yAudit

July 2026

Prepared for
Lynxify

Audited by
fedebianu
devtooligan

LYNX Token Contracts

Smart Contract Security Assessment

Contents

1 Review Summary

1.1 Protocol Overview

Lynxify is a Hedera-native yield protocol centered on an HBAR vault and an HTS bridge that represents vault ownership as LYNX. StrategyManager allocates assets across composite staking and SaucerSwap liquidity strategies, while keeper workflows manage harvesting, position maintenance, withdrawals, and migrations. The protocol also depends on governance, guardian, and operator roles and integrates Hedera Token Service and Hedera Schedule Service operations.

1.2 Audit Scope

This audit covers 17 smart contracts totaling approximately 3049 lines of code across 4 days of review.

```
hardhat/contracts
├── HarnessStrategy.sol
├── HederaResponseCodes.sol
├── HederaStakingStrategy.sol
├── LynxCompositeStakingStrategy.sol
├── LynxHTSBridge.sol
├── LynxKeeperRouter.sol
├── LynxLpAllocator.sol
├── LynxStakingAllocator.sol
├── LynxVault.sol
├── SaucerSwapLiquidityStrategy.sol
├── StrategyManager.sol
├── VaultMigrationStrategy.sol
├── governance
│   ├── LynxGovernor.sol
│   ├── LynxTimelock.sol
│   └── LynxVotes.sol
├── interfaces
│   ├── IHederaScheduleService.sol
│   ├── ILynxCompositeStakingOps.sol
│   ├── ILynxKeeperRouterSplit.sol
│   ├── ILynxLpAllocator.sol
│   ├── ILynxStakingAllocator.sol
│   ├── ILynxVault.sol
│   ├── IStrategy.sol
│   └── IStrategyManager.sol
└── libraries
    ├── FullMath.sol
    ├── HssOps.sol
    ├── HtsOps.sol
    ├── LiquidityAmounts.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	3
Medium	11
Low	11
Informational	10

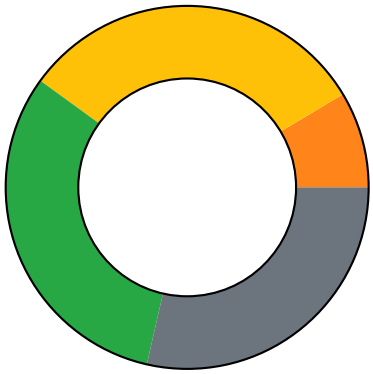

Figure 1: Distribution of security findings by impact level

1.5 Overall Assessment

The main risks concern migration sequencing and accounting, permissionless keeper workflows, asset valuation and loss recognition, and HTS share accounting. The protocol has a clear role

model and uses established primitives, but privileged operations and Hedera-specific integrations require careful deployment, migration, and keeper procedures.

2 Audit Overview

2.1 Project Information

Protocol Name: Lynxify
Repository: https://github.com/Lynxify-xyz/lynx-contracts/
Commit Hashes:

- 116b6c5dddb1a5f75f7f442631df3adcbcd937e4
- b27206b8c99c459a1689aa45d153f68454f8e1ea

Commit URLs:

- https://github.com/Lynxify-xyz/lynx-contracts/tree/116b6c5dddb1a5f75f7f442631df3adcbcd937e4
- https://github.com/Lynxify-xyz/lynx-contracts/tree/b27206b8c99c459a1689aa45d153f68454f8e1ea

2.2 Audit Timeline

The audit was conducted from July 20 to 23, 2026.

2.3 Audit Resources

- Code repositories and documentation

Category	Mark	Description
Access Control	Average	The role model is explicit, but safety-critical keeper and migration paths rely on correctly configured privileged roles and operational procedures. Public keeper workflows also require strong validation of the state transitions they can trigger.
Mathematics	Low	Accounting and valuation logic produced several material findings, including profit recognition, loss accounting, LP valuation, and HTS share-scaling issues. These calculations directly affect NAV, withdrawals, and user value.
Complexity	Low	The vault, HTS bridge, strategy manager, staking and liquidity strategies, keeper router, and migration flows are tightly coupled. Correct behavior depends on preserving invariants across multiple contracts and Hedera-specific services.
Libraries	Good	No finding arose from a third-party library. The reviewed risk is concentrated in custom accounting, state-management, and Hedera integration logic.
Decentralization	Low	Governance, guardian, operator, and owner roles control core configuration, strategy operations, pausing, and migrations. These roles are central trust boundaries.
Documentation	Average	The contract and interface structure communicates the main system components, but several security-relevant assumptions about migrations, accounting, and operational flows were not sufficiently explicit.
Testing and verification	Average	The suite covers core workflows, but the audit identified gaps in adversarial state transitions, including migration initialization, complete LP lifecycles, cross-component accounting, and external-balance conditions.

Table 2: Code Evaluation Matrix

2.4 Critical Findings

None.

2.5 High Findings

2.5.1 Migration seed ordering can brick cutover or let migrated NAV be captured

Technical Details

The documented migration order cannot seed the successor bridge after the source has already been drained. Reordering the same operations avoids the self-brick, but opens a direct-share capture path while the successor vault is unpaused for seeding.

The runbook drains the source vault into the successor first (step 4), then seeds the successor bridge (step 5) (`docs/migration.md:88-114`). The drain sends the source's WHBAR straight to the successor vault as a raw transfer with no shares minted (`VaultMigrationStrategy.withdraw()`). So by the time the seed runs, the successor holds the migrated money but has issued zero shares.

Seeding the bridge means depositing into the successor to mint it the vault shares that back the circulating LYNX. But an ERC-4626 vault that already holds assets while zero shares exist prices its first shares off those sitting assets, and the deposit needed explodes.

Concretely, for a 100 HBAR migration:

- The source held 100 HBAR. With `_decimalsOffset() == 6` that is `1e16` vault share units, so the bridge must be seeded with `1e16` shares to cover the LYNX supply.
- After the drain, the successor holds 100 HBAR of assets and 0 shares. To mint `1e16` shares into it, the deposit required is about 1 trillion HBAR.
- The seed script sizes this deposit by probing `previewDeposit()` (`seed-vault-backing.ts:108-137`), and its search overflows its own bound and reverts (`seed-vault-backing.ts:124`).

The migration will then halt at the seed step with the source already emptied. LYNX holders cannot be moved to the successor, and any holder that unwraps to source vault shares is left with shares whose `maxRedeem()` is zero until governance reruns the migration through a corrected path.

Alternate fund draining — theft if the order is reversed

The obvious way to dodge the pricing problem is to seed the successor while it is still empty, then move the migrated money in. That removes the denial, but it opens a theft window. The successor vault may be auto-paused after deployment, but the runbook and seed script require the vault to be unpaused before seeding (`docs/migration.md:103-114`, `seed-vault-backing.ts:7,243-248`). During that unpaused window, direct ERC-4626 deposits are public, and the migrated money still arrives later as a raw donation that mints no shares.

An attacker deposits into the successor before the raw migrated money lands. On an empty, offset-6 vault, a `0.01` HBAR deposit mints a large share count. The operator then mints the bridge its backing shares and moves the migrated NAV in as a raw donation. That NAV backs

every share pro rata, so the attacker's shares, bought for `0.01` HBAR, are now worth about half the migrated NAV. In the PoC, the attacker turns `0.01` HBAR into `49,999.98` HBAR against a `100,000` HBAR NAV. Depositing before the bridge seed makes the attacker one of the two large shareholders; depositing before any seed or drain can make the attacker the dominant shareholder.

The seed's success check does not catch this. It verifies only `bridge.backingShares() == LYNX.totalSupply()` (`seed-vault-backing.ts:355-371`), and `backingShares()` is `vault.balanceOf(bridge)`, which the attacker's shares do not change. The script reads `vault.totalSupply()` but only logs it, so it prints `SUCCESS` with the attacker's shares in place.

Proof-of-Concept

Two Foundry tests pass at commit `116b6c5`.

`test/poc/MigSeedDosSelfBrick.t.sol` follows the documented drain-then-seed order and shows the seed cannot be funded — a 100 HBAR migration needs about 1 trillion HBAR, and even the entire network supply falls short.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.28;

import {LynxPocBase} from "./PocTemplate.t.sol";

contract MigSeedDosSelfBrick is LynxPocBase {
    function test_drainBeforeSeedMakesSeedingImpossible() public {
        uint256 nav = 100 * ONE_HBAR; // 100 HBAR migrated
        uint256 targetShares = nav * 1e6; // backing shares the seed must mint

        // Drain: NAV lands in the empty successor as raw WHBAR, no shares minted.
        whbar.mint(address(vault), nav);
        assertEq(vault.totalSupply(), 0);
        assertEq(vault.totalAssets(), nav);

        // Seed: even depositing the entire HBAR supply cannot mint the backing shares.
        uint256 totalHbarSupply = 50_000_000_000 * ONE_HBAR;
        assertLt(vault.previewDeposit(totalHbarSupply), targetShares, "whole supply
falls short");

        uint256 neededBaseUnits = targetShares * (vault.totalAssets() + 1) / 1e6;
        emit log_named_decimal_uint("NAV migrated        (HBAR)", nav, 8);
        emit log_named_decimal_uint("seed deposit needed (HBAR)", neededBaseUnits, 8);
        emit log_named_decimal_uint("whole HBAR supply   (HBAR)", totalHbarSupply, 8);
    }
}
```

`test/poc/MigSeedCaptureMinimal.t.sol` shows the theft in the reversed order — a `0.01` HBAR deposit captures about half a `100,000` HBAR NAV, and the seed check still passes.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.28;

import {LynxPocBase} from "./PocTemplate.t.sol";

