About 2.04×10^3

`closeLPPosition()` and `rotateClosePosition()` both decrease the returned amount before `burn`. The NPM rejects a decrease larger than the position's liquidity, so both functions revert. The only exception is a fee-growth word of zero, which effectively never occurs. Three allocator paths go through these functions:

- **Keeper rotate:** `_finishRotate()` first decreases the stored liquidity, then calls `rotateClosePosition()` to collect and burn. That decrease also refreshes the position's fee-growth snapshot, so word 7 is still non-zero.
- `emergencyWiden()`, through `rotateClosePosition()`.
- `emergencyUnwind()`, through `closeLPPosition()`.

The repository tests use the 12-word mock, so they pass.

The read was added by the fix for internal finding #3 (PT-012), which protects the close paths against liquidity that a third party donates to the NFT:

- `ca33ab4` decreased the stored `position.liquidity`.
- PT-012 (`2810bb2`) decoded `positions()` through the 12-word interface. That decode would also have reverted on the 10-word return.
- `281eafb` replaced it with the raw word-7 read.

The mainnet strategy `0.0.10824909` predates this change. The issue applies from the deployment of the new strategy.

Impact

Medium. From the first day of the new strategy, the range manager cannot rotate any position, and both break-glass paths revert in every market condition.

- **Rotates:** the keeper's attempt reverts every week. Phase 3 catches the revert, counts it as the week's attempt and reports it as a rotate (see #1). The first confirmed target therefore also blocks every target after it.
- **Out-of-range positions:** they stay out of range and earn no fees, and the divert keeps sending the surplus to staking.
- **Emergency exits:** `emergencyWiden()` and `emergencyUnwind()`, the documented escape hatches, are unusable.

Principal is not lost. The drain path decreases the stored liquidity (`_decreaseDisabledPosition()`), so the owner can disable a target and let phase 2 drain it, subject to #5 and #12.

Recommendation

Read `liquidity` from word 5 and require the exact production return length:

```
-         if (!ok || ret.length < 0x100) {
+         if (!ok || ret.length != 0x140) {
              assembly {
                  revert(add(ret, 0x20), mload(ret))
              }
          }
          assembly {
-             liquidity := mload(add(ret, 0x100))
+             liquidity := mload(add(ret, 0xc0))
          }
```

Also align the interface and `MockSaucerSwapV2NftManager.positions()` with the 10-word production layout, and fix the NatSpec ("the 8th return"). Add a mainnet-fork test that checks `_onChainLiquidity()` against the stored liquidity of a live position.

Developer Response

Fixed in commit b794fbf.

2.6.3 One persistently reverting target starves range management of every later target

Technical Details

`LynxKeeperRouter._rangeOps()` always walks the allocator targets from index zero and calls `rebalanceOne()` until one attempt is "used". The fix for internal audit finding #7 sets `rotated = true` in the `catch` branch as well as on success:

```
if (!rotated) {
    try lpAllocator.rebalanceOne(t.positionSN, deadline) returns (bool r) {
        if (r) rotated = true;
    } catch {
        rotated = true; // at most one attempt per phase
    }
}
```

Bounding the phase to one attempt is needed to stay under Hedera's child-record cap. But every revert counts as the consumed attempt, and there is no persistent cursor between cycles. A target that reverts every week therefore keeps every target after it from ever reaching `rebalanceOne()`. `_evaluateRotate()` only runs inside `rebalanceOne()`, so for the skipped targets `oorStreak` never increments, the "in range" reset never runs, and no rotate can fire. The `checkRangeStatus()` call the router makes for them does not maintain this state. Monitoring does not see the condition either: after a reverted attempt `rotated` is `true`, so the phase emits `RangeOpsCompleted(true, …)` as if a rotate had executed.

`rebalanceOne()` can revert persistently for several reasons:

- **Swap size versus pool depth.** The decrease of an out-of-range position returns a single leg, so the rotate needs the ratio swap. `_swapSurplus()` sells roughly half of the position's value, plus any idle balances the rotate sweeps in by design (internal #15). Its `minOut` is

a linear TWAP quote minus `swapSlippageBps` (default 1%), with no allowance for the pool fee or price impact. The strategy also caps execution at a TWAP-derived price limit (`harvestSwapSlippageBps`, 100 bps on sqrt, about 2% on price). When the swap is large relative to the pool's in-range depth, it cannot meet these bounds, and the rotate reverts on every attempt. On a 1% fee tier with the default slippage it fails in practice (see #12).

- **Spot displaced from the TWAP.** A position goes out of range because the price moved, and the rotate sells the leg the move pushed the position into. While the move continues, spot sits past the 30-minute TWAP in that direction. The swap falls short of its TWAP-based `minOut` once that displacement exceeds the margin left after the pool fee. Once spot is about 2% past the TWAP, the strategy's price limit is behind spot and the pool rejects the swap (`SPL`). `SpotTwapDeviationExceeded` and the NPM check on the TWAP-priced mint minimums fail in the same conditions.
- **An attacker-induced revert.** `_swapSurplus()` sizes the ratio swap from the strategy's *full* WHBAR and pair-token balances. Anyone can donate pair tokens to the strategy and enlarge the swap until its own price impact breaks `minOut`. Every rotate attempt for that target then reverts. The same donation pattern blocks `drainPosition()` (see #5).
- **A short oracle buffer.** After the target-enabled and policy checks, and before any `_evaluateRotate()` gate, `rebalanceOne()` reads the TWAP tick through `_getTwapTick()`. That call reverts `TwapNotReady` when `observe([twapWindowSec, 0])` fails, while the `_isTwapReady()` check before it only compares `observationCardinality` with `minObservationCardinality` (default 2). A target with a range policy on a pool whose ring buffer does not cover the 1800 s window therefore reverts even when its position is in range. The same root cause is described for valuation in #15. It does not affect the current mainnet catalog: on 2026-10-08 the five catalog pools reported an observation cardinality of 500 to 1200, and all of them served `observe([1800, 0])`. It applies to newly added or low-cardinality pools.
- `InvalidRange` from `planRotate()` near the tick bounds, and `InsufficientMintFee` while the router treasury cannot fund the allocator.

Each revert rolls back the allocator's tracking writes but not the router's consumed attempt, so the same target reverts again the next week.

This is a regression from the fix for internal finding #7. Before that fix, a revert let the loop try the next target.

Impact

Medium. One target whose rotate keeps reverting can permanently disable automated range management for every target after it in `targets[]`. No attacker is needed: a position that is large for its pool's depth, or a 1% fee tier, is enough. An attacker can also force the condition by sacrificing pair tokens. The skipped positions can stay out of range indefinitely and earn no fees. Because `checkRangeStatus()` keeps reporting them as out of range, `oorDivertActive` stays set and phase 4 sends 100% of the vault surplus to staking every week, so the LP sleeve receives no new capital. The phase reports each reverted attempt as a rotate, so the pattern is easy to miss. Recovery requires an operator to notice it and then reorder or disable targets, freeze range ops, or use the emergency paths. No principal is transferred to an attacker.

Recommendation

Keep the one-execution-attempt budget, but stop a single target from consuming it forever:

- Advance a persistent selection cursor after every execution attempt, including failures, so a failing target cannot starve the others.
- Separate evaluation from execution. Run `_evaluateRotate()` for every enabled target, since it only reads the TWAP and writes streak state, then execute at most one confirmed candidate.
- Make the pre-gates non-reverting. In `rebalanceOne()`, use `LpRangeMath.tryTwapTick()` and `return false` (emitting `RotateSkipped("twap not ready")`) instead of reverting through `_getTwapTick()`. Make `_isTwapReady()` verify that `observe(twapWindowSec)` succeeds, not only the cardinality.
- Make the rotate swap executable. Include the pool fee in `minOut` (see #12), and size the swap from the balances the decrease produced rather than the strategy-wide balances, so a donation cannot inflate it. Because the replacement mint deploys the full balances by design (internal #15), the mint amounts must follow the same scoping.
- Report a reverted attempt separately from an executed rotate. For example, emit an event in the `catch` branch with `positionSN` and the revert data, and add a distinct flag to `RangeOpsCompleted`.

Developer Response

Fixed in commits 25d3de3, 7d3902b, f0384d3, 6cec9e0.

2.7 Low Findings

2.7.1 A reverting collect blocks the paged harvest for every later position

Technical Details

`_collectHarvestPage()` walks `lpPositionSerials` from `harvestCollectIndex` and calls `_collect()` on each active serial. `_collect()` makes a single NPM `collect()` that transfers both tokens to the strategy. If either transfer fails, the whole page reverts, including the cursor increment. The swap page isolates failures per token (`HarvestPairSwapFailed`); the collect page does not.

On Hedera, a transfer can fail without any change on the Lynx side. A token's issuer can freeze an account (the strategy's or the pool's), pause the token, or revoke KYC. Of the current catalog's pair tokens, only USDC (`0.0.456858`) has a freeze key, and none has a pause or KYC key.