contract MigSeedCaptureMinimal is LynxPocBase {
```

```
7       address internal attacker = makeAddr("attacker");
8
9       function test_dustDepositCapturesHalfTheDonatedNAV() public {
10          uint256 target = 10_000 * ONE_HBAR; // shares the operator must seed to the
    bridge
11          uint256 donation = 100_000 * ONE_HBAR; // migrated NAV, moved in as raw WHBAR
12
13          // Attacker deposits dust first; on an empty offset-6 vault this buys ~target
    shares.
14          uint256 attackerShares = depositWhbar(attacker, target / 1e6);
15
16          // Operator seeds the bridge with ~target shares, then moves the NAV in (no new
    shares).
17          whbar.mint(address(this), target / 1e6);
18          whbar.approve(address(vault), target / 1e6);
19          vault.deposit(target / 1e6, address(bridge));
20          whbar.mint(address(vault), donation);
21
22          // Seed's own check passes (bridge backs ~target) but supply is ~2*target.
23          assertApproxEqRel(bridge.backingShares(), target, 0.01e18);
24          assertApproxEqRel(vault.totalSupply(), 2 * target, 0.01e18);
25
26          vm.prank(attacker);
27          uint256 got = vault.redeem(attackerShares, attacker, attacker);
28
29          emit log_named_decimal_uint("attacker deposit  (HBAR)", target / 1e6, 8);
30          emit log_named_decimal_uint("attacker captured (HBAR)", got, 8);
31
32          assertGt(got, (donation * 40) / 100);
33          assertLt(got, (donation * 60) / 100);
34      }
35  }
```

Impact

High. As documented, the migration cannot complete: the source is drained and the successor bridge cannot be seeded with a realistic deposit, stranding holders until governance reruns the migration another way. Reordering to seed the empty successor first removes the denial, but lets a direct successor-vault depositor capture migrated NAV for a dust deposit while the seed script still reports success. The operation is operator-initiated and infrequent, which lowers likelihood, but the vulnerable state is created by the published migration sequence and seed requirements.

Recommendation

Mint the bridge's backing shares while the successor stays paused, before any migrated money reaches the vault, and unpause only once the backing is in place. Seeding into an empty, paused vault mints the shares at the correct price (removing the denial) and blocks any third-party deposit during the seed (removing the theft), since `deposit`/`depositHbar` are gated by `whenNotPaused`. Give the operator an owner-only entry point that mints the backing shares to the bridge while paused, then moves the migrated money in.

Also harden the verify step to require `vault.totalSupply() == bridge.backingShares()`. After a correct seed the only shares in existence are the bridge's, so that equality fails the

moment anyone else holds shares, which the current `backingShares() == LYNX.totalSupply()` check misses.

Developer Response

Fixed in commits 1fa97af, f944ead, 420f5b2, and fafd5d1.

2.5.2 Public keeper paths can execute DEX operations with weak slippage bounds

Technical Details

Several market-sensitive operations are reachable through public router functions: `weeklyOps()`, `refillBuffer()`, `reconcileInactiveLp()`, `pushAndDeploy()`, and `allocateSurplus()`. The router is the manager keeper in the deployment flow, so these public calls can move vault capital through strategies.

The on-chain slippage bounds are absolute, not quote-scaled. The LP reconciliation path drains disabled LPs with `amount0Min = 0`, `amount1Min = 0`, and `drainSwapMinOut`; the `constructor` sets `drainSwapMinOut = 1`. `LynxLpAllocator.deployLpIdle()` relies on global `minSwapOut`, `amount0Min`, and `amount1Min`; `setTargets()` stores those globals and `_swapAndPrepareAmounts()` applies the same absolute `minSwapOut` per target. The deploy script defaults `KEEPER_SWAP_MIN_OUT` to `1` when swaps are enabled and encodes LP mint mins as zero (keeper deploy defaults).

The composite staking unwind path also hardcodes `minOut = 1` in `_swapHoldLegsToWhbar()` when selling hold legs for WHBAR. Both LP and composite swaps use `swapSqrtPriceLimitX96`, which defaults to zero and therefore imposes no price limit (LP declaration, composite declaration).

The same public composite unwind has an accounting effect beyond slippage. When `refillBuffer()` needs a small deficit, `_swapHoldLegsToWhbar()` sells the entire hold-leg balance, and `_sendWhbarToVault()` caps the returned amount at the requested deficit. In `StrategyManager.pull()`, the returned amount then equals the basis reduction, so no loss and no gain is booked. The excess swap proceeds sit idle at the strategy, hold-leg basis diverges from actual holdings, and `harvest()` never revalues hold legs to repair it. The result is not a booked loss but a silent NAV overstatement, so remaining shareholders redeem at an inflated price.

Impact

High. Anyone can trigger swaps and LP mint/decrease flows during a manipulated or sandwiched market state. Because minimum outputs can be `1` or zero for production-sized amounts, the protocol can realize large actual trading losses while satisfying all on-chain checks.

Recommendation

Make slippage protection amount-scaled and enforce it on-chain for all router-triggered operations. Compute `minOut`, LP amount mins, and `sqrtPriceLimitX96` from fresh trusted quotes. Consider making market-executing router functions authenticated, or require signed keeper quotes with expiry for public execution.

For the composite unwind, forward all swap proceeds to the vault and report the realized gain or loss, or reduce basis by the full amount sold, so a partial refill cannot leave unbooked proceeds and an overstated NAV, in addition to the slippage bounds above.

Developer Response

Fixed in commits c21e813 and 451fbd6.

I went with protecting the routes - we don't need public calls for incentivized ops or anything like that - along with scaling slippage. We can maintain it with the keeper or internal ops calls as backup.

2.5.3 `LynxCompositeStakingStrategy` withdrawals can turn remaining HBAR principal into fake harvest profit

Technical Details

`LynxCompositeStakingStrategy.withdraw()` first sends WHBAR to the vault and only then calls `_reducePrincipal()`. `_sendWhbarToVault()` computes available liquid native value, swaps whole hold legs if there is a deficit, wraps only the WHBAR shortfall, and transfers the requested amount to the vault.

After that transfer, the principal reducer re-reads the remaining WHBAR plus native HBAR balance and treats that post-transfer balance as the amount that should reduce `nativePrincipal`. This is backwards: the remaining HBAR is still held by the strategy, while the sold hold leg has actually been consumed.

Example scenario: with equal 25% weights, pushing `100_000` records `25_000` native basis and `25_000` basis in each hold leg. Pulling `26_000` uses `25_000` native HBAR plus `1_000` wrapped HBAR after selling an entire hold leg. The strategy then has `24_000` HBAR and zero tokens in that sold hold leg, but the reducer leaves only `1_000` native basis and leaves `23_000` basis on the empty hold leg. The next `harvest()` sees `24_000 - 1_000 = 23_000` as native gain.

Proof of concept

Add the following test to `test/LynxCompositeStakingStrategy.test.ts`:

```
it("PoC: withdrawal after hold-leg liquidation creates false harvest profit", async () =>
  const s = await deployCompositeStack();
  await (await s.allocator.connect(s.governance).setStakingWeights([2500, 2500, 2500, 250
  await fundVault(s.whbar, s.vault, s.alice, 100_000n);

  const compositeAddr = await s.composite.getAddress();
  await (await s.manager.connect(s.keeper).push(compositeAddr, 100_000n)).wait();
  await (await s.manager.connect(s.keeper).pull(compositeAddr, 26_000n)).wait();

  expect(await conn.ethers.provider.getBalance(compositeAddr)).to.equal(24_000n);
  expect(await s.xsauce.balanceOf(compositeAddr)).to.equal(0n);
  expect(await s.composite.nativePrincipal()).to.equal(1_000n);
```

```
  expect(await s.composite.holdPrincipalWhbar(2)).to.equal(23_000n);

  const [gain, loss] = await s.manager.connect(s.keeper).harvest.staticCall(compositeAddr
  expect(gain).to.equal(23_000n);
  expect(loss).to.equal(0n);

  await (await s.manager.connect(s.keeper).harvest(compositeAddr)).wait();
  expect(await s.vault.totalAssets()).to.equal(100_000n);

  await conn.networkHelpers.time.increase(7 * 24 * 60 * 60 + 1);
  expect(await s.vault.totalAssets()).to.equal(123_000n);
});
```

Impact

High. The vault can stream non-existent gains into NAV. Existing and future share pricing becomes inflated after `lockedProfit` unlocks, allowing mispriced deposits/redeems and making reported assets exceed physical assets.

Recommendation

Do not derive principal reduction from post-withdrawal balances. Track the cost basis consumed from native HBAR, idle WHBAR, and each hold leg during liquidation. Clear a leg's basis when fully sold, or reduce it proportionally when partially sold. Account for swap proceeds above or below the released basis as realized gain or loss, and keep the strategy's internal basis synchronized with the manager's principal.

Developer Response

Fixed in commits 7ac9fcf, 01e9b17, and 1d3db37.

2.6 Medium Findings

2.6.1 Spot deviation can veto impairment synchronization

Technical Details

The LP and composite strategies value positions through a mature TWAP, but their valuation helpers also require the current spot price to remain within `maxSpotDeviationBps`. A single pool outside that range makes the aggregate `impairmentValue()` return `(false, 0)`.

`StrategyManager._syncImpairment()` silently preserves the previous impairment on this failure. Nevertheless, `harvest()` unconditionally updates `lastCheckedTime`, so the strategy is marked fresh even though no valid accounting observation was recorded.

After a real loss has matured into the TWAP, a shareholder can briefly move the spot price of one pool beyond the deviation threshold at the predictable keeper execution time. The mature TWAP itself need not be manipulated. The harvest succeeds without booking the loss; the attacker reverses the spot trade and can redeem idle assets at the stale higher NAV. In the LP

path, the same fail-open behavior can hide a loss already realized by draining another position because any unrelated active pool can veto aggregate valuation.

Impact

Medium. A shareholder can externalize part of a genuine strategy loss to remaining holders by redeeming against stale NAV. Extraction is bounded by the unrecognized loss, the attacker's shares, and available idle liquidity. The unconditional timestamp update also allows subsequent manager operations to rely on a valuation that never succeeded.

Recommendation

Do not let an instantaneous spot observation veto a mature TWAP used only for accounting; retain the spot-deviation guard for swap execution. Make `_syncImpairment()` return a success flag, update a dedicated last-valid-valuation timestamp only on success, and emit a failure event otherwise. Loss-sensitive pulls should revert or use authoritative per-position realization when valuation fails. Prevent stale valuations from pricing exits through a bounded conservative policy or fresh-snapshot settlement.

Developer Response

Fixed in commit 15197ae.

2.6.2 Failed pull valuation can leave impairment above principal

Technical Details

`StrategyManager.pull()` reduces the strategy's principal before calling `_syncImpairment()`. If `impairmentValue()` returns `ok == false`, `_syncImpairment()` returns without changing the old absolute impairment.

A sufficiently large pull can therefore leave `impairmentOf[strategy] > principal[strategy]`. `totalAccountedPrincipal()` subtracts global impairment from global principal and floors the result at zero, so excess impairment from one strategy can also suppress healthy basis belonging to another strategy.

The invalid state can additionally break later accounting: `_increaseImpairment()` computes `target - current` after capping `target` to basis, and `_bookPullRealization()` computes `basisReduce - impairmentReleased`. Either subtraction can underflow when stale impairment exceeds the reduced basis.

Impact

Medium. Deposits made while NAV is artificially suppressed receive excess shares and benefit when impairment is later corrected, diluting incumbents. The same invariant violation can block later loss booking or pulls. The impact can include healthy strategy principal, although reaching the state requires both existing impairment, a material basis reduction, and an unavailable post-pull valuation.

Recommendation

Enforce `impairmentOf[strategy] <= principal[strategy]` atomically on every basis reduction. If post-pull valuation is unavailable, either revert the pull or cap impairment to the new basis and account for the exact released amount in realized P&L. Aggregate `totalAccountedPrincipal()` per strategy as defense in depth, and use saturating or explicitly checked arithmetic in subsequent impairment and realization calculations.

Developer Response

Fixed in commit 4a9509d.

2.6.3 Composite deployments lack amount-scaled slippage protection

Technical Details

`LynxStakingAllocator.deploySplit()` passes the same global absolute `minSwapOut` to every non-native hold leg, regardless of the input amount, output token, token decimals, or market price. The deployment configuration initializes this value to one base unit.

`LynxCompositeStakingStrategy.swapFromWhbar()` then calls `_swap()` without a pool address. Consequently, `_swapWithPoolLimit()` cannot derive either an amount-dependent TWAP minimum or a direction-correct price limit from the configured canonical pool. The caller-provided fee is also not bound to the fee of the pool used for valuation and withdrawals.

An attacker can move a hold-token pool before a predictable keeper allocation, let the strategy execute a large swap against the near-zero absolute floor, and reverse the move afterward. Later impairment accounting recognizes the resulting loss but cannot recover the transferred value.

Impact

Medium. A successful sandwich directly reduces vault-owned strategy assets. The affected amount scales with each deployed leg rather than with `minSwapOut`. Exploitability depends on the size and predictability of pushes and the liquidity of the configured pools, so High severity would require production-specific economic evidence that a material fraction of vault assets can be extracted profitably.

Recommendation

Resolve the hold-leg index before execution, require the leg's configured fee, and bind the swap to its configured pool. Derive `amountOutMinimum` and a direction-correct `sqrtPriceLimitX96` from the trusted TWAP for the actual input amount, enforcing the stricter of that value and any caller-provided floor. Validate each configured pool's token pair and fee.

Developer Response

Fixed in commit c078c52.

2.6.4 Cached impairment misprices vault entry and exit

Technical Details

`LynxVault.totalAssets()` includes deployed assets through `StrategyManager.totalAccountedPrincipal()`, which returns stored principal minus stored impairment. Market-value changes do not update this snapshot automatically; `_syncImpairment()` runs only through keeper-controlled manager operations.

Vault deposits, mints, withdrawals, and redemptions use the cached value without requiring a complete current valuation. `maxStaleAge` does not protect these user operations because it is enforced only before manager pushes and pulls.

This creates both sides of the same stale-NAV problem:

- after an impaired strategy economically recovers, a user can deposit before synchronization at a depressed NAV and share in the subsequent impairment release;
- after a strategy economically loses value, a shareholder can redeem against idle WHBAR at the stale higher NAV and leave the loss to remaining holders.

Impact

Medium. Stale accounting can transfer value between existing holders and users entering or exiting around a keeper snapshot. Immediate loss extraction is bounded by available redemption liquidity, but the misissued shares remain a claim on future vault assets even when the full amount cannot be redeemed immediately.

Recommendation

Settle entry and exit against a fresh, common impairment snapshot for every material strategy. Prefer an atomic vault-authorized synchronization that reverts settlement if any required valuation is unavailable. If synchronous valuation is unsafe or too expensive, queue user operations for the next complete accounting snapshot. A deposit-only gate is insufficient because stale overvaluation also affects exits.

Developer Response

Fixed in commit ba407ca.

2.6.5 Migration drain can report success while LP positions and tokens remain behind

Technical Details

The migration drain treats a strategy's recorded principal as if the full amount is liquid. For the SaucerSwap LP strategy, that assumption is false. An operator can complete the documented drain with source value still deployed in LP non-fungible token (NFT) positions or held as the pool's pair token.

`drain-to-successor.ts` reads each non-migration strategy's principal and calls `StrategyManager.pull(strategy, principal)`. `StrategyManager.pull()` asks the strategy to withdraw that amount, but reduces the strategy's principal by the requested amount even when the strategy returns less. It books the difference as a vault loss.

`SaucerSwapLiquidityStrategy.withdraw()` only transfers loose WHBAR held by the strategy. It does not decrease, collect, or close an active LP position, and it does not convert a loose pair-token balance. Let:

- `P` be the LP strategy's recorded principal
- `L` be the loose WHBAR returned by `withdraw(P)`
- `R` be the recoverable value left in active positions and pair tokens

When `L < P`, the pull has these effects:

1. The vault receives `L`.
2. The manager reduces the LP strategy's principal from `P` to zero.
3. The vault books a loss of `P - L`.
4. Residual value `R` remains in the source strategy.

The accounting loss `P - L` is not necessarily equal to `R`: the position's market value may have changed since deployment. The drain sends the source vault's idle WHBAR to the successor, but none of `R` moves.

The script's post-checks do not catch the shortfall. They check only `totalPrincipal()` and `totalAssets()` on the source. Both can be zero because the pull erased the LP principal and booked the missing amount as a loss. The script prints the amount received by the successor but never compares it with the source's pre-drain net asset value. It then instructs the operator to seed the successor bridge and retire the source stack.

The runbook makes this failure plausible. It briefly says to pull deployed LP capital before the drain, but its drain step says the script pulls all non-migration strategies, including the LP strategy. The prerequisite checklist does not require all LP positions to be closed, and the script does not enforce that condition.

Pausing does not guarantee that the prerequisite remains true. In router-managed configurations, anyone can call `pushAndDeploy()` while the vault is paused, so an account can create or increase LP exposure before the operator transfers the manager keeper role for migration. This is the same pause limitation reported in issue #19, but here it can create the state that the drain mishandles.

An owner can recover `R` while the source contracts and roles remain available by closing each position, converting the pair tokens to WHBAR, and pulling the recovered WHBAR through the manager. Recovery becomes more involved after the false success:

- `totalPrincipal == 0` allows the vault owner to replace the old strategy manager despite the residual LP value.
- A later pull through the old manager then reverts when it calls `vault.reportGain()`, unless the owner temporarily restores that manager or uses a direct ownership-based recovery path.

- If the value is pulled through the manager after its principal reached zero, the manager reports all of it as gain. The vault profit-locks the recovery, which can temporarily reduce the migration strategy's cap to zero.

These are recoverable governance and sequencing problems while the old stack remains controlled. Retiring its contracts, keys, or operational support before checking the LP state can strand the value permanently.

Impact

Medium. A migration with active LP positions or pair-token balances can move only the strategy's loose WHBAR while writing off its remaining principal. The successor omits residual value `R` even though the drain's accounting checks pass. The capital remains recoverable while the source stack and its governance roles remain available, which limits the severity. Retiring the source before recovery can make the loss permanent.

Recommendation

Abort the migration unless every source strategy has zero active exposure after its strategy-specific unwind procedure. For the LP strategy, require `getActiveLPPositionCount() == 0` and zero residual pair-token balance before reducing its principal. Disable public deployment and complete the required keeper and owner role handoffs before beginning the drain.

Add a value-conservation check to the drain. Compare the successor's WHBAR increase with the source's pre-drain recoverable value and fail on a shortfall outside an explicit tolerance. Do not treat zero accounting principal or zero `totalAssets()` as proof that no strategy exposure remains.

Update the runbook to require closing all LP positions, converting pair tokens to WHBAR, and verifying zero residual exposure before running the drain.

Developer Response

Fixed in commits 42b970c and 33f010d.

2.6.6 Migration omits source shares held outside bridge escrow

Technical Details

The source vault's ERC-20 shares are canonical claims and may legitimately be held outside `LynxHTSBridge`. Direct ERC-4626 deposits mint shares to EVM users, while `unwrap()` burns a holder's LYNX and returns the corresponding escrowed source-vault shares. Shares may also be deposited into integrations such as `LynxVotes`. Consequently, the following relationship can hold during normal operation:

```
sourceVault.totalSupply() > sourceBridge.backingShares() == LYNX.totalSupply()
```

The documented migration does not account for this difference. After source assets have been consolidated into `VaultMigrationStrategy`, its `withdraw()` transfers WHBAR directly to the successor vault and reports the transferred amount as `received`. `StrategyManager.pull()` therefore reduces source `principal` and `totalPrincipal` as though the assets had been returned to the source vault. Once the full balance has been relayed, the source vault can report zero `totalAssets()` even though its share supply remains outstanding.

Successor allocation is performed separately. The migration procedure mints successor-vault shares to the successor bridge until its `backingShares()` equals the circulating LYNX supply, then treats that equality as the cutover invariant. It does not mint or reserve successor claims for source shares held anywhere other than the source bridge. The invariant can therefore pass while direct source shareholders have neither assets remaining in the source vault nor shares in the successor vault.

Pausing the source stack does not close the accounting gap. Pausing intentionally blocks `wrap()` and deposits but leaves `unwrap()` available, so a LYNX holder can burn LYNX for source shares during the multi-transaction cutover. This simultaneously reduces the LYNX supply used to size successor backing and moves the holder into the omitted source-share set.

Impact

Medium. A completed drain can leave direct ERC-4626 shareholders, unwrapped LYNX holders, and shares held by other EVM integrations without source backing or a successor claim. If the source is fully drained, affected holders can lose nearly the entire value represented by their shares, while the migrated assets back only the successor-side allocation derived from circulating LYNX.

Exploitation is not required: direct deposits and unwrapping are supported protocol flows. The likelihood depends on whether any shares are outside bridge escrow at cutover, but the impact to each omitted holder is severe and the documented bridge-backing check does not detect the condition.

Recommendation

Account for the full source-vault share supply during migration. Escrow or burn every source share and provide an on-chain claim mechanism that issues the corresponding successor share or LYNX under a fixed, audited conversion ratio. Finalization should reserve an immutable successor claim pool for any source shares that have not yet been converted.

If migration is intentionally limited to circulating LYNX, require `sourceVault.totalSupply() == sourceBridge.backingShares() == LYNX.totalSupply()` before the drain and prevent `unwrap()` from invalidating that equality until migration is finalized. The check and the transition into this migration-lock state must be atomic. Also verify total successor share supply and all intended allocations rather than treating `successorBridge.backingShares() == LYNX.totalSupply()` as a complete economic invariant.

Developer Response

Fixed in commits a33b90c and 6e88691. With this current iteration, we will be focusing strictly on the HTS interface for our users to avoid the complexity. This will need to be extended in the

future, however. Updated with the minimal surface change option – no orphan shares for lock.

2.6.7 Loss booking is unreachable because the keeper is the router, which has no path to call it

Technical Details

`StrategyManager.bookLoss()` and `StrategyManager.bookLossToExposure()` are the manual NAV-correction functions. They write down `principal[strategy]` and `totalPrincipal`, call `LynxVault.reportLoss()`, and refresh `lastCheckedTime[strategy]`. Both carry the `onlyKeeper` modifier, so only the address stored in `keeper` can call them.

In the deployed configuration, the keeper is the `LynxKeeperRouter` contract. `hardhat/scripts/deploy/lynx-keeper-stack.ts` calls `manager.setKeeper(router)`, and `hardhat/scripts/ops/governance/audit-roles.ts` asserts that `StrategyManager.keeper` equals the router and fails otherwise. The router must be the keeper because its automation calls `manager.harvest()`, `manager.pull()`, `manager.push()`, and `manager.pushSplit()`, all `onlyKeeper`.

The router never calls `bookLoss()` or `bookLossToExposure()`. Its full external surface is `weeklyOps()`, `harvestLp()`, `harvestStaking()`, `refillBuffer()`, `reconcileInactiveLp()`, `pushAndDeploy()`, `allocateSurplus()`, and the owner-only `pushStrategy()`. The only manager methods it invokes are `harvest`, `pull`, `push`, `pushSplit`, and `principalOf`. There is no generic passthrough. So while the router is the keeper, no caller can reach `bookLoss` or `bookLossToExposure`.

The ops runbook assumes otherwise. `hardhat/scripts/ops/book-loss.ts` submits `bookLossToExposure(strategy)` from an operator EOA it labels "keeper (or manager keeper key)". Because the keeper is the router contract and not that EOA, the call reverts with `NotAuthorized`.

Loss booking matters because harvest does not capture every loss. `LynxCompositeStakingStrategy.harvest()` measures only WHBAR plus native HBAR balance and hard-codes `loss = 0`, so depreciation of the HBARX, DOVU, or XSAUCE hold legs is never marked down through harvest. `bookLossToExposure()` is the designated correction: it books `principal - exposureValue()`, and `exposureValue()` prices the hold legs. With the correction unreachable, `StrategyManager.pull()` still realizes a loss on the portion actually withdrawn, but the write-down of the unwithdrawn principal cannot be applied.

The only way to run the correction is for the owner to `setKeeper()` to an EOA, call `bookLossToExposure()` from that EOA, then `setKeeper()` back to the router. During that window, the router is not the keeper, so the weekly harvest, buffer refill, and deploy automation are all halted, and `audit-roles.ts` reports the keeper role as misconfigured.

Impact

Medium. In the shipped configuration, the manual loss-booking functions cannot be called, so a loss that harvest does not capture, such as a depreciated staking hold leg, cannot be written down. Share price then stays above recoverable value until the position is withdrawn and the loss realized through `pull()`, and any redeemer who exits during that window takes more than

their share and dilutes the rest. The documented `book-loss.ts` runbook reverts. Note: the owner can recover by pointing the keeper at an EOA, booking the loss, and restoring the router, but that halts keeper automation and breaks the enforced keeper role for the window.

Recommendation

Give the keeper router a path to the loss-booking functions so the intended correction runs without changing the keeper role. Add thin owner/operator-gated forwarders on the router.