When a page fails, the router does the following:

1. It treats the failure as a phase-0 failure and runs phase 0 again, up to `MAX_PHASE0_FAILURES` = 8 failed attempts in total (`_shouldStayOnPhase0()`). Each attempt is a paid scheduled execution.
2. It emits `LpHarvestGiveUp` and leaves phase 0.

3. `_advanceOpsPhase()` aborts the pass, which resets the cursor to 0 (`_completeHarvestPass()`).

The next week's pass starts again from the first serial and fails at the same one. As long as the condition lasts, the serials after it are never collected.

Impact

Low. While a pair token cannot be transferred to the strategy:

- the fees of every later position are not collected, so they do not reach NAV;
- each week, eight scheduled executions are spent on a failure that is certain to repeat.

The fees are not lost, because they stay in the NFTs. The affected position cannot be decreased, closed or rotated until the token can be transferred again; that part is unavoidable. The trigger requires an action by the token issuer, and in the current catalog only USDC's issuer can take it.

Recommendation

Isolate collects per serial. Call the NPM's `collect()` in a `try/catch`, emit an event with the serial and the revert data on failure, and advance the cursor past the failing serial. Credit fees only for successful collects. Keep the close and rotate paths fail-closed for the affected position.

Developer Response

Fixed in commits 134250d, 2354f6d.

2.7.2 The fixed LP mint fee overpays SaucerSwap, and the excess is refunded to the next NPM caller

Technical Details

`mintFeeHbar` defaults to 1 HBAR (L177). The strategy forwards it in full on every mint and increase: `createLPPosition()` and `increaseLPPosition()`.

The production NPM does not consume the whole amount. In its verified source, `LiquidityManagement.addLiquidity()` reads the factory's `mintFee()`, which is set in tinycents. It converts the fee through the exchange-rate precompile, adds 1 tinybar for rounding, and forwards only that amount to the pool. The rest stays on the NPM.
`PeripheryPayments.refundETH()` is permissionless and sends the NPM's entire HBAR balance to its caller. Neither strategy function calls it or checks its HBAR balance afterwards, so the excess goes to whoever calls `refundETH()` next.

Two mainnet transactions show this:

1. In `0xce638de3be8b624037478d7e7a1b6a1e181992ecb1e069b46107156586ba0d4e`, strategy `0.0.10824909` forwards 100,000,000 tinybar to the NPM. The NPM forwards 63,520,295 to the pool, so 36,479,705 tinybar stays on the NPM.

2. The next direct NPM call, `0x88ef00e540c9f0bbfe0dfb174aaa2af5456d9714908a0e2430fa131f609284fa`, comes from an unrelated account. Its own unused HBAR is 12,652,942 tinybar, but `refundETH()` pays it 49,132,647. The difference, 36,479,705, is exactly the strategy's leftover.

On 2026-10-12:

- the factory's `mintFee()` is 500,000,000 tinycents;
- the exchange rate is 30,000 HBAR = 278,664 cents;
- the NPM therefore charges 53,828,267 tinybar, and each mint or increase leaves about 46,171,733 tinybar (0.46 HBAR) for the next caller.

The code predates the delta. The delta adds paying call sites: the rotate and widen replacement mints (`_mintReplacement()`), funded by the allocator, in addition to the weekly per-target increases funded by the router.

The allocator has no passthrough for `setMintFeeHbar()`. Once it owns the strategy, the fee can only be changed through a strategy ownership handover.

Impact

Low. About 0.46 HBAR leaves the router or allocator treasury for an arbitrary third party on every LP mint and increase, which means every weekly deposit per target and every rotate or widen. The loss is direct and repeats every week, but it is small. A lower `mintFeeHbar` reduces it, but no fixed value removes it, because the required fee moves with the exchange rate.

Recommendation

Call the NPM's `refundETH()` in the same transaction, immediately after each `mint()` and `increaseLiquidity()`, or batch both through the NPM's `multicall()`. The strategy already has a `receive()` function, so it can receive the refund. Alternatively, compute the exact fee before each call from `factory.mintFee()` and the exchange-rate precompile, plus SaucerSwap's 1-tinybar rounding allowance.

Developer Response

Fixed in commits fbe16f4, ecbf405, 2354f6d.

2.7.3 Stale oorStreak lets a target rotate early

Technical Details

`LynxLpAllocator` documents `oorStreak` as counting "consecutive weekly OOR phases" with a reset on reentry, and the default `minOorCycles = 2` exists so that a single out-of-range reading does not trigger a rotate. The streak has no time key. `_evaluateRotate()` increments it on every out-of-range evaluation and resets it only at Gate 5, when it evaluates the target and finds it in range.

Range ops does not evaluate every enabled target each week:

- The router makes at most one `rebalanceOne()` attempt per run. A successful rotate, or a caught revert, of an earlier target sets `rotated = true`, so later targets are not passed to `rebalanceOne()` that week (L715-L726). This one-attempt rule is the fix for internal finding #7. The loop still reads each target's `checkRangeStatus()`.
- A frozen range phase returns before the loop (L676-L680).
- The TWAP-readiness pre-check (L519-L535) also skips Gate 5.

A target that comes back into range in a skipped week therefore keeps its old streak and `lastOorTick`:

1. Week N: concentrated targets A (index 0) and B (index 1) both read out of range. Each gets streak 1.
2. Week N+1: A rotates, or its rotate reverts and is caught. B is back in range but is not evaluated, so it keeps streak 1.
3. Week N+2: B leaves the band again, at least as far as in week N. B reaches streak 2 and rotates on this one new reading, although the previous week was in range.

The minimum-gap fix proposed in #16 does not close this path, because the stale reading is more than one interval old. The fix for internal finding #9 resets only the target that rotated.

A Hardhat test using the production router, allocator, LP strategy, StrategyManager and vault reproduces it. A's rotate in week N+1 leaves `oorStreak(B) = 1` while `checkRangeStatus(B)` reports in range, and B then rotates in week N+2 from `[-600,600]` to `[60,1320]`. In the control, B's in-range week is evaluated, its streak resets, and week N+2 produces only a candidate.

Impact

Low. The vault pays for one premature rotate of one LP position: the pool fee and up to `swapSlippageBps` on the swapped surplus, plus fees forgone if the price returns and the new range is out of range. Nothing is locked, and no third party gains beyond ordinary arbitrage. The outcome equals one rotate under the owner-permitted `minOorCycles = 1`.

It requires two or more concentrated targets that go out of range together, followed by an out, in, out price sequence. The current `lp-pools.json` targets are full-range.

Recommendation

1. On each unfrozen range-ops run, evaluate Gate 5 for every enabled, active target, independent of the single rotate attempt. At minimum, reset the streak when the target is in range. The router already reads `checkRangeStatus()` for each target.
2. Restart the streak at 1 when the previous out-of-range reading is older than about one `weeklyIntervalSec` plus slack, so a skipped or frozen week restarts confirmation instead of counting toward it.

For the same-week replay, see #16.

Developer Response

Fixed in commits 51e02c7, 93a62a2, 25d3de3, e1ad890.

2.7.4 Phase 2 overflows the 50 child-record cap

Technical Details

The keeper runs the weekly cycle as one HIP-1215 scheduled transaction per phase so that each transaction stays under Hedera's 50 child-record cap. Phase 2 does not stay within that budget. `_weeklyOpsBody()` runs `refillBuffer()`, `reconcileInactiveLp()` and `_pullLpRecoveredWhbar()` in one transaction, and `weeklyOps()` appends `deleteSchedule` and `scheduleCall` to it.

`refillBuffer()` pulls from the composite staking strategy, which pays from its native leg first and then swaps its XSAUCE, DOVU and HBARX hold legs. Each hold-leg swap is sized with an exact-output quote that ignores the pool fee, so each leg returns slightly less than requested and the refill moves on to the next leg. Once the native leg is empty, a refill swaps all three legs. The drain adds a decrease, a collect and a full-balance swap, and the recovery pull adds a second `manager.pull()`.

The overflow needs two conditions in the same week:

1. The composite native leg is empty. Under sustained net redemptions this happens after roughly 10% of TVL in cumulative outflow, because pushes refill the native leg only on surplus.
2. A funded LP position is not an enabled target, so a drain is pending. This follows any owner removal or deselection with `setTargets()`. On mainnet, phase 2 has been draining four unselected funded positions (NFTs 76609 to 76612), one per week: 76609 on 29 September, 76610 on 6 October and 76611 on 13 October. 76612 is still funded and is next.

A Hardhat test running the production router, allocator, LP strategy, manager, vault and composite staking contracts traces 68 child records for phase 2 at one target, and 67 to 76 at 1, 4 and 8 targets. These are modelled counts: the mocks do not enforce the cap or emit status 328. The model counts each HTS call, including `balanceOf` reads, as one record. If HTS reads produced no records, phase 2 would be about 18 to 20 records and would fit. The client's own mainnet measurements (keeper playbook: a swap at 7 to 8 records with about 3 mutating calls, a collect at 4.4 with 2 transfers) indicate that reads do count.

Status 328 cannot be caught, so the whole transaction rolls back. `opsPhase` stays at 2 and `pendingSchedule` still names the consumed schedule, so `retrySchedule()` reverts. The documented recovery (stop, then start the loop) replays phases 0 to 2 into the same overflow while both conditions hold.

This is related to, but not covered by, these issues:

- Internal residual AD-4 does not list phase 2.

- 3 6 covers the same phase-2 coupling for catchable reverts.

Impact

Low. While both conditions hold, the weekly loop stops at phase 2. The buffer refill, the drain, the recovery pull and phases 3 to 5 do not run, and redemptions are limited to the vault's idle balance. No funds are lost and NAV is unchanged because the rollback is total.

The owner can restore the loop without code changes. After a stop and start, the owner calls `refillBuffer()` directly, which `onlyOpsExecutor` allows. That runs the three-leg swap in its own transaction. The next scheduled phase 2 then carries only the drain and the recovery pull, modelled at about 37 records, and commits. No attacker is required and no one profits.

Recommendation

Move `reconcileInactiveLp()` and `_pullLpRecoveredWhbar()` out of phase 2 into their own scheduled phase immediately after the refill, as #6 also suggests. In that phase:

- Attempt one `drainPosition()` per transaction.
- Run the recovery pull in the same revert scope as that drain.
- Advance a round-robin cursor so that one failing position does not block the others.

Do not rely on try/catch inside the current phase 2. A caught revert on Hedera still commits its child records, so the transaction can still exceed the cap.

Until the split ships, add phase 2 to the AD-4 residual list and to the 328 runbook, including the `refillBuffer()` recovery above. After the split, phase 2 alone is modelled at about 45 to 48 records with the native leg empty. Measure it on a Hedera testnet.

Developer Response

Fixed in commits 79d4d3d, 619c5ee.

3.0.1 `deployLpIdle()` swaps a fixed share of each target's budget regardless of its range, so concentrated and out-of-range targets leave much of every deploy idle

Technical Details

For every enabled target, `_swapAndPrepareAmounts()` swaps `perTarget × swapBps / 10_000` WHBAR into the pair token. `swapBps` is a single global value (5000 in the catalog). `_increaseLpPosition()` then adds both legs. The NPM only takes as much of each leg as the position's current ratio requires, and the rest stays on the strategy.

A fixed split fits the full-range positions the allocator was built for, whose value split stays close to 50/50. Layer C makes targets concentrated, and the ratio of a concentrated position depends on where the price sits inside its range. It is 50/50 only at the geometric centre and reaches

100/0 at the edges. A rotate only re-centres a target after `minOorCycles` weekly confirmations, so positions spend most of their life off-centre. For example, if the position is 80/20 by value at the current price, a 50/50 deposit is limited by the scarce leg, and 37.5% of the budget stays idle.

An out-of-range target accepts only one token. Phase 5 still swaps half of that target's budget, adds the accepted leg single-sided to a position that earns no fees, and leaves the other leg idle. When the accepted leg is the pair token, the deploy keeps buying the token the market moved against and parks it in an inactive range. That is the opposite of what the OOR divert is meant to do ("keeps new capital out of an LP position that cannot earn"). The playbook notes that phase 5 "still runs after a divert" (`docs/keeper-playbook.md`).

What stays idle is not wasted outright, but it is not productive either:

- Idle WHBAR is re-split in later weeks and goes through the same 50/50 swap again.
- Idle pair tokens are not credited fees, so harvest does not convert them. They wait for the next rotate's full-balance sweep (internal #15).
- Every unused swap also pays the pool fee and price impact.

`LpRangeMath.quoteSurplus()` already computes the correct ratio for a given range and price, but only the rotate and widen paths use it.

Impact

Low. With concentrated ranges, a large and recurring share of the LP sleeve's weekly capital does not reach the liquidity it was pushed for, and out-of-range targets keep receiving single-sided adds. Yield is lost and swap costs are paid for nothing, but no principal is lost.

Recommendation

Size each target's deploy swap with `LpRangeMath.quoteSurplus()`, from the target's WHBAR budget and its range at the TWAP, as the rotate and widen paths already do. For targets whose TWAP is outside their range, either skip them and leave their share for the next cycle, or deploy only the leg they accept without swapping. With per-target sizing, `swapBps` is no longer needed.

Developer Response

Fixed in commit 48375e7.

3.0.2 The playbook's stop/start recovery replays phase 3 and can confirm a rotate on a single out-of-range observation

Technical Details

`RangePolicy.minOorCycles` is the number of "consecutive weekly phases OOR before rotate fires" (default 2, `ILynxLpAllocator`). `_evaluateRotate()` enforces it by counting calls: every call that finds the TWAP outside the band increments `oorStreak`, and nothing checks how much time has passed since the previous increment. The weekly spacing therefore depends on phase 3 running once per cycle.

The documented recovery breaks that assumption:

- After a consumed 328 abort, the playbook prescribes `stopWeeklyLoop()` followed by `startWeeklyLoop()` (`docs/keeper-playbook.md`, "Hedera 328", step 2).
- The ledger lists phases 4 and 5 as able to 328 on their own (`docs/audits/keeper-lp-review-2026.md`).
- `stopWeeklyLoop()` resets `opsPhase` to 0, and `startWeeklyLoop()` schedules phase 0 one `phaseGapSec` later.

Phases 0 to 3 therefore replay within minutes of the phase 3 that already ran in that cycle. The same happens when the loop is stopped for a strategy ownership round trip and restarted within the same cycle.

On the replayed phase 3, every target that is still out of range gets a second streak increment from a 30-minute TWAP window that overlaps the first one almost entirely. With the default `minOorCycles = 2`, a target whose first out-of-range observation came minutes earlier reaches the threshold. Gate 6b compares two almost identical ticks and only resets the streak if the TWAP moved back in between, so an unchanged or further TWAP confirms the rotate.

The internal audit described this consequence for a premature `retrySchedule()` (internal finding #6: "the streak is counted twice on overlapping TWAP windows, so a premature rotate is possible"). Its fix only changed the delay `retrySchedule()` uses, so the prescribed stop/start path still replays phase 3 within the same cycle.

Impact

Low. A rotate can be confirmed from a single out-of-range observation when the loop is restarted after phase 3. That happens when operators follow the documented recovery after a 328 in phase 4 or 5, or restart the loop after an ownership round trip. In that case `minOorCycles` no longer protects against rotating on a transient move, so the rotate crystallizes impermanent loss and swap costs.

PoC (if any)

Scenario, not executed:

1. Target A has `minOorCycles = 2` and no recent rotate. In week N its TWAP leaves the band, and phase 3 records `oorStreak[A] = 1`.
2. Phase 5 of week N aborts with 328, which leaves a stale `pendingSchedule`.
3. Following the playbook, the operator runs `stopWeeklyLoop()` and `startWeeklyLoop()`. Phases 0 to 3 replay a few minutes later.
4. Phase 3 evaluates A again. The streak reaches 2, and Gate 6b sees `currDist >= prevDist` because the TWAP has barely moved, so the rotate executes in week N.

Recommendation

Make the confirmation time-based. Record when a target's streak last counted an out-of-range observation, and ignore observations closer than about one `weeklyIntervalSec` to it.

Alternatively, record when `_rangeOps()` last ran and skip the rotate attempt when a restart replays phase 3 within the same interval.

Developer Response

Fixed in commit 93a62a2, together with #20.

3.0.3 Drain proceeds are not pulled back once the LP strategy's principal reaches zero

Technical Details

`_pullLpRecoveredWhbar()` pulls the LP strategy's idle WHBAR back to the vault after a drain, but returns early when the strategy has no recorded principal:

```
1 if (manager.principalOf(lpStrategy) == 0) return;
```

The manager tracks one aggregate principal for all LP positions, and every pull reduces it by the WHBAR received, capped at the remaining basis. A profitable wind-down, in which earlier drains return more than their share of basis, can therefore bring principal to zero while other positions are still open. From then on, the value of those positions is unrealized gain, which cost-basis accounting keeps out of NAV until a pull realizes it.

The guard blocks that realization. When governance later disables and drains another position, its proceeds stay on the strategy instead of returning to the vault, and phase 5 may redeploy them into the remaining enabled targets. The guard is not needed for accounting. With zero basis, `StrategyManager.pull()` reduces no principal, and `_bookPullRealization()` reports the whole receipt as gain, which then streams into NAV through profit locking.

This was filed during the previous fix review (yAudit/linxify-report#36) and closed without changes. The code is unchanged in this delta.

Impact

Low. Realized LP value stays outside NAV and redemption liquidity until new principal is pushed to the strategy and a later drain triggers a pull that recovers it. Shares minted in the meantime are priced against understated assets and share in the eventual recovery. Reaching zero principal with positions still open requires a profitable wind-down, so the likelihood is low.