```
function bookLossToExposure(address strategy) external onlyOwner returns (uint256 booked
) {
    booked = manager.bookLossToExposure(strategy);
    emit LossBooked(strategy, booked);
}

function bookLoss(address strategy, uint256 amount) external onlyOwner returns (uint256
booked) {
    booked = manager.bookLoss(strategy, amount);
    emit LossBooked(strategy, booked);
}
```

Alternatively, gate `bookLoss` / `bookLossToExposure` on the manager owner in addition to the keeper, and update `book-loss.ts` to submit from the authorized key. Either way, align the runbook with the on-chain access model.

Developer Response

Fixed in commit fb0613a.

2.6.8 Incorrect HTS `approveNFT()` selector prevents LP position closure and exhausts the position registry

Technical Details

`IHederaTokenService.approveNFT()` declares the NFT serial number as `int64`. However, the official Hedera Token Service interface declares it as `uint256` (see also Hedera docs):

```
// Local interface
function approveNFT(address token, address approved, int64 serialNumber)
    external returns (int64 responseCode);

// Official HTS interface
function approveNFT(address token, address approved, uint256 serialNumber)
    external returns (int64 responseCode);
```

The parameter type is part of the ABI signature, so the two declarations produce different selectors:

```
approveNFT(address,address,int64)   -> 0x10585c46
approveNFT(address,address,uint256) -> 0x7336aaf0
```

Consequently, `HtsOps.approveNFT()` downcasts the serial number and calls an entry point that the HTS precompile does not expose.

`SaucerSwapLiquidityStrategy.closeLPPosition()` invokes the affected helper before burning the LP NFT and removing its serial from `lpPositionSerials`. Failure therefore reverts the complete close transaction before these state updates:

```
HtsOps.approveNFT(lpNft, nftManager, tokenSN);
ISaucerSwapV2NonfungiblePositionManager(nftManager).burn(tokenSN);
position.active = false;
_removeSerial(tokenSN);
```

This also affects the standalone `approveLPNFTForBurn()`.

The strategy limits the tracked serial array to `MAX_ACTIVE_LP_POSITIONS = 32` to bound iteration gas. New positions are rejected when `lpPositionSerials.length` reaches this limit. Since successful closure is the only path that calls `_removeSerial()`, affected deployments cannot reclaim occupied slots and will reject every later position creation after 32 serials have been registered.

Impact

Medium. The production HTS ABI mismatch makes the advertised LP close operation deterministically unusable. As a result, an LP NFT cannot be burned, marked inactive, or removed from the bounded registry through the intended lifecycle. Any deployment that creates 32 distinct positions over time, including through ordinary range or pool rotation, permanently exhausts the registry and causes all subsequent `createLPPosition()` calls to revert.

This is persistent broken functionality in the LP allocation subsystem rather than a transient configuration failure. Restoring the ability to close positions and reclaim registry capacity requires fixing or replacing the deployed strategy and rewiring its operational ownership. The issue is not High because liquidity can still be removed and collected through `decreaseLPPosition()` and the allocator's drain flow, and it does not directly block vault withdrawals or cause asset loss.

Recommendation

Match the official HTS ABI by declaring `approveNFT()` with a `uint256 serialNumber` and pass the serial directly without `_toInt64()`:

```
function approveNFT(address token, address approved, uint256 serialNumber)
    external returns (int64 responseCode);

function approveNFT(address token, address approved, uint256 serialNumber) internal
returns (int64 rc) {
    rc = IHederaTokenService(HTS).approveNFT(token, approved, serialNumber);
    if (rc != HederaResponseCodes.SUCCESS) {
        revert HTSCallFailed("approveNFT", rc);
    }
}
```

Developer Response

Fixed in commit ec929b9.

2.6.9 Unvalued pair-token balances enable false loss booking and share dilution

Technical Details

`SaucerSwapLiquidityStrategy.harvest()` collects fees from every active LP position. Since each position pairs WHBAR with another token, `_collect()` can transfer both WHBAR and non-WHBAR pair tokens into the strategy. However, `harvest()` recognizes fee gain only from the increase in the strategy's WHBAR balance:

```
uint256 before = _whbarBalance();
// Collect both tokens from every active position.
uint256 feeGain = _whbarBalance() - before;
```

The subsequent mark-to-market calculation sets `currentTotal` to idle WHBAR plus the value of liquidity still held inside active LP NFTs. `_tryTotalLpWhbarValue()` does not include non-WHBAR token balances held directly by the strategy. Such balances can arise from fee collection, liquidity decreases, or tokens left unused when adding liquidity.

```
uint256 idleNow = _whbarBalance();
uint256 currentTotal = idleNow + lpValue;
uint256 costBasis = IStrategyManager(manager).principalOf(address(this));
if (costBasis > currentTotal) {
    loss = costBasis - currentTotal;
}
```

Consequently, the strategy can report a loss even though the apparently missing value remains in its custody as a loose pair-token balance. `StrategyManager.harvest()` immediately reduces the strategy's recorded principal and `totalPrincipal` by the reported loss and forwards it to `LynxVault.reportLoss()`.

`exposureValue()` contains the same omission. Moreover, the deployed keeper router exposes `harvestLp()` publicly, allowing any account to trigger the write-down once a material loose pair-token balance exists.

Impact

Medium. The false loss understates the strategy's recorded principal while the omitted value remains controlled by the strategy. Any portion not offset by the vault's remaining locked-profit buffer immediately lowers vault NAV and share price; an offset portion instead consumes legitimate locked profit that would otherwise have streamed into NAV.

An attacker can deposit while shares are underpriced and receive more shares than the strategy's complete asset value justifies. If the omitted pair tokens are later converted to WHBAR and realized through an operational pull or unwind, the recovered value is shared with the attacker's over-issued shares, diluting incumbent depositors. The maximum distortion is bounded by the WHBAR value of the omitted token balances.

Recommendation

Before reporting a loss, include every directly held pair-token balance in the strategy's WHBAR-denominated value. Track the tokens used by active and recently drained positions and value their loose balances using a manipulation-resistant price source. Apply the same complete valuation in `exposureValue()`.

Alternatively, atomically convert all collected non-WHBAR amounts to WHBAR with enforceable, amount-scaled slippage bounds before calculating gain or loss. If a directly held token cannot be valued or safely converted, suppress loss booking until its value can be accounted for instead of treating the balance as worthless.

Developer Response

Fixed in commits d0f45d5, 986902e, and 6635226.

2.6.10 Raw 1:1 share wrapping can exhaust HTS LYNX supply at approximately 92,233 WHBAR

Technical Details

`LynxVault` sets the OpenZeppelin ERC-4626 `_decimalsOffset()` to `6`. Since WHBAR has 8 decimals, vault shares have 14 decimals, as confirmed by the repository's `decimals offset` test. At the initial 1:1 asset/share exchange rate, one WHBAR base unit therefore mints `10^6` vault-share base units.

The HTS LYNX token is configured with 8 decimals, but `LynxHTSBridge.wrap()` does not normalize between the two decimal domains. It transfers `shares` raw vault-share units into escrow and passes the same raw amount to `_mintLynxTo()`. `unwrap()` applies the same raw 1:1 mapping in reverse.

HTS mint and burn amounts are signed 64-bit integers. `HtsOps._toInt64()` correctly rejects any individual amount above `type(int64).max`, and `HtsOps.mint()` forwards that bounded amount to the HTS precompile. This prevents unsafe truncation, but it does not remove Hedera's `2^63 - 1` maximum cumulative token supply. Hedera specifies this limit in its token mint documentation.

Consequently, the raw backing model limits the entire bridge to:

```
maximum LYNX / backed share base units = 2^63 - 1
                                       = 9,223,372,036,854,775,807

maximum displayed vault shares = (2^63 - 1) / 10^14
                               = 92,233.72036854775807
```

At an initial 1:1 WHBAR exchange rate, this corresponds to only approximately `92,233.72036854` WHBAR of fully wrapped vault capacity. The asset-denominated threshold changes with the vault share price, but the raw-share ceiling is fixed.

Any single wrap above `int64.max` reverts with `AmountExceedsInt64`, as the existing `HTS int64 downcast guard` test confirms. More importantly, once cumulative LYNX supply

approaches `int64.max`, further mints fail at the HTS layer even when each individual wrap is below the limit. Splitting a wrap across transactions therefore does not bypass the ceiling. `depositHbarAndWrap()` is affected by the same mint path, while direct vault deposits can continue creating canonical shares that cannot be represented as LYNX.

The bridge NatSpec additionally states that both tokens use 8 decimals. This is incorrect and hides both the `10^6` displayed-unit discrepancy and the much lower HTS capacity created by the raw-unit mapping.

Impact

Medium. Natural vault growth can hit the bridge's permanent HTS capacity ceiling at a relatively modest TVL. Once the ceiling is reached, users can no longer wrap new vault shares or use the combined HBAR deposit-and-wrap path, even though direct ERC-4626 deposits continue to work. Existing LYNX remains backed and can still be unwrapped, so the issue does not directly steal funds, but it disables a core Hedera-facing entry path, fragments users between HTS LYNX and unwrapped vault shares, and complicates governance and migration accounting.

An account can also occupy the remaining HTS capacity by depositing and wrapping fully backed shares. The account retains its economic claim, but later users are denied LYNX wrapping until existing supply is burned.

Recommendation

Do not map 14-decimal vault-share base units directly to 8-decimal LYNX base units. Introduce an explicit scale factor:

```
SHARE_SCALE = 10^(vaultShareDecimals - lynxDecimals) = 10^6
lynxAmount  = shareAmount / SHARE_SCALE
shareAmount = lynxAmount * SHARE_SCALE
```

The backing invariant should become:

```
vault.balanceOf(bridge) == LYNX.totalSupply() * SHARE_SCALE
```

Developer Response

Fixed in commits e958ee1 and cd9eba7.

2.6.11 Permissionless spot-price harvest can persistently understate LP NAV

Technical Details

`SaucerSwapLiquidityStrategy.harvest()` values active LP positions from each pool's current `slot0()` price. `_tryTotalLpWhbarValue()` reads the pool price for every active position, and `_positionWhbarValue()` converts token amounts using that same instantaneous price. If the computed value is below manager cost basis, the LP harvest reports `loss`.

`StrategyManager.harvest()` immediately reduces both the strategy's principal and `totalPrincipal`, then calls `vault.reportLoss()`. Because `LynxVault.totalAssets()` includes the manager's recorded principal, the loss immediately lowers share NAV.

The accounting is one-sided. A later price recovery increases the calculated LP value, but `harvest()` reports only newly collected WHBAR fees as `gain`; it does not restore principal previously removed by mark-to-market losses. Ignoring pushes and withdrawals, every complete valuation therefore applies the equivalent of:

```
recordedPrincipalAfter = min(recordedPrincipalBefore, sampledCurrentValue)
```

The recorded principal only moves down to the lowest sampled value and never recovers, even with an honest keeper and a perfect, manipulation-resistant price source.

The deployment flow additionally wires `StrategyManager.keeper` to `LynxKeeperRouter` by calling `StrategyManager.setKeeper()`. The router exposes `weeklyOps()` and `harvestLp()` without access control. There is no minimum harvest interval; `lastCheckedTime` is updated after harvest but does not prevent another harvest. Any account can therefore choose the exact `slot0()` sample used to trigger the ratchet, including immediately after moving the pool price. This permissionless spot-price path amplifies the structural bug but is not its root cause.

`LynxVault.reportLoss()` also reduces `_lockedProfit` by up to the reported loss. Locked profit acts as a first-loss buffer, so this offsets rather than compounds the immediate NAV decrease. Nevertheless, a false write-down consumes legitimate realized profit that would otherwise have streamed into NAV.

The write-down is not irrecoverable in an absolute sense. After a complete unwind and conversion to WHBAR, an operational `pull()` that returns more than the reduced principal can recognize the excess as realized gain. However, this requires a separate operational realization path and is not triggered by price recovery or subsequent harvests.

The automated pipeline does not close this gap. `reconcileInactiveLp()` -> `LynxLpAllocator.drainPosition()` unwinds a disabled position and swaps the pair token back to WHBAR at the strategy, but never calls `StrategyManager.pull()`, and the weekly `refillBuffer()` pulls only the staking strategy, never the LP strategy. After an automated drain the recovered WHBAR sits at the strategy, outside both recorded principal and the redemption buffer, and NAV stays understated until a separate manual `pull()`. The shipped drain script defaults its pull amount to pre-drain principal, leaving above-basis surplus stranded. A depositor entering while NAV is understated captures value from incumbents once the recovery is booked.

Impact

Medium. Routine volatility is sufficient to trigger the issue. Whenever a weekly harvest samples an LP value below recorded principal, it locks in that lower value. A subsequent market recovery does not restore the write-down. Repeated harvests around local lows can therefore make recorded principal drift toward the lowest sampled valuation even though the LP position later recovers. The permissionless spot-price path creates an additional adversarial variant. An attacker can temporarily move `slot0()`, call `harvestLp()` at the distorted price, restore the pool price, and deposit at the resulting understated NAV. This does not produce atomic profit: the attacker must wait for an operational unwind or another realization path, and profitability depends on manipulation cost, fees, strategy exposure, deposit capacity, and realization timing.

Recommendation

Consider fixing the one-sided accounting rather than relying only on access control around `harvestLp()`. Either of the following accounting designs would work:

1. To retain immediate deposits and redemptions, value positions with a manipulation-resistant TWAP or independent oracle, enforce freshness and deviation bounds, and make impairments reversible. Track historical cost basis separately from accounted value so a recovery restores a prior write-down up to cost basis without being treated as newly transferred WHBAR. For conservative cost-basis accounting, the target book value can be `min(historicalCostBasis, robustMarketValue)`.
2. To retain oracle-free cost-basis NAV, remove unrealized loss recognition from `harvest()` and settle withdrawals through a queue or batch after the corresponding LP assets are unwound. Removing mark-to-market loss booking without changing immediate redemptions is insufficient because informed users could exit at stale NAV before a genuine loss is realized.

Making the harvest keeper-only and spacing harvests out helps against an attacker who picks a bad price moment to trigger one, but it does not fix the accounting. Even the honest keeper's normal weekly harvest still books losses that never reverse when the price recovers.

Separately, when governance retires an LP position, the weekly keeper run unwinds it and converts it back to WHBAR but leaves that WHBAR sitting in the LP strategy, without telling the vault the value came back. NAV stays low until someone manually moves it over. Have the unwind move the recovered WHBAR to the vault and record the gain in the same step, so it is not left for a separate manual transfer.

Developer Response

Fixed in commits ed5e8d9 and 48f7661.

2.7 Low Findings

2.7.1 Extreme composite pool prices can halt the keeper loop

Technical Details

`LynxCompositeStakingStrategy._holdLegWhbarValue()` values a hold token with two `FullMath.mulDiv()` calls. When the held token sorts after WHBAR, the inverse-price branch first calculates `sqrtPriceX96 * sqrtPriceX96 / 2^96` and uses that result as the denominator of the next division. A nonzero SaucerSwap price below `2^48` makes the intermediate result round down to zero, so `exposureValue()` reverts instead of returning a conservative value.

The supported Uniswap-style price range includes this input. The repository's `TickMath` defines the minimum valid square-root price as `4,295,128,739`, well below `2^48`. The composite deployment script resolves WHBAR hold-leg pools and configures them without a lower valuation bound.

`StrategyManager.pushSplit()` prechecks the exposure of both split strategies. The public keeper route calls `pushSplit()` from `allocateSurplus()`, and `weeklyOps()` calls that route

before its tail call to `_scheduleNext()`. A pool at the affected price therefore makes the complete weekly operation revert before the next HSS call is created. The previous HSS call is one-shot; the router only creates its successor after a successful `weeklyOps()` body. No contract path creates a replacement schedule after this revert. The stale `pendingSchedule` value also remains because the reversion rolls back the call that would clear it.

The condition is direction-dependent. It applies only to a configured hold leg whose HTS EVM address is numerically greater than the WHBAR address and that has a nonzero balance. The deployment flow does not enforce an address ordering or otherwise handle this branch's zero denominator.

Impact

Low. A party able to move a configured composite hold-leg pool below the arithmetic threshold can make the scheduled keeper run fail once. The loop has no successor schedule until an account later calls `weeklyOps()` after the price recovers, or the owner stops and starts the loop. This delays harvesting, buffer maintenance, and surplus deployment. Exploitability depends on the liquidity and attainable price range of the configured pool.

Recommendation

Make the inverse quote calculation safe when its rounded denominator is zero. A zero denominator can return zero WHBAR value, or the strategy can clamp the price to a configured minimum and return a bounded valuation. `weeklyOps()` should also isolate its scheduling step from market-sensitive maintenance so a failed body cannot stop autonomous execution.