Recommendation

Remove the zero-principal guard so that every successful drain returns its proceeds to the vault. `StrategyManager.pull()` already books receipts above the remaining basis as gain.

Developer Response

Fixed in commit 6377792.

3.0.4 A failed drain rolls back the same phase's buffer refill

Technical Details

Phase 2 runs the buffer refill and the inactive-LP drain in the same body call (L748-L753):

```
bool refilled = refillBuffer();
bool drained = reconcileInactiveLp();
if (drained) _pullLpRecoveredWhbar();
```

`refillBuffer()` pulls staking principal into the vault's idle buffer. `reconcileInactiveLp()` then calls `lpAllocator.drainPosition()`, and `_pullLpRecoveredWhbar()` calls `manager.pull()`, both without a `try/catch`. A revert in either one unwinds the whole `executeWeeklyOpsBody()` call, including the refill that already succeeded. `_runWeeklyOpsBodyBestEffort()` catches the failure and emits `WeeklyOpsBodyFailed`, and the router moves on to phase 3. The buffer is therefore not refilled until the next cycle.

The drain is market-sensitive:

- `_decreaseDisabledPosition()` prices its minimums through `_trustedSqrtPriceX96()`. That call reverts when the TWAP is unavailable or spot deviates from it by more than `maxSpotDeviationBps`, and the NPM then checks the TWAP-priced minimums against spot.
- The full-balance swap fails as described in #5.
- `manager.pull()` reverts `ImpairmentValuationFailed` when the LP strategy cannot be valued.

When the failure persists, as with the donation in #5, the refill is lost every week.

A broader version of this issue, filed against the single-call weekly body during the previous fix review (yAudit/linxify-report#35), was closed without changes. The phase split in this delta isolates the other stages but leaves the refill and the drain in one call.

Impact

Low. A drain failure in phase 2 costs the vault that week's buffer refill. Redemptions beyond the remaining idle WHBAR are then delayed even though staking principal is available. A persistent failure suppresses the refill every week. No funds are lost, and the router owner can call `refillBuffer()` directly.

Recommendation

Isolate the drain and the recovery pull from the refill. For example, catch the drain inside `reconcileInactiveLp()` and the pull inside `_pullLpRecoveredWhbar()`, emitting a failure event:

```
-            lpAllocator.drainPosition(sn, deadline);
-            emit InactiveLpDrained(sn);
-            return true;
+            try lpAllocator.drainPosition(sn, deadline) {
```

```
+                emit InactiveLpDrained(sn);
+                return true;
+            } catch (bytes memory err) {
+                emit InactiveLpDrainFailed(sn, err);
+                return false;
+            }
```

Alternatively, run the refill in its own phase.

Developer Response

Fixed in commits 79d4d3d, 619c5ee, together with #19. The drain now runs in its own phase.

3.0.5 A pair-token donation can permanently block the drain of a disabled LP position

Technical Details

`drainPosition()` removes all liquidity from a disabled position. It then swaps the strategy's entire balance of the pair token to WHBAR through `_swapPairBalanceToWhbar()`. The `minOut` comes from `_minOut()`: a linear TWAP quote minus `swapSlippageBps` (default 1%), which ignores the pool fee and price impact. The strategy also caps execution at a TWAP-derived price limit (`harvestSwapSlippageBps`, 100 bps on sqrt). A sale large enough to move the pool past that margin fills short of `minOut` and reverts.

Anyone can transfer the pair token to the strategy. A donation sized to push the sale past the margin makes every drain of that position revert. The donation is not undone by the reverted drain, and nothing else converts it: the harvest swap page only sells credited fees, and the allocator has no generic swap path (see #14). `reconcileInactiveLp()` retries the first disabled position above dust every week, without a `try/catch`. That position is therefore never drained, and the disabled positions after it in `lpPositionSerials` are never reached. Each failure also rolls back the phase-2 buffer refill (see #6).

The same revert occurs without a donation when the position is large relative to the pool's in-range depth. It also occurs on a 1% fee tier with the default slippage (see #12).

The full-balance sweep was reported in the previous engagement (yAudit/linxify-report#6) for its imprecise scope. The client kept the sweep by design and documented it. That issue did not cover this denial of service. The same donation pattern in the rotate swap is part of #1.

Impact

Low. By sacrificing pair tokens, an attacker can keep a governance-disabled position, and every disabled position after it, out of the automated drain indefinitely. Through #6, the same failure also suppresses the weekly buffer refill. The donation becomes protocol value, and the position remains recoverable through `emergencyUnwind()` and a strategy ownership round trip, so no funds are lost.

Recommendation

Swap only the pair balance that the decrease produced. `drainPosition()` already records `pairBalBefore`, so the position-scoped amount is the balance after the decrease minus `pairBalBefore`. Sweep pre-existing balances through a separate owner-gated, size-bounded swap, which would also give the unwound residual in #14 a conversion path. If the full sweep must stay by design, cap the amount swept beyond the position's own proceeds so that a donation cannot make the sale unexecutable, and include the pool fee in `minOut` (see #12).

Developer Response

Fixed in commits 239aa3d, 8197b10.

3.0.6 The allocator's governance powers cannot be exercised through `LynxGovernor`

Technical Details

`LynxLpAllocator` gives its `governance` address (the DAO timelock) several powers:

- `setEnabledFlags()` (`onlyGovernance`), the DAO's control over which targets receive capital (`docs/governance-dao-mapping.md`), and therefore its trigger for the phase-2 drain of a deselected position. The owner can set the same flags through `setTargets()`, which writes each target's `enabled` field.
- `setOpsRangeFrozen(false)`, described as "the DAO escape hatch so the owner cannot permanently disable range operations" (L819-L837).
- `emergencyWiden()` and `emergencyUnwind()`, owner OR governance.

With the Governor as its only proposer, the timelock only executes what `LynxGovernor._propose()` accepts. That function rejects any action with a non-zero `value` and only allows, on the allocator, the `setEnabledFlags()` selector, and only when the target equals the Governor's `immutable lpAllocator`:

```
if (target == lpAllocator) {
    return selector == SET_ENABLED_FLAGS_SELECTOR;
}
```

This has two consequences:

1. **The emergency and unfreeze powers are unreachable by the DAO.** Proposals for `emergencyWiden()`, `emergencyUnwind()` or `setOpsRangeFrozen(false)` revert `ProposalActionNotAllowed`. The "DAO escape hatch" described in the NatSpec and playbook does not exist in the deployed governance stack. Even if these selectors were allowlisted, the governance latency would make them unsuitable for emergencies. On mainnet it is a 1-day voting delay, a 7-day voting period and a timelock delay of at least 48 hours. In the meantime a rotate can replace the targeted serial (internal #18).
2. **After the Layer C rollout, the DAO cannot even call `setEnabledFlags()`.** The Governor pins `lpAllocator` as `immutable`, and on mainnet it returns the live allocator

`0x…a52d05`. The rollout deploys a new allocator, because `lpStrategy` is immutable, and does not redeploy governance (one-pager, "Rollout for B (and B+C)"). Every proposal targeting the new allocator is therefore rejected. The DAO loses its only allocator power, while the owner keeps full control of target selection through `setTargets()`.

Even where the DAO could act, its role is not independent:

- `setGovernance()` is `onlyOwner` and takes effect immediately.
- The owner can override any DAO toggle at any time with `setTargets()`, and can disable automation through `setRangeRotateEnabled(false)`, which governance cannot reverse.
- The threat model assigns the timelock's `DEFAULT_ADMIN_ROLE` and the protocol `owner` to the same admin multisig (`docs/governance-threat-model.md`). On mainnet, the live allocator's `owner()` (`0xb19907cD…04bD`) holds `DEFAULT_ADMIN_ROLE` on the timelock. `LynxTimelock._execute()` only blocks role changes executed through the timelock itself. The admin can therefore grant itself `PROPOSER_ROLE` directly and schedule any call, including the emergency ops with a non-zero value. The `governance` branches are reachable by the admin, not by token holders.

Impact

Low. No funds are at risk, but the DAO controls advertised for the range manager are not available through governance. The unfreeze escape hatch and the emergency ops never were, and after the rollout target enable/disable is not either. All of them remain with the owner, who as timelock admin can also act through the `governance` address. The separation of powers that the NatSpec and playbook describe therefore does not exist in the deployed stack.

Recommendation

Before the Layer C rollout, deploy a Governor whose allowlist points at the new allocator, or make the allocator address in the Governor updatable through governance itself. Decide which allocator powers the DAO should hold and add them to the allowlist, at least `setOpsRangeFrozen` and, if intended, `emergencyUnwind` / `emergencyWiden`. If the DAO is not meant to hold them, remove the `governance` branches and correct the NatSpec and playbook. Consider letting only the current `governance` change `governance`. If the DAO is meant to constrain the owner, give the timelock's `DEFAULT_ADMIN_ROLE` to a different party than the protocol owner, or renounce it once the roles are final.

Developer Response

Fixed in commits 511a01b, 2354f6d. The allocator is now owner-only, and the governor no longer allows any allocator selector.

3.0.7 `emergencyWiden()` still depends on a ready TWAP close to spot, so it fails in the conditions it is advertised for

Technical Details

`emergencyWiden()` is documented as the break-glass path that "must always work", for "violent market moves, unready TWAP, or frozen automated range ops", with "No TWAP, no swap, no rotate gates" (NatSpec at L546-L549). It deliberately centres on the raw `slot0()` price so it can proceed when the regular spot-deviation guard would fail (L637-L650).

However, whenever the requested range contains spot, which is the normal goal of a widen, `_swapWidenSurplus()` runs the ratio swap. An old out-of-range position yields a single-token balance. The widen also invests the strategy's whole WHBAR balance, so the WHBAR leg may not be empty, but the two legs are almost never in the new range's ratio, so in practice re-ranging around spot almost always needs a non-zero swap. The swap was added by the fix for internal finding #12, and the NatSpec still describes the earlier swap-free path. Contrary to the NatSpec, the code therefore performs a swap, and that swap is TWAP-gated twice:

1. In the allocator, `LpRangeMath.quoteSurplusSwap()` → `SlippageQuotes.minOutFromTrustedPool()` → `trustedSqrtPriceX96()` reverts with `TwapNotReady` when cardinality is insufficient. It also reverts on any `observe()` failure, and with `SpotTwapDeviationExceeded` when spot deviates from the TWAP by more than `maxSpotDeviationBps` (default 500 bps on sqrt, about 10% on price).
2. In the strategy, `_swap()` → `_deriveSwapPriceLimit()` repeats `trustedSqrtPriceX96()` with the strategy's own TWAP configuration whenever no explicit `swapSqrtPriceLimitX96` override is set. Relaxing only the allocator's settings is therefore not enough.

The strategy's price limit is also the tighter bound. `_deriveSwapPriceLimit()` places `sqrtPriceLimitX96` at the strategy TWAP ± `harvestSwapSlippageBps` (default 100 bps on sqrt, about 2% on price). Re-ranging an out-of-range position around spot sells the leg the market moved the position into, so the swap trades in the direction of the move, and after a fast move spot is already past the TWAP in that direction. Once spot is about 2% past the TWAP, the limit is behind spot and the pool rejects the swap (`SPL`). Closer to the limit, the swap fills only up to it and fails `minOut`. In a violent move, the widen therefore reverts long before the 500 bps deviation bound is reached.

Any of these reverts rolls back the whole widen, including the preceding `rotateClosePosition()`. The allocator exposes no caller-selected emergency slippage bounds. The owner's only workaround is to relax `maxSpotDeviationBps` through the allocator's `setTwapConfig()`. Neither the strategy's TWAP configuration nor its `setSwapSqrtPriceLimitX96()` override, which would bypass the derived limit, has a passthrough on the allocator, so the second check and the limit cannot be relaxed without a strategy ownership handover. When the widen does not need a swap, these checks are skipped and it works as documented.

Impact

Low. The emergency re-range is unavailable in exactly the market and oracle conditions it was built for, contrary to its NatSpec. After a violent move, it only becomes executable once the 30-minute TWAP is within about 2% of spot. `emergencyUnwind()` does no swap and remains available while the harvest lock is clear, so funds can still be exited, just not re-ranged.

Recommendation

Give the emergency path explicit, caller-provided bounds, such as `swapMinOut` and `sqrtPriceLimitX96` or a `skipSwap` flag, and enforce them through both the allocator and a dedicated owner-only strategy swap entry that does not derive a TWAP limit. Do not weaken the regular rotate path. If oracle-independent widening is intentionally unsupported, document the restriction and direct emergency callers to `emergencyUnwind()` when the TWAP is unavailable or too far from spot.

Developer Response

Fixed in commits 8ca49b5, 66eb6dd.

3.0.8 Permissionless `retrySchedule()` on phase 0 mid-pass postpones the cycle by a week and keeps the harvest lock

Technical Details

After the fix for internal finding #6, `retrySchedule()` picks the delay from `opsPhase` alone:

```
_scheduleNext(opsPhase == 0 ? weeklyIntervalSec : phaseGapSec);
```

The assumption is that `opsPhase == 0` always means "the 5→0 wrap just failed to schedule". With paged harvest, phase 0 has a second meaning: the router stays on phase 0 between harvest pages, and after a catchable phase-0 body failure, until the liveness caps trip (L351-L355):

```
if (from == 0 && _shouldStayOnPhase0(bodyOk)) {
    if (loopActive) {
        _scheduleNext(phaseGapSec);   // may emit ScheduleFailed and leave pendingSchedule == 0
    }
    return;
}
```

If this intra-cycle `_scheduleNext()` fails (HSS capacity, treasury, transient error), `pendingSchedule == 0` and the page or failure state is preserved. `opsPhase` alone cannot tell this continuation apart from a completed-cycle wrap. The next `retrySchedule()`, which anyone can call, including an ordinary recovery bot, schedules the next page `weeklyIntervalSec` (604,800 s) later instead of `phaseGapSec` (60 s). Once that schedule exists, further retries revert `PendingScheduleExists` and cannot bring execution forward. The keeper playbook even prescribes this retry: its recovery steps state that with `loopActive && pendingSchedule == 0` anyone can call `retrySchedule()` to run the current phase soon (`docs/keeper-playbook.md`, "Hedera 328").

For that week:

- When a pass is pending, `harvestPassActive` stays `true` on the LP strategy, so every path guarded by `_requireNoHarvestPass()` reverts: `createLPPosition()`, `closeLPPosition()`, `rotateClosePosition()`, `swapToWhbar()` and `swapFromWhbar()`. This blocks the governance/owner break-glass paths `emergencyUnwind()` and

`emergencyWiden()`, as well as drain swaps and manual swaps. If the router stayed on phase 0 only because the first page of a pass failed, no lock is held and only the delay below applies.
- The rest of the current cycle is delayed by a full week: staking harvest, phase 2 `refillBuffer()` (the redeem buffer is not refilled from staking while withdrawals drain vault idle), range ops, surplus push and LP deploy.

Impact

Low. No funds are at risk, but a one-off HSS failure combined with a routine retry freezes LP emergency operations and buffer refill for a week. The router owner can recover through manual `weeklyOps()` calls or `stopWeeklyLoop()` + `startWeeklyLoop()`.

Recommendation

Remember the delay intended by the failed `_scheduleNext()` call and reuse it in `retrySchedule()`. Alternatively, treat phase 0 as a continuation whenever the cycle is mid-flight:

```
-         _scheduleNext(opsPhase == 0 ? weeklyIntervalSec : phaseGapSec);
+         bool midPhase0 = opsPhase == 0
+             && (ILpHarvestPaged(lpStrategy).harvestPassPending()
+                 || lpHarvestPagesThisCycle != 0
+                 || consecutivePhase0Failures != 0);
+         _scheduleNext(opsPhase == 0 && !midPhase0 ? weeklyIntervalSec :
phaseGapSec);
```

Developer Response

Fixed in commit 1a35e8b.

3.1 Gas Savings Findings

None.

3.2 Informational Findings

3.2.1 Range policies near the maximum width cannot be built when the TWAP is far from tick 0

Technical Details

`setRangePolicy()` accepts `halfWidthTicks` up to `MAX_HALF_WIDTH_TICKS = 800_000` and `|centerOffsetTicks|` up to 100, without reference to the pool's price. At rotate time, `planRotate()` centres the range on the TWAP. It reverts with `InvalidRange` when an edge falls outside `[MIN_TICK, MAX_TICK]`.

A policy with half-width `W` therefore cannot be built when `|TWAP tick|` exceeds about `887,272 - W - |offset|`. At the maximum width, this means about ±87,000 ticks. Each attempt reverts and consumes the week's rotate attempt (see #1).

Impact

Informational. Nothing is affected at current prices. On 2026-10-12, the catalog pools sat at ticks 69,930 (USDC), 43,737, 3,604, 25,866 and −20,424. For USDC, the pool with the smallest margin, a policy at the maximum width would fail only after a price move of about 5.5×. The issue needs both a near-maximum width, which is practically full range, and a pool far from tick 0.

Recommendation

When an edge crosses the tick bound, shift the range inward, keeping the width and keeping the TWAP inside the range. Otherwise, skip the target with a typed reason and do not consume the week's rotate attempt.

Developer Response

Fixed in commit 65e0bab.

3.2.2 `_clearPendingSchedule()` ignores the response code of `deleteSchedule()`

Technical Details

`_clearPendingSchedule()` calls `HssOps.deleteSchedule(prev)` and discards the returned response code. It then clears `pendingSchedule` regardless of the result. Two functions use it: `stopWeeklyLoop()`, and `_scheduleNext()` before it creates the next schedule.