```
uint256 denominator = FullMath.mulDiv(sqrtPriceX96, sqrtPriceX96, q96);
if (denominator == 0) return 0;
return FullMath.mulDiv(q96, tokenBal, denominator);
```

Developer Response

Fixed in commits e18070c and 5fb4c33.

2.7.2 Mainnet migration scripts use an incompatible manager ABI

Technical Details

The documented source `StrategyManager` is the mainnet contract `0.0.10510268`, deployed on June 3. It predates the July change from absolute strategy caps to BPS caps. Its ABI exposes `registerStrategy(address,uint256)` and `setCap(address,uint256)`.

The migration deployment script instead sends `registerStrategy(address,uint16)` and the cap script sends `setCapBps(address,uint16)`. These have different four-byte selectors, so the deployed source manager cannot dispatch either call. The registration transaction reverts before the migration strategy is registered.

The scripts do not check the registration receipt's success status before calling `writeDeployment()` and `upsertEnv()`. After a reverted registration, they still record the new

migration strategy as the configured strategy. The drain script then reads the legacy manager's absolute `cap` field as `capBps` and cannot pass its registration precheck.

Mainnet mirror-call validation from the documented owner address confirms the mismatch. Calls using the current selectors `0xe413d244` (`registerStrategy(address,uint16)`) and `0xcb39331f` (`setCapBps(address,uint16)`) revert with empty data. Calls using the legacy selectors `0xa8da683f` (`registerStrategy(address,uint256)`) and `0x80ad2cf3` (`setCap(address,uint256)`) reach the source manager and return its `ZeroAddress` and `UnknownStrategy` custom errors for a zero strategy address.

This is separate from the migration-cap finding. That finding begins after a migration strategy has been registered. This failure prevents the documented mainnet procedure from registering one at all.

Impact

Low. The documented mainnet migration procedure cannot start through the checked-in scripts. A failed deployment leaves a locally recorded but unregistered migration strategy, and the subsequent drain refuses to proceed. No attacker can exploit this path or move funds; the owner can recover by using a legacy-ABI-aware operational procedure and correcting the local deployment state.

Recommendation

Read and validate the source manager ABI before deploying a migration strategy. For `0.0.10510268`, use `registerStrategy(address,uint256)` and `setCap(address,uint256)` with an absolute WHBAR cap, then make the drain script interpret the legacy `strategyInfo` field accordingly.

Alternatively, deploy a compatible source manager and perform an explicitly reviewed manager migration before the vault cutover. In either case, check every transaction receipt for success before writing deployment files or environment entries.

Developer Response

Fixed in commit 5674410.

2.7.3 Replacing a funded hold leg hides the previous token

Technical Details

`configureHoldLeg()` overwrites a hold leg's token, pool, and fee without checking whether the previous token has a balance or whether the leg has recorded WHBAR cost basis. The reconfiguration script calls this setter directly and does not require an unwind first.

`exposureValue()` and `_swapHoldLegsToWhbar()` enumerate only the current `holdTokens` registry. After an owner replaces a funded leg, the previous token remains in `LynxCompositeStakingStrategy`, but normal valuation and withdrawal no longer read or sell it.

For example, deploying 100 WHBAR entirely into the first hold leg creates 100 units of the original hold token and records 100 WHBAR of basis. Replacing that leg with another token leaves the original 100-token balance unchanged. `exposureValue()` returns zero because the replacement token has no balance. A subsequent manager pull returns zero, reduces the strategy's principal by 100 WHBAR, and books the invisible position as a vault loss.

The owner can recover the position only by restoring the original configuration and unwinding it through a supported path. If the prior pool configuration is unavailable or forgotten, the token remains outside the normal strategy lifecycle.

Impact

Low. Reconfiguring a funded hold leg causes the vault to stop valuing and withdrawing the previous token. A later pull can permanently record the asset as lost even though it remains in the strategy. The issue requires an owner-initiated reconfiguration and is recoverable while the previous configuration and liquidity route remain available.

Recommendation

Reject a hold-leg replacement when the current token has a nonzero strategy balance or the leg has nonzero recorded basis, so the owner must unwind the leg through a supported path before reconfiguring it.

Developer Response

Fixed in commit 0acaa13.

2.7.4 Quorum tracks optional LynxVotes deposits instead of total share ownership

Technical Details

`LynxVotes` is an `ERC20Wrapper` over LynxVault shares. Its supply includes only shares that users have deposited through `depositFor()`. All deposited wrapper tokens count toward quorum, including undelegated tokens. Vault shares held directly, escrowed in `LynxHTSBridge`, represented by circulating HTS LYNX, or held by another integration do not count toward `LynxVotes.totalSupply()`.

`LynxGovernor` passes this wrapper to `GovernorVotesQuorumFraction`. The inherited quorum calculation is:

```
quorum = LynxVotes.getPastTotalSupply(snapshot) * quorumNumerator / 100
```

The defaults intended for production set `quorumNumerator` to 4 and `proposalThreshold` to zero. The intended user flow makes the supply gap routine: HTS holders must unwrap LYNX into vault shares, deposit those shares into `LynxVotes`, and delegate before they have binding votes. The bridge-held shares are deliberately not delegated.

An account can therefore become the full voting supply without owning a meaningful fraction of the protocol:

1. Almost all vault shares and LYNX remain outside `LynxVotes`.
2. The account deposits one vault-share base unit into `LynxVotes` and delegates it.
3. At the proposal snapshot, `LynxVotes.getPastTotalSupply()` is one.
4. Integer division makes the 4% quorum zero.
5. The account casts one delegated `for` vote, which is still required because the Governor also requires more `for` than `against` votes.
6. The account queues the proposal and executes it through the timelock.

The same issue exists above the rounding boundary. Quorum is always a percentage of the opt-in wrapper supply, not a percentage of total vault ownership. A small holder can pass proposals whenever most holders have not moved their claims into the voting wrapper before the snapshot.

The voting delay gives other holders time to opt in and oppose a proposal, but safety then depends on inactive LYNX holders monitoring every proposal and completing a multi-step conversion before the snapshot. The on-chain quorum does not enforce the documented 4% participation level against total protocol ownership.

The latest wired testnet stack demonstrates the condition with a nonempty vault. On July 23, 2026, its `LynxVotes` contract had zero total supply while the underlying vault had `166,000,000` share units. The Governor reported a 4% quorum numerator and a zero proposal threshold. No mainnet Governor deployment was verified.

Impact

Low. A holder with negligible economic ownership can pass governance proposals when voting-wrapper participation is low. The timelock can change LP/staking allocation, LP target enablement, staking weights, and inherited Governor settings. These permissions do not directly transfer protocol ownership, but they can redirect future deployment into riskier strategies or pools and weaken the voting rules for later proposals. The intended one-day snapshot delay, seven-day voting period, two-day timelock, and a retained `DEFAULT_ADMIN_ROLE` that can grant itself a canceller role provide a substantial intervention window. That limits likelihood and immediate impact, but the contract does not enforce meaningful voter participation.

Recommendation

Do not use the opt-in wrapper's supply as the only quorum denominator. Use a custom quorum that measures total economic ownership, including vault shares represented by HTS LYNX, or enforce an absolute quorum floor derived from the total vault-share supply. The total-supply value used at a proposal snapshot must itself be checkpointed.

Set a non-zero proposal threshold in the same economic units and apply an absolute minimum quorum so a near-empty wrapper cannot round quorum to zero. Until that change is deployed, keep a protocol-admin cancellation path and monitor the ratio between `LynxVotes.totalSupply()` and the underlying vault's total supply.

Developer Response

Fixed in commit 6ce3018.

2.7.5 Migration drain cap checks fail after capital is consolidated

Technical Details

The migration drain checks whether the source vault's initial idle WHBAR fits the migration strategy's cap. It then pulls every other strategy in separate transactions and pushes the larger post-pull idle balance into the migration strategy. The precheck does not prove that this final push will pass.

`StrategyManager._precheckPush()` requires the migration strategy's existing exposure plus the new push to be no larger than `vault.totalAssets() * capBps / 10,000`. The migration strategy values its exposure as its raw WHBAR balance. The vault's `totalAssets()` excludes locked profit.

The mismatch produces three failure modes:

1. A cap below 100% can pass the script's initial-idle precheck and always fail after the drain consolidates assets. With 100 idle WHBAR, 900 WHBAR in another strategy, and a 90% migration cap, the precheck accepts 100 against a 900-WHBAR limit. Pulling the other strategy leaves 1,000 idle WHBAR, but the subsequent push exceeds the unchanged 900-WHBAR limit.
2. A 100% cap fails whenever profit remains locked. With 1,010 raw WHBAR and 10 WHBAR of locked profit, `totalAssets()` is 1,000 WHBAR. The script pushes all 1,010 idle WHBAR and receives `CapExceeded`.
3. An account can transfer one raw WHBAR unit directly to the migration strategy. At a 100% cap, pushing the whole 1,000-WHBAR vault balance then compares 1,001 WHBAR of strategy exposure with a 1,000-WHBAR cap and reverts. The strategy is associated with WHBAR during deployment, so it can receive this transfer.

The script executes its pulls before the failing push. The source therefore remains paused with previously deployed capital consolidated as idle WHBAR, while the migration has not relayed any of that capital to the successor. An attacker can repeat the one-unit transfer after the operator clears it, extending the interruption throughout the migration window.

The runbook also labels `STRATEGY_CAP` as an idle-WHBAR base-unit amount. `set-strategy-cap.ts` treats that value as basis points and rejects values greater than 10,000, so the documented command fails for any normal vault balance.

Impact

Low. A migration can halt after earlier strategy pulls have committed. An attacker can make the failure repeatable for one WHBAR base unit whenever a full-cap migration strategy is registered and WHBAR-associated. No funds are stolen, and the owner can recover by clearing the excess balance, setting a 100% cap, and waiting for or clearing locked profit. The failure blocks a time-sensitive cutover and leaves the source in a partially consolidated state until recovery completes.

Recommendation

Require a 100% cap, zero locked profit, and zero unsolicited migration-strategy balance before starting the drain. Simulate the post-pull push amount before submitting any state-changing pull.

Fix the runbook to use `STRATEGY_CAP_BPS=10000`. For a durable fix, make `VaultMigrationStrategy.exposureValue()` report only manager-funded migration principal rather than raw token balance, or add an owner-controlled recovery path for unsolicited WHBAR that does not affect the cap calculation.

Developer Response

Fixed in commit 8b79b1a.

2.7.6 External fee recipient can block redemptions

Technical Details

`LynxVault._withdraw()` sends the exit fee to `feeRecipient` after the vault burns shares and transfers the user's net WHBAR. This is the common path for `withdraw`, `redeem`, `redeemHbar`, and bridge `redeemAndUnwrapHbar`.

WHBAR is an HTS token. A transfer to an account that is not associated with WHBAR reverts. `SafeERC20.safeTransfer()` bubbles that failure, so the fee payment reverts the complete redemption.

The unsafe configuration requires both of these owner-controlled changes:

1. `setExitFee()` sets a nonzero exit fee.
2. `setFeeRecipient()` selects an external account that cannot receive WHBAR.

The recipient can be unassociated when it has no automatic-association capacity, when its capacity is exhausted, or after it empties its WHBAR balance and dissociates. The vault can associate only itself with a token; it cannot associate an external fee recipient. A failed fee payment therefore blocks every asset exit until the owner changes the configuration.

The constructor is safe: it initializes the fee recipient to the vault and the exit fee to zero. Mainnet is also currently safe because the configured recipient remains the vault. Plain bridge `unwrap()` remains available, but the returned vault shares cannot be redeemed for WHBAR or HBAR while the bad fee configuration remains active.

Impact

Low. An owner-selected external fee recipient that cannot receive WHBAR blocks `withdraw`, `redeem`, native-HBAR redemption, and bridge redemption for all users while a nonzero exit fee is active. The owner can immediately restore the vault as recipient, set the fee to zero, or arrange association, so this is an exit-availability failure rather than a loss of funds.

Recommendation

Do not make user redemptions depend on a third party's HTS association state. Use OpenZeppelin's `SafeERC20.trySafeTransfer()` and retain a fee when its transfer fails, or accrue fees in a pull-based claim mechanism.

If fees are accrued for later claim, exclude the accrued liability from vault NAV and idle redemption liquidity. Otherwise a later fee claim removes assets that the vault already priced into user shares.

Checking association only when the recipient is configured is insufficient because that account can later dissociate.

Developer Response

Fixed in commit 4690a0b.

2.7.7 Unrestricted proposals can weaken allocation-governance safeguards

Technical Details

The governance playbook limits DAO proposals to allocation-management operations, and `proposal-guard.ts` describes its selector policy as an off-chain guard. `LynxGovernor`, however, does not enforce the documented target and selector restrictions. Its `_propose()` override delegates directly to OpenZeppelin and accepts arbitrary targets and calldata.

Because the Governor inherits `GovernorSettings`, `GovernorVotesQuorumFraction`, and `GovernorTimelockControl`, unrestricted proposals expose inherited `onlyGovernance` functions including `setVotingDelay()`, `setVotingPeriod()`, `setProposalThreshold()`, `updateQuorumNumerator()`, and `updateTimelock()`. A proposal that passes under the current rules can target the Governor itself and execute these calls through the timelock, which is the Governor's executor.

`LynxTimelock._execute()` does not prevent this path. It blocks four selectors only when the timelock itself is the target; calls targeting the Governor are unaffected. A successful proposal can consequently set quorum and voting delay to zero and reduce the timestamp-based voting period to one second. Later proposals can pass with substantially less participation and almost no reaction window. Alternatively, `updateTimelock()` can disconnect the Governor from the executor currently authorized by the protocol contracts.

The off-chain guard protects only scripts and interfaces that choose to invoke it. Any eligible proposer can call the Governor's on-chain `propose()` function directly and bypass those checks.

Impact

Low. A voting bloc capable of passing one proposal under the intended rules can weaken the rules for all subsequent proposals and preserve control with materially less participation. It can then influence staking/LP allocation, pool eligibility, and any other operation accepted from the timelock without the quorum and review window expected by token holders.

Recommendation

Enforce the documented target and selector allowlist on-chain in `LynxGovernor._propose()`. Validate each target-selector pair and reject Governor or timelock self-administration unless that operation is explicitly part of the intended DAO authority.

If governance parameters must remain mutable, expose only the required changes and enforce immutable minimums for quorum, voting delay, voting period, and proposal threshold. Retain `proposal-guard.ts` as a UI and operational safeguard, but do not rely on it as the authorization boundary.

Developer Response

Fixed in commit ce1d3ca.

2.7.8 Duplicate position serials in the LP allocator alias enabled state between deploy and drain

Technical Details

`LynxLpAllocator` stores push targets in the `targets` array and never enforces that a `positionSN` appears at most once. `setTargets()` validates each entry through `_validateTarget()`, which checks only that the serial is non-zero, the position is active, the pool fee matches, and the pair token is a pool leg. It does not reject a serial that is already present. The off-chain catalog that feeds `setTargets()` has the same gap: `loadLpPositionCatalog()` maps every `lp-pools.json` row that carries a `positionSN` into a target with no deduplication, so two rows with the same serial become two targets.

Two code paths read targets by array index, and two read them by first serial match, so a duplicated serial can carry conflicting state:

- `deployLpIdle()` and `setEnabledFlags()` iterate `targets` by index. `deployLpIdle()` computes `perTarget = whbarStart / enabledTargetCount()` and then deploys `perTarget` once per enabled index.
- `isTargetEnabled()` and `_resolvePairToken()` return the first array entry whose `positionSN` matches and stop.

This split produces two harmful states from the same missing invariant.

First, if the same serial is enabled twice, `enabledTargetCount()` counts it twice and the deploy loop calls `increaseLPPosition()` on that position on each matching index. The position receives a multiple of the intended equal split at the expense of the other positions, and each pass pays the SaucerSwap mint fee again. With targets `[SN=5 enabled, SN=5 enabled, SN=9 enabled]`, position 5 receives two thirds of idle WHBAR and position 9 one third, rather than an even split.

Second, if a serial appears once disabled and once enabled, the drain guard reads the wrong flag. `LynxKeeperRouter.reconcileInactiveLp()` skips a serial only when `isTargetEnabled(sn)` is true, and `drainPosition()` reverts with `TargetEnabledCannotDrain` only under the same

check. With targets ordered `[SN=5 disabled, SN=5 enabled]`, `isTargetEnabled(5)` returns the first entry's `false`, so `reconcileInactiveLp()` drains position 5 even though the second target keeps it enabled. `deployLpIdle()` then redeploys idle WHBAR into position 5 through the enabled index. `weeklyOps()` is permissionless, so each call runs a full drain-then-redeploy on the position: `drainPosition()` removes all liquidity and swaps the entire pair leg to WHBAR, and the following `deployLpIdle()` swaps WHBAR back to the pair and re-adds liquidity. Anyone can call `weeklyOps()` repeatedly to repeat the round trip.

Impact

Low. A duplicated-enabled serial skews the equal split and pays redundant mint fees, and a disabled-then-enabled serial lets the permissionless `weeklyOps()` path drain and redeploy an in-use position on every call, bleeding swap fees and slippage on the position notional until governance rewrites the target set. The effect is value leakage rather than direct theft. Both states require the owner/admin to register the same `positionSN` twice through `setTargets()`, which is owner-gated and normally sourced from a hand-edited manifest, so likelihood is low. `lp-catalog-verify` cannot flag the inconsistency because it also reads state through `isTargetEnabled(sn)`, which collapses duplicates to the first match.

Recommendation

Enforce a single entry per `positionSN` when the target set is written. Reject a duplicate in `setTargets()`, and add the same check to the off-chain catalog loader so the manifest cannot encode one.

```
for (uint256 i = 0; i < len; ++i) {
    PositionTarget calldata t = targets_[i];
    _validateTarget(t, whbar);
    for (uint256 j = 0; j < i; ++j) {
        if (targets_[j].positionSN == t.positionSN) revert DuplicatePosition(t.
positionSN);
    }
    targets.push(t);
}
```

Developer Response

Fixed in commit 26ac6c4.

2.7.9 Pausing the vault does not stop permissionless deployment of idle assets into strategies

Technical Details

`LynxVault.pause()` and `LynxHTSBridge.pause()` are the incident-response levers. Both are reachable by the guardian or the owner and both are scoped to inflows only: the vault gates `_deposit()` and `depositHbar()` with `whenNotPaused`; the bridge gates `wrap()` and `depositHbarAndWrap()`. Redemptions stay open by design. The documented runbook, `hardhat/scripts/ops/pause-stack.ts`, calls only these two `pause()` functions.

Capital deployment is a separate path that reads no pause state. `LynxKeeperRouter.pushAndDeploy()`, `allocateSurplus()`, `refillBuffer()`, and `weeklyOps()` carry no access modifier, so anyone can call them at any time, including while the vault is paused. `allocateSurplus()` pushes idle WHBAR above `effectiveTargetBuffer()` into strategies through `manager.pushSplit()`, and `pushAndDeploy()` then routes LP idle into SaucerSwap positions through the allocator. The weekly HSS schedule also fires `weeklyOps()` autonomously.

The only gate on the push side lives in `StrategyManager._precheckPush()`, which reverts `StrategyIsPaused` when the target strategy's per-strategy flag is set. That flag is controlled by `StrategyManager.setPaused()`, which is `onlyOwner`. `_precheckPush()` does not consult the vault's pause state.

Two consequences follow. First, a guardian who pauses the vault for fast incident response cannot stop capital from flowing into strategies, because the per-strategy pause is owner-only and the deploy entrypoints are permissionless. The guardian's lever halts deposits but not deployment. Second, when the incident is in a strategy, pausing the vault does nothing to keep the vault's idle WHBAR away from that strategy. Any account can call `pushAndDeploy()` or `allocateSurplus()` while the vault is "paused" and move surplus idle into the strategy under attack, up to that strategy's `capBps` limit. Stopping this requires the owner to call `setPaused(strategy, true)`, a step not present in the pause runbook, and `LynxKeeperRouter.stopWeeklyLoop()` only halts self-scheduling, not the public deploy functions.

Impact

Low. When the vault is paused during an incident, idle WHBAR can still be swept from the redemption buffer into strategies by any caller, including into a strategy that is the reason for the pause. The guardian, which is the fast-reaction role, cannot prevent this because deployment is halted only by the owner-only per-strategy pause. The moved funds land in protocol-owned strategies that governance can later recover rather than being stolen, and realizing harm requires a live incident together with idle above the target buffer, so likelihood is low. The gap is that the emergency pause does not freeze new capital-at-risk and the guardian's authority does not cover the deployment path.

Recommendation

Make the vault pause halt deployment as well as inflows so a single guardian-reachable action freezes new capital-at-risk. Have `StrategyManager._precheckPush()` revert when the vault is paused:

```
  function _precheckPush(address strategy, uint256 amount) internal view {
      _requireRegistered(strategy);
      _requireFresh(strategy);
      StrategyInfo storage info = strategyInfo[strategy];
      if (info.paused) revert StrategyIsPaused(strategy);
+     if (IVaultPause(vault).paused()) revert StrategyIsPaused(strategy);
```

If deployment should stay under owner control separate from inflows, give the guardian authority to call `StrategyManager.setPaused()` and add a strategy-pause step to `pause-stack.ts` so the incident runbook freezes both inflows and deployment.

Developer Response

Fixed in commit ae19320.

2.7.10 Undeployed staking split slices unlock as false vault yield

Technical Details

`StrategyManager._push()` records the full pushed amount as strategy principal before it transfers WHBAR to `LynxCompositeStakingStrategy.deposit()`. The composite strategy delegates allocation to `LynxStakingAllocator.deploySplit()`, which computes per-leg amounts from `stakingWeightsBps`.

`deploySplit()` silently skips any nonzero leg amount below `minLegDeploy`. The skipped WHBAR remains on the composite strategy contract. It is still included in `StrategyManager.principal`, because the manager recorded the full push, but it is not added to `nativePrincipal` through `unwrapToNative()` and it is not added to `holdPrincipalWhbar` through `swapFromWhbar()`.

On the next composite `harvest()`, the strategy computes:

```
gain = WHBAR/native balance - nativePrincipal
```

The skipped WHBAR balance exceeds `nativePrincipal`, so the strategy transfers it to the vault and reports it as gain. `StrategyManager.harvest()` then calls `vault.reportGain(gain)` without reducing `principal[strategy]`. While the gain is locked, `lockedProfit` masks the double count. After the profit unlock period decays, the same skipped slice is counted once as idle WHBAR in the vault and once as still-deployed manager principal.

The default deployment sets `STAKING_MIN_LEG_DEPLOY_WHBAR` to `0.01`, so a 1 bps leg in a 10 HBAR push leaves `0.001` HBAR undeployed. The Foundry PoC uses that default: the `0.001` HBAR slice is harvested as gain, manager principal remains 10 HBAR, and `totalAssets()` rises by `0.001` HBAR after the lock expires.

This is the same class of defect as issue `LynxCompositeStakingStrategy withdrawals can turn remaining HBAR principal into fake harvest prof`. Both let `harvest()` report phantom gain when the strategy's `nativePrincipal` drifts out of sync with the manager's recorded principal. The referenced issue creates that drift on the withdrawal path; this creates it on the deposit path, where a skipped split slice is never recorded as strategy principal.

Impact

Low. Split dust can be reported as realized yield even though it is principal that was already counted by `StrategyManager`. After `profitUnlockPeriod`, vault NAV is overstated by the skipped amount, which can let redeemers receive slightly more than the vault's real assets support. With the documented default `minLegDeploy` of `0.01` HBAR, each skipped slice is small, so impact is bounded unless governance configures a larger threshold or repeatedly creates skipped slices over many pushes.

Recommendation

Do not leave skipped split slices untracked. Either fold any below-threshold amount into a deployed leg, revert when a nonzero slice is below `minLegDeploy`, or track skipped WHBAR as composite strategy principal before returning from `deploySplit()`.

For example, add an allocator-only accounting hook for retained WHBAR principal:

```
function retainWhbarPrincipal(uint256 amount) external onlyAllocator {
    if (amount == 0) revert ZeroAmount();
    nativePrincipal += amount;
}
```

Then call it for each skipped amount, or once with the total skipped amount. The exact implementation can differ, but the invariant should be that every pushed base unit is either deployed into a leg and recorded as principal, returned to the vault with a matching manager principal reduction, or causes the push to revert.

Developer Response

Fixed in commit 06136d5.

2.7.11 Native HBAR sent to the vault is stranded and cannot be recovered

Technical Details

Native HBAR is used in only one place in `LynxVault`, the `payable` `depositHbar()`, which wraps its own `msg.value` into WHBAR through `_wrapHbar()`. Every other asset flow uses the WHBAR token. The vault does not otherwise need to accept native HBAR, yet it still exposes a bare payable `receive()` that silently accepts it.

Impact

Low. Native HBAR sent to the vault is basically lost. It is excluded from `totalAssets()`, backs no shares, and cannot be recovered on-chain.

Recommendation

Remove `receive()` and let native transfers revert.

Developer Response

Fixed in commit 11cb467.

2.8 Gas Savings Findings

None.

2.9 Informational Findings

2.9.1 Core contracts lack renewal funding and expire at a fixed date

Note Hedera's documented policy is that an unfunded contract expires and is eventually deleted, but to date Hedera has not enforced contract expiry on mainnet. This is filed as Informational on that basis. It is worth staying on top of, because Hedera could change course and begin enforcing expiry, at which point the unfunded state described below would put the core contracts at risk.

Technical Details

Hedera contracts are backed by cryptocurrency accounts. At each auto-renewal, the network charges the contract account's native HBAR balance. Hedera documents that an account with insufficient HBAR is extended only as far as its balance permits and that an empty account is deleted when it expires. See ContractGetInfo.

The deployment helper creates a `ContractCreateTransaction` with bytecode, gas, and an optional admin key, but never configures the transaction's `initialBalance` or `autoRenewPeriod`. The vault, manager, bridge, and SaucerSwap strategy deployment scripts all use this helper. The optional vault seed is sent to `depositHbar`, which wraps the supplied HBAR into WHBAR; it does not retain native HBAR in the contract account for renewal. Hedera exposes `initialBalance` and `autoRenewPeriod` on `ContractCreate`; see ContractCreate.

Mainnet mirror-node data queried on 2026-07-23 shows that all core contracts have no `auto_renew_account`, a 7,776,000-second auto-renew period, and a zero native-HBAR account balance:

Contract	Contract ID	Expiration timestamp	Native HBAR balance
Vault	`0.0.10510261`	`1788237335.178343703`	`0` tinybar
Strategy manager	`0.0.10510268`	`1788237464.005748000`	`0` tinybar
HTS bridge	`0.0.10510271`	`1788237615.585799000`	`0` tinybar
SaucerSwap LP strategy	`0.0.10510275`	`1788237965.963496000`	`0` tinybar

Those timestamps fall on 2026-08-31. Unless an operator extends the contracts or provides a renewal balance before then, the expiration process makes the protocol's core contracts unavailable. This is not a generic monitoring concern: the checked-in deployment path creates the exact unfunded state present on mainnet.

The expiration can be extended before the deadline. Hedera permits an expiration-only `ContractUpdate` signed by the paying account, so this finding does not assume that a failed renewal is permanently unrecoverable. See ContractUpdate.

Impact

The mainnet vault, strategy manager, bridge, and active LP strategy carry a deterministic expiry date (~2026-08-31) with no renewal funding source. Hedera does not currently enforce contract expiry, so there is no immediate outage. If the network begins enforcing it and operators do not extend the contracts first, deposits, redemptions, strategy operations, and the bridge path could become unavailable while user assets are held in the affected contracts.

Recommendation

Before 2026-08-31, submit expiration-only `ContractUpdate` transactions for every deployed core contract and verify their new expiration times on a mirror node.

Update the deployment helper to establish a renewal policy for each contract: provide a dedicated native-HBAR renewal balance and fail deployment when the resulting expiration horizon is below the operational threshold. Add monitoring that alerts well before the renewal balance or expiration horizon becomes unsafe.

Developer Response

Fixed in commit 065fa37.

2.9.2 Redeem rounding underpays an external fee recipient

Technical Details

`previewRedeem()` calculates a user's net assets by subtracting `_feeOnTotal(grossAssets, exitFeeBps)` from the gross asset value of redeemed shares. `_withdraw()` then calculates the fee transfer separately as `_feeOnRaw(netAssets, exitFeeBps)`.

Those rounded calculations are not always equal. At a one-BPS fee, redeeming shares worth 10,002 raw WHBAR produces:

```
gross assets:                    10,002
fee deducted by previewRedeem:        2
net assets returned to user:     10,000
fee transferred by _withdraw:         1
residue retained by vault:            1
```

The user receives the amount quoted by `previewRedeem()`, but an external `feeRecipient` receives one raw WHBAR unit less than the fee reflected in the burned shares. The residue remains in vault NAV and accrues to remaining shareholders. `withdraw()` does not have this discrepancy because it starts from the requested net asset amount and uses `_feeOnRaw()` for both share calculation and payment.

The default configuration leaves fees in the vault because `feeRecipient == address(this)`, so the discrepancy is economically invisible until the owner selects an external recipient and a nonzero fee.

Impact

Informational. Each affected `redeem`, `redeemHbar`, or bridge `redeemAndUnwrapHbar` call underpays an external fee recipient by at most one WHBAR base unit. A caller can split redemptions to create multiple discrepancies, but transaction costs exceed the one-unit amount and users do not receive the residue directly. The issue is inactive under the current vault-recipient configuration.

Recommendation

Pay the fee recipient the exact difference between the burned shares' gross value and the net assets paid to the user. That amount always equals the fee `previewRedeem()` (and `previewWithdraw()`) quoted, so the quote and the transfer agree by construction and no rounding residue is left in NAV.

```
 function _withdraw(
     address caller,
     address receiver,
     address owner,
     uint256 assets,
     uint256 shares
 ) internal override {
-    uint256 fee = _feeOnRaw(assets, exitFeeBps);
+    // Fee is the gross value of the burned shares minus the net assets paid out,
+    // so the quote and the actual fee always agree and nothing is stranded.
+    uint256 gross = super.previewRedeem(shares);
+    uint256 fee = gross > assets ? gross - assets : 0;
     address recipient = feeRecipient;
```

`previewRedeem` continues to return `assets - _feeOnTotal(...)`, so the user still receives exactly the quoted net, while the fee recipient now receives exactly `gross - net`.

Developer Response

Fixed in commit 81870a3.

2.9.3 Paused vault reports unlimited deposit and mint capacity

Technical Details

`LynxVault` pauses every ERC-4626 deposit and mint through the `whenNotPaused` modifier on `_deposit()`. It does not override OpenZeppelin's `maxDeposit()` and `maxMint()`, which always return `type(uint256).max`.

When the vault is paused, an integrator can therefore read unlimited deposit and mint capacity, prepare an action that is within that advertised limit, and receive `EnforcedPause` when it calls `deposit()` or `mint()`.

ERC-4626 requires `maxDeposit()` and `maxMint()` to return a limited value when the vault temporarily disables the corresponding operation. The pause is a deposit and mint limit, but the inherited view functions do not represent it.

Impact

Informational. ERC-4626 integrations cannot use `maxDeposit()` or `maxMint()` to detect the vault's emergency or migration pause. Their transactions revert after the limit check passes. The behavior does not transfer funds or grant privileges.

Recommendation

Override both limit views to return zero when the vault is paused.

```
function maxDeposit(address receiver) public view override returns (uint256) {
    return paused() ? 0 : super.maxDeposit(receiver);
}

function maxMint(address receiver) public view override returns (uint256) {
    return paused() ? 0 : super.maxMint(receiver);
}
```

Developer Response

Fixed in commit 000eb93.

2.9.4 Default mainnet staking manifest contains testnet hold-token IDs

Technical Details

The default mainnet composite staking manifest (`staking-tokens.json`) appears to still contain testnet/stale token IDs for two of the three non-native hold legs:

```
{ "id": "hbarx", "symbol": "HBARX", "tokenId": "0.0.2231533" }
{ "id": "sauce", "symbol": "XSAUCE", "tokenId": "0.0.1418651" }
```

Both IDs are valid on Hedera testnet, but not on Hedera mainnet:

```
mainnet 0.0.2231533 -> 404 / no token
testnet 0.0.2231533 -> HBARX, 8 decimals

mainnet 0.0.1418651 -> 404 / no token
testnet 0.0.1418651 -> XSAUCE, 6 decimals
```

The intended mainnet IDs appear to be HBARX `0.0.834116` and XSAUCE `0.0.1460200`. The mainnet mirror node returns those rows as HBARX / HBARX and xSAUCE / XSAUCE, respectively. SaucerSwap's deployment docs also list `0.0.1460200` as the mainnet xSAUCE token ID, while listing `0.0.1418651` as testnet xSAUCE.

The deploy script uses `hardhat/config/staking-tokens.json` by default on `hederaMainnet`. It converts each `tokenId` directly into a Solidity address, associates WHBAR and every hold token with the deployed composite strategy, resolves SaucerSwap pools for each hold token, and configures each hold leg. `tokenToEvm()` is only a syntactic conversion from `0.0.x` to an EVM address; it does not validate that the token exists on the selected network.

As a result, a mainnet deployment using the default manifest can create the composite strategy and allocator, then fail during hold-token association or pool resolution. If an operator resumes the script after editing around the failed step, the same wrong-network addresses could still be configured as hold legs.

Proof of Concept

This is a configuration finding rather than a Solidity exploit. It can be reproduced with current mirror-node lookups:

```
curl -fsS 'https://mainnet-public.mirrornode.hedera.com/api/v1/tokens/0.0.2231533'
# 404

curl -fsS 'https://testnet.mirrornode.hedera.com/api/v1/tokens/0.0.2231533'
# token_id 0.0.2231533, name Hbarx, symbol HBARX, decimals 8

curl -fsS 'https://mainnet-public.mirrornode.hedera.com/api/v1/tokens/0.0.1418651'
# 404

curl -fsS 'https://testnet.mirrornode.hedera.com/api/v1/tokens/0.0.1418651'
# token_id 0.0.1418651, name xSAUCE, symbol XSAUCE, decimals 6
```

Impact

Informational. This is a deployment-readiness/configuration issue, not an attacker-controlled path. A mainnet deployment that relies on the default manifest can fail after the composite strategy and allocator have already been created, requiring cleanup or a resumed deployment with corrected IDs. If the team treats this file as a placeholder until launch, the risk is accidental use of the placeholder in a production deploy.

Recommendation

Before mainnet composite staking deployment, replace the two rows with the intended production IDs, currently HBARX `0.0.834116` and XSAUCE `0.0.1460200`, or remove the default mainnet manifest until the production IDs are finalized.

Add a preflight validation step before contract creation that queries the selected network's mirror node for every non-null `tokenId`, verifies symbol and decimals, rejects duplicate hold tokens, and fails before deploying if any token or pool is missing.

Developer Response

Fixed in commit 3afca19.

2.9.5 Duplicate composite hold tokens are double-counted in exposure

Technical Details

`LynxCompositeStakingStrategy` lets governance configure each hold leg independently, but `setHoldLegs()` and `configureHoldLeg()` do not reject duplicate token addresses. If the same token is assigned to more than one hold slot, the strategy reads the same token balance once per slot.

`swapFromWhbar()` records hold-leg basis by calling `_holdLegIndex(tokenOut)`. That helper returns the first matching slot, so a swap into a later duplicate token still increments the first slot's `holdPrincipalWhbar`. The later slot receives no basis.

`exposureValue()` then loops across all hold slots and calls `_holdLegWhbarValue(i)`. Each duplicate slot executes `IERC20(token).balanceOf(address(this))` for the same token and values the full shared balance. With two duplicate slots, the same held tokens are counted twice; with three, they are counted three times.

The current testnet staking manifest configures `HLQT` for both the `hbarx` and `sauce` stand-in legs (`staking-tokens.testnet.json`), so a deployment from that manifest can hit this path. The issue is not limited to that file: the contract has no uniqueness invariant, so any future duplicate hold-leg configuration has the same accounting behavior.

Impact

Informational. This is a governance configuration foot-gun. If the same token is assigned to two hold slots, `exposureValue()` double-counts the shared balance, which can block later pushes through the manager cap check and stop `bookLossToExposure()` from booking a real hold-leg loss.

Recommendation

Reject duplicate hold-token addresses when configuring hold legs. `holdTokens` is a fixed array of `HOLD_LEG_COUNT` slots, so both setters can check against it directly.

`setHoldLegs()` writes all slots at once, so reject any address that repeats within the incoming array:

```
for (uint256 i = 0; i < HOLD_LEG_COUNT; ++i) {
    if (tokens_[i] == address(0)) revert ZeroAddress();
    for (uint256 j = i + 1; j < HOLD_LEG_COUNT; ++j) {
        if (tokens_[i] == tokens_[j]) revert DuplicateHoldToken(tokens_[i]);
    }
}
```

`configureHoldLeg()` writes one slot at `index`, so reject the new token if it already sits in another slot:

```
for (uint256 j = 0; j < HOLD_LEG_COUNT; ++j) {
    if (j != index && holdTokens[j] == token) revert DuplicateHoldToken(token);
}
```

The testnet manifest also assigns HLQT to two legs; give each leg a distinct token there so a testnet deployment does not trip the same path.

Developer Response

Fixed in commit ccfe23b.

2.9.6 Public deploy calls spend the router's HBAR reserve on LP mint fees

Technical Details

`LynxKeeperRouter`'s deploy functions (`pushAndDeploy`, `allocateSurplus`, `refillBuffer`) are public. When a deploy sends WHBAR into a SaucerSwap LP position, the router pays the SaucerSwap mint fee, about one HBAR per enabled pool, from its own native balance through `_fundLpMintFeeIfNeeded()`. That balance is the same reserve that funds its HIP-1215 scheduled runs. Since anyone can trigger a deploy, someone could repeatedly deposit to create surplus and call the deploy, making the router spend that reserve on mint fees until it can no longer fund its own scheduling.

Impact

Informational. This is bounded and self-costly. The caller pays gas and an exit fee each cycle, gains none of the spent HBAR, and the reserve is topped back up by the owner. The protocol loses about one HBAR per pool per cycle at current fee rates, and how much HBAR sits in the reserve is a config choice.

Recommendation

Consider having the public deploy path take the mint fee from the caller as `msg.value`, and keeping a separate keeper-only path that spends the router's own reserve. A caller who triggers a deploy then pays its own fee, while the scheduled automation still funds itself.

Developer Response

Fixed in commit c21e813.

2.9.7 Strategy-reported gains are booked without verifying delivery

Technical Details

`StrategyManager.harvest()` trusts the gain a strategy reports and forwards it to `reportGain()` without checking that the strategy actually delivered that WHBAR to the vault. Nothing measures the vault's balance to confirm the gain arrived.

Yearn's `Vault.vy` `report()` function guards against exactly this. It refuses to book a gain the strategy does not actually hold.

```
# No lying about total available to withdraw!
assert self.token.balanceOf(msg.sender) >= gain + _debtPayment
```

(Source: Yearn v2 `Vault.vy`, `report()`). Lynx has no equivalent check.

If a reported gain is wrong, from a strategy bug or a bad strategy, locked profit grows without the assets to back it. Across harvests it can exceed the vault's real balance, so `totalAssets()` floors to zero while shares are still outstanding. Share pricing then breaks. The next deposit mints a huge share count, and existing redemptions return dust. Strategies are trusted and governance-registered, so this needs a strategy to misbehave, not an outside attacker.

Impact

Informational. No problem exists in the current code. This is a precaution for the future, as new strategies are added, in case one has a bug or is compromised. If a trusted strategy over-reports gain, it can drive `totalAssets()` to zero and break share pricing. The path needs a strategy to misbehave, not an unprivileged caller.

Recommendation

Add a sanity check like Yearn's. Before booking a gain, confirm the strategy actually delivered it, the simplest way being to measure the vault's WHBAR balance before and after the strategy call and book only the increase rather than trusting the returned number.

Developer Response

Fixed in commit 9186fd1.

2.9.8 The timelock delay cannot be updated after deployment

Technical Details

OpenZeppelin's `TimelockController.updateDelay()` accepts calls only from the timelock itself. Under the standard implementation, changing the delay therefore requires scheduling and executing an operation whose target is the timelock and whose calldata invokes `updateDelay()`.

`LynxTimelock._execute()` blocks precisely that execution path:

```
if (target == address(this) && data.length >= 4) {
    bytes4 selector = bytes4(data[:4]);
    if (
        selector == _GRANT_ROLE || selector == _REVOKE_ROLE || selector == _RENOUNCE_ROLE
            || selector == _UPDATE_DELAY
    ) {
        revert TimelockSelfAdministrationBlocked(selector);
    }
}
```

A direct call from an external administrator reverts in the inherited `updateDelay()` because its caller is not the timelock. A scheduled self-call passes that caller check but reverts first in `_execute()` because its selector is `_UPDATE_DELAY`. Consequently, the constructor value of `minDelay` can never be changed after deployment, regardless of which accounts hold administrative roles.

This may be intentional hardening, but it differs from the administration model exposed by the inherited OpenZeppelin API.

Impact

Informational.

Recommendation

If the delay is intended to be permanently immutable, document this invariant explicitly in the contract NatSpec and governance runbooks.

If delayed governance updates are required, remove `_UPDATE_DELAY` from the blocked selectors. To prevent governance from weakening the timelock below an acceptable safety window, override the update path with an immutable minimum delay rather than disabling all updates.

Developer Response

Fixed in commit 08041e4.

2.9.9 LP drain implicitly sweeps the strategy's entire pair-token balance

Technical Details

`LynxLpAllocator.drainPosition()` drains one disabled LP position by calling `SaucerSwapLiquidityStrategy.decreaseLPPosition()` for the selected serial. After the decrease, however, it reads the strategy's full `pairToken` balance and swaps all of it to WHBAR through `swapToWhbar()`.

The balance read is not scoped to the position being drained:

```
uint256 pairBal = IERC20(pairToken).balanceOf(strategyAddr);
if (pairBal > 0) {
    lpStrategy.swapToWhbar(pairToken, poolFee, pairBal, swapMinOut_, deadline);
}
```

The LP decrease collects tokens into the strategy account. Because the function does not snapshot the balance before that operation, any pre-existing balance of the same pair token is indistinguishable from the amount produced by the selected drain. Such a balance can come from fee collection, residual dust, previous operations, direct transfers, or donations.

Consequently, a position-scoped action performs an implicit strategy-wide sweep of that token. This does not remove liquidity from other LP NFTs; it affects only fungible tokens already held directly by the strategy.

Impact

Tokens not produced by the disabled position can be converted unexpectedly, while `PositionDrained` attributes the operation to a single `positionSN`. This makes the function's scope and emitted accounting less precise.

Recommendation

If `drainPosition()` is intended to be position-scoped, snapshot the pair-token balance before decreasing liquidity and swap only the post-operation delta. Handle pre-existing balances through a separate, explicitly named sweep operation. If sweeping the entire balance is intentional, document that strategy-wide behavior and emit an event that clearly reports the full amount swept.

Developer Response

Fixed in commits 339bd16 and 9dce0ff.

The full-balance sweep is retained by design; the change makes it explicit and observable rather than implicit. `drainPosition()` was split into `_decreaseDisabledPosition()` and `_swapPairBalanceToWhbar()`; its NatSpec now states that the swap covers the strategy's entire balance of that pair token — fees, dust, residuals and donations included — and `PositionDrained` now emits `pairBalanceBefore` alongside `pairSwapped`, so the portion attributable to the drained position can be derived on-chain. Together with the `onlyOpsExecutor` gate and the quote-derived slippage bound now applied to the swap, the reported accounting opacity is resolved.

2.9.10 HSS failures can leave an enabled loop without a pending schedule

Technical Details

`startWeeklyLoop()` and `startWeeklyLoopAndRun()` set `loopActive = true` before attempting to create the first HSS schedule. `_scheduleNext()` also runs after every `weeklyOps()` execution while the loop is enabled.

`_scheduleNext()` clears `pendingSchedule` and then calls `HssOps.scheduleCall()`. If HSS returns an error or a zero schedule address, the function emits `ScheduleFailed` but otherwise returns normally. As a result, both an initial scheduling failure and a later tail-rescheduling failure can leave:

```
loopActive      == true
pendingSchedule == address(0)
```

This state means that the loop remains enabled by configuration but has no future autonomous execution pending. The router is not permanently deadlocked: `weeklyOps()` is public and calls `_scheduleNext()` whenever `loopActive` is true, so any account can retry after the HSS failure condition clears. The owner can also stop and restart the loop, and repository monitoring detects a missing pending schedule and watches `ScheduleFailed` events.

However, the available permissionless retry requires executing the complete weekly operations pipeline, including harvests, buffer refill, inactive-position reconciliation, and surplus deployment, even when the caller only intends to restore scheduling. Initial activation also reports an enabled loop despite never having successfully created its first schedule.

Impact

Informational.

Recommendation

Make `_scheduleNext()` return whether schedule creation succeeded. During initial activation, set `loopActive` only after the first schedule has been created successfully. If tail rescheduling fails, expose a dedicated `retrySchedule()` entry point callable only when `loopActive == true && pendingSchedule == address(0)`. The retry should attempt only schedule creation rather than execute the complete weekly operations pipeline.

Developer Response

Fixed in commit 908d29b.

LYNX Token Contracts

Completed 2026-07-23