If HSS rejects the delete while the schedule is still pending, the router reports the loop as stopped, but the schedule stays armed:

- **After `stopWeeklyLoop()`:** the schedule later calls `weeklyOps()`. The call passes `onlyOpsExecutor`, because the scheduled call executes as the router itself. One phase body runs. The schedule does not reschedule itself, because `loopActive` is false.
- **In `_scheduleNext()`, with a schedule still pending** (for example, when the owner calls `weeklyOps()` manually): two schedules are live while the router tracks one.

On mainnet, the router's schedules use the router itself (`ContractID 0.0.10826535`) as admin key, so hiero issue #22862, which concerns `DELEGABLE_CONTRACT_ID` keys, does not apply. On the normal path the delete returns `SUCCESS`; for example, schedule `0.0.10907633` at `1791299196.064620525`. The four `stopWeeklyLoop()` calls on mainnet returned `INVALID_SCHEDULE_ID`, but each one targeted a schedule that had already executed. For example, schedule `0.0.10850584` executed at `1788878000` and the stop came at `1788878869`. Nothing stayed armed. A rejected delete on a still-pending schedule requires a race with the schedule's own execution.

Impact

Informational. The case is rare, and the result is one extra phase execution.

Recommendation

Check the response code. Treat `SUCCESS`, and the codes meaning that the schedule no longer exists (already executed, already deleted, or `INVALID_SCHEDULE_ID`), as cleared. For any other code, emit an event and keep `pendingSchedule`. Do not revert on every non-`SUCCESS` code: the playbook's stop/start recovery after a 328 stops the loop on a schedule that has already executed, and that delete returns `INVALID_SCHEDULE_ID`.

Developer Response

Fixed in commits c797493, 2354f6d, 426f684, 3470f5f.

3.2.3 A replacement mint deploys credited pair-token fees and erases their credit, so the fees skip the profit lock

Technical Details

`harvestFeeCredit[pairToken]` marks pair-token fees that the harvest has collected but not yet converted. `_tryIdlePairTokenWhbarValue()` excludes them from `impairmentValue()`. They reach NAV only after the swap page converts them and the WHBAR is reported as locked profit. When a harvest swap fails, the `catch` branch keeps both the tokens and their credit (L929-L934).

A rotate or widen before the next successful conversion breaks this:

1. `_mintReplacement()` deploys the strategy's full WHBAR and pair-token balances, including the credited fees.
2. `createLPPosition()` clamps the credit to the remaining idle balance (L583-L584), so the credit drops to zero or dust.
3. The fees now sit inside the new NFT, and `impairmentValue()` counts them immediately.

Impact

Informational. This matters only while the LP strategy is impaired. In that case, the next impairment sync reduces the impairment by the fee amount, and NAV rises at once instead of over the 7-day unlock. A depositor who entered before the sync captures part of the increase. Three conditions are needed: a failed harvest swap, a rotate or widen before a successful retry, and an impaired strategy. The amount is bounded by the stuck credit.

This is the same class as internal findings #13 (pair fees entering NAV unstreamed) and #15 (the rotate sweeps all balances). Today #22 makes both paths revert.

Recommendation

Leave credited pair tokens out of the replacement mint: subtract `harvestFeeCredit[pairToken]` from the pair balance in `_mintReplacement()`, so the next swap page converts them. Alternatively, report the credit as profit when it is deployed instead of clamping it away.

Developer Response

Fixed in commit d234a53.

3.2.4 Harvest swap backlog can outgrow one swap

Technical Details

`_swapHarvestPage()` sells each pair token's full `min(balance, harvestFeeCredit[pairToken])` in a single `exactInputSingle()` call (L902-L906), with no per-swap cap. Its minimum output, from `_quoteMinWhbarOut()`, is linear in the amount, while the pool's actual output falls behind it as the size grows because of the pool fee and price impact.

When a swap fails, the catch branch only clamps the credit to the token balance (L930-L934). Each later collect adds to the credit (`_creditHarvestFees()`), so the credit only grows. Once it exceeds a pool-depth threshold, every later swap fails at the same pool state. At the default 100 bps that threshold is about 0.71% of the in-range pair reserve at fee 3000 and about 0.86% at fee 1500. The retained tokens stay excluded from NAV until they are swapped.

A Hardhat test using the production strategy, manager and vault, with a router built on SaucerSwap V2's `SwapMath.computeSwapStep`, reproduces it:

- Configuration: WHBAR/PACK, fee 3000, full range, and a chosen 150 PACK of fees per week.
- After two failed weeks, every swap from week 3 to week 10 fails at the original price, leaving 1,500 PACK (about 148.7 WHBAR) unconverted.
- A control run without the initial failures converts every week.

This differs from #12, where a fee-blind minimum output makes 1% tier swaps fail every week. Here the cause is the growing credit, and a pool-fee haircut alone does not remove it.

Impact

Informational. Nothing is lost. The pair-side fees of one target stay on the strategy, excluded from NAV and not streamed as profit, until in-range liquidity grows or the owner raises `harvestSwapSlippageBps`. As #12 notes, raising it requires a strategy ownership handover from the allocator. In testing, raising the tolerance to 200 bps cleared the full backlog on the next pass.

It is unlikely on the current setup. On the launch USDC/WHBAR target Lynx holds about 0.3% of the pool, and reaching the threshold would take roughly 600 to 700 consecutive failed

weeks. It becomes realistic only if Lynx holds a large share (10% to 33% or more) of a thin fee-1500 or fee-3000 pool.

Recommendation

Add an owner-set, per-pair-token cap on each harvest swap, at least the dust threshold. Swap at most one capped chunk per token per pass, and leave the remaining credit for later passes. For the credit debit after a partial fill, apply the fix recommended in #11.

Developer Response

Fixed in commit 4e975c3.

3.2.5 Rotates keep deploying idle capital while the vault is paused

Technical Details

The vault guardian can pause the vault in response to an incident. Since the fix for yAudit/linxify-report#19 in the previous engagement, the pause also halts deployment: while `vault.paused()` is true, `pushAndDeploy()`, `deployLpIdle()` and `allocateSurplus()` return early (L434, L445, L536), and `StrategyManager._precheckPush()` reverts. During a pause, WHBAR already pushed to the LP strategy therefore stays idle.

The rotate added in Layer C does not respect the pause:

- `_rangeOps()` gates only on the allocator, `opsRangeFrozen` and the harvest lock.
- `rebalanceOne()` adds no pause check.

The replacement mint deploys the strategy's full WHBAR and pair-token balances (internal #15). A rotate during a pause therefore does more than re-range the rotated position: it also invests the idle WHBAR that `deployLpIdle()` is holding back. Only the owner can stop this, through `setOpsRangeFrozen(true)`; the guardian cannot.

Impact

Informational. If an incident leads the guardian to pause the vault, the keeper can still move idle capital into a new LP position until the owner also freezes range ops. The rotate's TWAP and slippage checks still apply.

Recommendation

Skip `rebalanceOne()` in `_rangeOps()` while the vault is paused, keeping the divert computation, or let the guardian freeze range ops.

Developer Response

Fixed in commit 6dcd5b4.

3.2.6 Cardinality is treated as TWAP readiness, so a pool with short observation history can block vault settlement

Technical Details

`TwapQuotes.tryTwapSqrtPriceX96()` only checks `observationCardinality >= minObservationCardinality` (default 2) before calling `observe([twapWindowSec, 0])` with a 1800 s window. Cardinality does not prove that the ring buffer actually covers the window. SaucerSwap V2 pools overwrite the oldest observation once capacity is reached, and `observe()` reverts (`OLD`) if the oldest retained sample is younger than the requested window. Nothing on-chain enforces more than `minObservationCardinality = 2`, and the only provisioning script in the repository (`hardhat/testing/warm-saucerswap-twap.ts`, testnet-only) grows pools to exactly 2. At cardinality 2, two observations written within 30 minutes are enough. The oracle writes at most one observation per block timestamp, on tick-changing swaps and in-range liquidity changes.

In `SaucerSwapLiquidityStrategy`, both `_tryTotalLpWhbarValue(true)` and `_tryIdlePairTokenWhbarValue(true)` return `(false, 0)` when any relevant pool fails. This covers every active position with non-zero liquidity (`_tryTotalLpWhbarValue()` skips the TWAP read when `pos.liquidity == 0`, L758). It also covers every tracked pair token with a configured pool and a non-zero balance net of `harvestFeeCredit`, including the idle pair token left behind by a drained position. The dust exemption only applies to pair tokens without a pool (L809-L813). `impairmentValue()` then reports an incomplete valuation, and `StrategyManager.syncValuations()` reverts `ImpairmentValuationFailed`. Every vault deposit, mint, withdraw and redeem runs this sync, so all of them revert while the condition lasts.

`tryTwapSqrtPriceX96()` already catches the `observe()` failure and fails closed, so the code behaves as designed. The problem is that nothing provisions or checks enough observation history. With a small buffer, ordinary pool activity can trigger the outage, and an attacker can sustain it with small tick-changing swaps. In the delta, the same assumption also drives `checkRangeStatus()` (permanent OOR divert) and the `rebalanceOne()` revert described in #1.

The current mainnet catalog is not in this state. On 2026-10-08 the five pools in `hardhat/config/lp-pools.json` reported the following observation cardinalities: 1200 for USDC, DOVU and SAUCE, 1000 growing to 1200 for HBARX, and 500 for PACK. All of them served `observe([1800, 0])`. The shortest buffer, PACK's, reached back about 9.6 days. Exhausting any of them within 30 minutes would take 500 to 1200 writes at distinct block timestamps in every window, for as long as the attacker wants the outage to last. The assumption is still neither enforced nor checked. A pool whose buffer was never grown stays at cardinality 1 and fails `twapPriceReady()` outright, which is why the testnet pools needed the warm-up script. A pool grown only to 2 fails under ordinary trading.

Impact

Informational. With the current catalog, normal activity cannot cause the outage, and forcing it would take a sustained, costly stream of swaps. If a pool with a small observation buffer is added, vault deposits and redemptions, including redemptions from the idle buffer, are suspended while it cannot serve the 30-minute observation. No funds are extracted, and anyone can fix it permanently by calling `increaseObservationCardinalityNext()` on the pool. The fix

is not immediate: the new slots only fill as observations are written, and the ring buffer must then cover the 30-minute window.

Recommendation

Raise the observation cardinality of every valuation-relevant pool well above the number of observations writable within `twapWindowSec`, and set `minObservationCardinality` on the strategy and allocator to that level. Before funding a position or registering a pool, check that `observe([twapWindowSec, 0])` succeeds, and use that same check, rather than cardinality alone, in `twapPriceReady()` / `_isTwapReady()`. Keep valuation fail-closed and do not fall back to spot.

Developer Response

Fixed in commits 644d444, fa530e6, 9788149.

3.2.7 The post-unwind recovery in the NatSpec and keeper playbook omits the required strategy ownership handover

Technical Details

`emergencyUnwind()` burns the NFT through `closeLPPosition()`, sets `targets[idx].positionSN = 0`, and leaves both legs idle on the strategy. That the target then needs a new NFT is known and documented. Internal finding #16 accepts that it stays enabled. The playbook ("Recovery path after unwind", `docs/keeper-playbook.md`) and `taskplane-tasks/CONTEXT.md` ("Bootstrapping flow") describe re-entry as: the owner "mints a fresh LP position on `SaucerSwapLiquidityStrategy`" and re-points the target with `setTargets()`.

The allocator cannot perform that step. The strategy's `createLPPosition()` is `onlyOwner`, and the strategy's owner is the allocator, which exposes no `createLPPosition()` passthrough. `deployLpIdle()` skips `positionSN == 0` (L1052) and only calls `increaseLPPosition()`. `rebalanceOne()` and `emergencyWiden()` both need a live serial, and `setTargets()` → `_validateTarget()` requires an existing active NFT. The `emergencyUnwind()` NatSpec, which says "re-entry requires a fresh `setRangePolicy()` + `deployLpIdle()`" (L1079), is also inaccurate.

The allocator cannot convert the non-WHBAR leg left by the unwind either:

- `drainPosition()` needs a non-zero serial (`_resolvePairToken()` reverts `PositionNotFound(0)`), and the burned serial is no longer in `lpPositionSerials`, so `reconcileInactiveLp()` never reaches it.
- The harvest swap page caps each swap at `harvestFeeCredit`, which only grows from fee-only collects, so the unwound principal is never sold.
- Rotates of other targets only swap their own `pairToken`.
- The allocator has no generic `swapToWhbar()` passthrough.

Both the re-entry and the residual conversion therefore require an Ownable2Step round trip of the LP strategy: `transferLpStrategyOwnership()` to the operator, manual operations, transfer

back, then `acceptLpStrategyOwnership()`. The repository already supports this round trip as a general recovery procedure (`ops:mainnet:lp-reclaim-strategy` and `ops:mainnet:lp-return-strategy`). The keeper playbook documents it for a configuration change that was applied on mainnet on 30 Aug 2026. The unwind documentation does not refer to it. During the round trip the allocator is not the owner, so every keeper path that reaches an `onlyOwner` strategy function through the allocator (`deployLpIdle()`, `drainPosition()`, `rebalanceOne()`) reverts. The playbook asks for the loop to stay stopped until ownership is back.

Impact

Informational. No funds are lost. The residual stays valued in `impairmentValue()` through the tracked pair, and the existing reclaim and return scripts make both re-entry and residual conversion possible. The unwind NatSpec and the playbook's recovery section, however, describe a re-entry the allocator cannot perform. They also omit the ownership round trip, which pauses automated LP operations while it lasts.

Recommendation

At minimum, correct the NatSpec on `emergencyUnwind()` / `ILynxLpAllocator.emergencyUnwind()` and point the playbook's "Recovery path after unwind" to the reclaim/return procedure. To avoid the round trip, add an owner-only re-entry function that mints a new position for a target with `positionSN == 0`, reusing `_swapSurplus()` + `_mintPlanned()` with caller ticks (the same as `emergencyWiden()` minus the close). Add an owner-gated, TWAP-protected `swapPairResidualToWhbar(pairToken, amountIn, minOut)` passthrough as well.

Developer Response

Fixed in commits bd9e39c, 5b4f45a, 81d1d8a.

3.2.8 Settlement valuation relies on a raw pool TWAP and counts unsolicited pair-token balances

Technical Details

`impairmentValue()` is used by `StrategyManager.syncValuations()` before every vault deposit, mint, withdraw and redeem. It values the LP strategy's positions and the full directly held balance of every tracked pair token, minus `harvestFeeCredit`, at the pool's 30-minute TWAP. This happens in `_tryIdlePairTokenWhbarValue()` and `_tryTotalLpWhbarValue()`. Two design properties follow:

1. The valuation does not distinguish strategy-generated balances from donations. Anyone can transfer a tracked pair token to the strategy, and it is valued immediately.
2. The TWAP is read through `tryTwapSqrtPriceX96()` without a spot-deviation bound or any independent oracle. Omitting the spot/TWAP veto is intentional: it is the fix for a previous finding (M-06 of the earlier engagement, "Spot deviation can veto impairment synchronization").

The settlement price is therefore only as robust as the 30-minute TWAP of each configured pool. A shareholder who could keep a shallow pool's TWAP pushed up would inflate the valuation, release reversible impairment at the next sync, and redeem against the restored accounted principal. When the TWAP normalizes, the remaining shareholders absorb the difference. A donation alone is not an accounting error: the donor gives up real value, and the impairment it releases is legitimate. It only amplifies such a manipulation by the difference between the manipulated and the real price.

We have not demonstrated that this manipulation is profitable for any pool in the catalog. That depends on the cost of moving the TWAP, the exploitable impairment, the attacker's share and the redeemable liquidity. We therefore report it as a design observation. It differs from M-01 of the previous engagement, which covered *unvalued* pair balances causing false loss booking, the opposite case.

Impact

Informational. Settlement safety depends on the depth and observation history of every configured pool. The upside is bounded by the strategy's outstanding impairment and reduced by the exit fee.

Recommendation

Restrict the catalog to pools whose TWAP is costly to move over `twapWindowSec`, and size `twapWindowSec` and cardinality accordingly. Consider valuing idle pair tokens as at most `min(actualBalance, recognizedBalance)`, where `recognizedBalance` only changes through verified strategy operations, and capping how much impairment a single sync may release.

Developer Response

Fixed in commits ea82bc7, cb7af3b, 14e04d9, 4472608, 44c58e7.

3.2.9 Swap `minOut` quotes ignore the pool fee, so swaps on 1% pools fail in practice

Technical Details

`_quoteMinWhbarOut()` prices the input at the pool's `slot0()` spot price, without the pool fee, and applies only `harvestSwapSlippageBps` (default 100 = 1%):

```
uint256 quoted = _tokenWhbarValue(pairToken, pool, amountIn); // spot value, no fee
minOut = quoted * uint256(10_000 - bps) / 10_000;
```

The swap output is at most `quoted × (1 − poolFee)`, before any price impact. On a 1% pool (`poolFee = 10000`), the best possible output equals `minOut`, so the swap fails whenever price impact exceeds the margin left by rounding. With the default configuration, `_executeHarvestSwap()` therefore reverts on practically every non-trivial page. The revert is caught (`HarvestPairSwapFailed`) and counts against `maxHarvestSwaps`. On a 0.3% pool, only 0.7% is left for price impact. The catalog `hardhat/config/lp-pools.json` contains a 1% WHBAR/DOVU pool (currently `selected: false`).

The allocator's quotes have the same omission. `SlippageQuotes.minOutFromTrustedPool()` applies only `swapSlippageBps` (default 100) to `quoteExactInput()`, which is documented as "no fee haircut". It prices the `deployLpIdle()` swap and the drain swap through `_minOut()`. In the delta, it also prices the rotate and widen ratio swap through `LpRangeMath.quoteSurplusSwap()`. With a 1% target enabled and the default `swapSlippageBps`, these swaps fail in practice for the same reason:

- `deployLpIdle()` reverts at that target's swap whenever `swapBps > 0` (the catalog uses 5000). The deploy loop has no per-target isolation, so the phase-5 deploy fails for every target, not only the 1% one. Any other per-target revert, such as a transient spot/TWAP deviation on one pool, has the same effect for that week. The undeployed WHBAR stays on the strategy. As long as phase 4 keeps pushing surplus (it does not during a divert), the idle balance grows for every target until the next rotate sweeps it into a single position (internal #15), which also enlarges that rotate's swap (see #1).
- Every confirmed rotate of that target reverts. This is one of the persistent reverts that starve later targets in #1.
- A drain of that target reverts as well, so a deselected 1% position stays in the pool and the phase-2 buffer refill is rolled back with it (see #5 and #6).

Impact

Informational. With the default slippage settings, enabling a 1% pool has three effects. First, harvest does not convert its pair-token fees in practice: they stay credited and excluded from NAV, never stream as profit, and every page spends a swap attempt, and its child records, on a swap that cannot succeed. Second, the weekly LP deploy fails for all targets, so the LP sleeve's new capital stays idle. Third, the pool's own position can neither rotate nor be drained. The owner can restore the allocator paths by raising `swapSlippageBps` (up to 500 bps, for every pool at once). `harvestSwapSlippageBps` can only be changed through a strategy ownership handover, since the allocator has no passthrough for `setHarvestSwapConfig()`.

Recommendation

Include the pool fee in both quotes, `_quoteMinWhbarOut()` and `SlippageQuotes.minOutFromTrustedPool()`, for example `minOut = quoted × (1 − poolFee) × (1 − slippageBps)`. Alternatively, require the slippage settings to exceed the fee of every tracked pool. Switching the harvest quote to the TWAP price alone does not fix the omitted fee. Deploy each target in isolation, for example through a per-target self-call wrapped in `try/catch`, so that one failing target keeps its share for the next cycle instead of blocking the others.

Developer Response

Fixed in commits 50cc7fc, a7194b1, 1e4fb25.

3.2.10 A partially filled harvest swap erases the fee credit of tokens that were not sold

Technical Details

`_swapHarvestPage()` caps the requested swap input at `harvestFeeCredit[pairToken]`. After a successful call it subtracts the entire *requested* input from the credit, without measuring how many pair tokens actually left the strategy:

```
harvestFeeCredit[pairToken] -= balance;
_clampHarvestFeeCredit(pairToken);
```

The requested input is not guaranteed to be fully spent. `_swap()` passes a non-zero `sqrtPriceLimitX96`, either configured or derived from the TWAP minus `harvestSwapSlippageBps`. A SaucerSwap V2 pool stops at that limit even if input remains, and the router accepts the result whenever `amountOutMinimum` is met. Payment follows the pool's actual input delta (see the pool and router implementations). The price limit and the output minimum are distinct protections, so a near-complete partial fill can satisfy `minOut`. For example, spot may already sit close to the TWAP-derived limit when the page executes. The strategy has no full-consumption check.

When only part of the credited amount is sold, the credit still drops to zero while unsold fee tokens remain. `_clampHarvestFeeCredit()` can only lower credit, never restore it, and the swap cursor advances.

From then on:

- The residual has no credit, so later harvests never swap it as fees. It is only converted when a strategy-wide sweep absorbs it: `_swapSurplus()` and the replacement mint in a rotate or `emergencyWiden()`, or `drainPosition()`.
- `_tryIdlePairTokenWhbarValue()` no longer subtracts a credit for it, so it is valued as ordinary idle principal.

The residual is small. `minOut` is `slot0` value × (1 − `harvestSwapSlippageBps`) on the *whole* requested amount, while the output of the filled part is at most its spot value × (1 − pool fee). The sold fraction must therefore be at least (1 − s) / (1 − fee). With the default s = 1%, at most about 0.7% of the page's credited fees can remain unsold on a 0.3% pool. On a 1% pool a partial fill can essentially never pass `minOut` (see #12).

Impact

Informational. A small fraction of a page's fees, bounded by about `harvestSwapSlippageBps` minus the pool fee, can leave automated fee realization until a later rotate, widen or drain sweeps it. When the strategy is impaired, that residual raises settled NAV through impairment release instead of passing through `reportGain()` and profit locking. The funds stay in the strategy.

Recommendation

Measure the pair-token balance before and after `_executeHarvestSwap()`, debit the credit only by the input actually spent, and keep the unsold credited remainder available for a later page. Report the actual input in `HarvestPairSwapped`, and apply `_clampHarvestFeeCredit()` after the measured debit.

Developer Response

Fixed in commits acbb297, 2354f6d.

3.2.11 `emergencyWiden()` is not payable and depends on the allocator's leftover HBAR for the mint fee

Technical Details

`emergencyWiden()` forwards `lpStrategy.mintFeeHbar()` as `{value: w.mintFee}` to `createLPPosition()` from the allocator's own balance, but the function itself is not `payable`. The router only tops the allocator up to one mint fee during phase 3 (`_fundAllocatorMintFeeIfNeeded()`), and a keeper rotate in the same cycle consumes it. A widen executed afterwards fails with a bare low-level revert from the value transfer. Unlike the rotate path, it has no `InsufficientMintFee` pre-check, and the caller cannot attach HBAR to the call. This is the `emergencyWiden()` residual of internal finding #17, whose fix (PT-026) funds the allocator only for the keeper's rotate.

Impact

Informational. The break-glass call can fail for an operational reason that is unrelated to the market and gives no clear error.

Recommendation

Operationally, the caller can pre-fund the allocator in a separate transaction, since `receive()` is open to anyone. A DAO batch cannot do this: `LynxGovernor` rejects proposals with a non-zero `value` and does not allow `emergencyWiden()` (see #4). In code, make `emergencyWiden()` `payable` (and refund any excess), or add the same `InsufficientMintFee` pre-check as `_finishRotate()` and document that the allocator must be pre-funded before an emergency widen.

Developer Response

Fixed in commit 12719c9.

3.2.12 OOR divert flag is computed from pre-rotate state, so a successful rotate can still divert the week's surplus

Technical Details

In `_rangeOps()`, `checkRangeStatus(t.positionSN)` is evaluated and folded into `anyOor` *before* `rebalanceOne()` is called for the same target. If that target was out of its raw range and is the one that rotates, its new position is centred on the TWAP and in range. `anyOor` still holds the stale `true`, though, and `_setOorDivertActive(anyOor)` arms the divert. Phase 4 then sends 100% of the cycle's surplus to staking, even though no target is out of range anymore. This does

not happen on every rotate: `rebalanceOne()` can also fire while the TWAP is only inside the `oorBufferTicks` wing, where `checkRangeStatus()` still reports the position as in range.

Impact

Informational. The LP sleeve can miss one week of new capital after a rotate, which skews the target split toward staking.

Recommendation

Re-evaluate the rotated target after a successful `rebalanceOne()`, by calling `checkRangeStatus(newSN)` through `getActivePositionSN(i)`. Alternatively, compute `anyOor` in a second pass after the rotate attempt.

Developer Response

Fixed in commit fd32346.

3.2.13 Owner-side harvest-lock hatches on the LP strategy are unreachable once the allocator owns it

Technical Details

The LP strategy documents two ways to clear a stuck harvest pass:

- `resetHarvestPass()`, `onlyOwner`, described as "the emergency hatch" in the contract NatSpec.
- `abortHarvestPass()`, owner or `StrategyManager.keeper()`.

In production the strategy's owner is `LynxLpAllocator`. The allocator never calls either function and has no generic passthrough, so the owner branch is dead code. The only live caller is the keeper router, and only from `_advanceOpsPhase(0)` and `stopWeeklyLoop()`, the latter only while `loopActive`.

One reachable state is left without a direct hatch. `startWeeklyLoopAndRun()` runs a phase-0 page that leaves `harvestPassActive == true`, then fails to schedule. It returns with `loopActive == false`, so `stopWeeklyLoop()` reverts `LoopNotActive`. `lpHarvestPagesThisCycle` is also left non-zero and is inherited by the next `startWeeklyLoop()`. Until the router owner drives the pass to completion by calling `weeklyOps()` by hand, or restarts the loop, every `_requireNoHarvestPass()` path is blocked, including `emergencyUnwind()` and `emergencyWiden()`.

Impact

Informational. Recovery is possible through the router owner, or by pointing `StrategyManager.keeper()` at an EOA. The advertised strategy-level hatch, however, cannot be used, and the emergency paths depend on keeper liveness.

Recommendation

Add an owner-only `resetHarvestPass()` passthrough on the allocator. In `startWeeklyLoopAndRun()`, when scheduling fails, abort any pending pass and reset `lpHarvestPagesThisCycle` / `consecutivePhase0Failures`, the same way `stopWeeklyLoop()` does.

Developer Response

Fixed in commits f58eb85, cff5d62.



LYNX LP Range Manager

Completed 2026-10-07