Phage
SECURITY

— SECURITY REVIEW · PREPARED FOR

Perpgame

DATE

17.06.2026

CONDUCTED BY

Pyro, Lead Security Researcher
BengalCatBalu, Lead Security Researcher
MarketerKA, Security Researcher

COMMIT

34c47cb

REMEDIATION

7dcfba6

Table of Contents

About Phage Security

Phage Security is a team of highly skilled smart contract security researchers collaborating with 10+ proven freelancers who have excelled in public contests and private audits alike. With numerous vulnerabilities uncovered and patched across our completed audits, we strive to deliver the absolute best security service and client experience. While 100% security can never be guaranteed, we are committed to giving our utmost effort to protect your protocol.

Check out our previous work at PhageSecurity.com or reach out on X to Pyro.

Disclaimer

Audits are a time, resource, and expertise-bound effort where trained experts evaluate smart contracts using a combination of automated and manual techniques to identify as many vulnerabilities as possible. Audits can reveal the presence of vulnerabilities **but cannot guarantee their absence**.

System overview

Perpgame lets users buy into a launched token by paying USDC, minting ERC-20 shares priced off the net asset value of a basket of Hyperliquid leveraged tokens. Sellers burn shares to redeem their pro-rata slice of the basket back to USDC, and a rebalancer sets target weights and rebalances the basket through Bounce.tech. The platform takes a 1% fee on every buy and sell, and a scaling premium on buys accrues to existing holders.

Overall Security Posture

Before the fix branch the rebalance and shared pool accounting carried a High and most of the Mediums that could freeze a fund or shift value between holders, and after remediation the protocol is materially hardened with the High and all but two Mediums resolved (the 2 left out are Acknowledged).

Before remediation

Severity distribution: `1 High, 12 Medium, 13 Low`. Almost everything clustered in `AgentTreasury` — specifically the rebalance step, the deploy and withdraw accounting, and the symbol bookkeeping — with a handful in the `AgentCurve` bonding-curve math.

- **Rebalance and curve math sampled NAV, rates, or supply at the wrong moment or scale** (M-01, M-02, M-03, M-04, M-07, M-08, L-01) — the rebalance step sized buys and sells against pre-fee, pre-checkpoint, or pre-sell values and mismatched USDC's 6-decimal scale with the 18-decimal buffer, so routine rebalances reverted or drifted off target. Async-redeemed USDC was re-deployed by weight on the next buy, undoing the rebalance, and the curve priced its premium at `supplyBefore`, letting one large buy pay less than many small ones.
- **A single misbehaving leg could freeze or brick the whole fund** (M-05, M-09, M-10, M-11, L-06) — one unpriceable LT price view froze every NAV read and blocked rotation, one stuck Bounce redemption left `rebalanceInFlight` set and halted all buys and sells, and an unbounded

symbols array or factory scan opened a DoS. The contracts had no admin recovery for a stranded leg and no protocol-wide stop.
- **Shared-NAV entry and exit transferred value between co-investors** (H-01, L-04, L-05, L-11, L-12) — the only High let a duplicate LT under two symbols double-count NAV so the rebalancer could drain co-investors, and idle-first seller payouts overpaid the first exiter while pushing streaming fees onto remaining holders. A launch could also be configured with a near-confiscatory premium, a paid buy could mint zero shares, and edge paths silently ignored a buyer's per-leg slippage floor.

After remediation

Fixed 20 of 26; the remaining 6 are acknowledged (2 Medium, 4 Low), none of them loss-of-funds. Several patches went beyond the literal bug. A separate `depositIdleUsdc` balance and a `_navWithPendingRedemptions()` view that keep redemption proceeds and in-flight redemptions out of weight-based deploys, a new recovery surface (`cancelRedeem`, `sweepDust`, `migrateLt`) for stranded legs, a factory wide `paused` stop, a `MAX_ASSETS = 20` cap, and a `ltExists` check replacing an unbounded catalog scan. Core fund-loss and fund-freeze classes are closed, but a 1% buy and sell fee and a `returnLts` sell option landed after this fix set, so run one regression pass over the merged branch, including those additions, before mainnet given how tightly these fixes interact.

Security Recommendations

- **Bug bounty.** Stand up a bounty on Immunefi or HackenProof, with the top tier covering fund loss.
- **Active invariant monitoring.** Run an off-chain watcher that reconciles each treasury against its expected invariants and pages on divergence:
  - `AgentTreasury.nav()` should equal the sum of each leg's `lt.ltToBaseAmount(balanceOf)` plus idle `USDC.balanceOf` — a gap flags a mispriced or double counted legs.
  - `AgentTreasury.depositIdleUsdc` should never exceed the treasury's `USDC.balanceOf` — deposit-sourced idle is a subset of total idle, so a higher value means redemption proceeds were booked as deposits and would be wrongly redeployed by weight.
  - `rebalanceInFlight` being true should coincide with at least one leg holding `lt.userCredit(treasury) > 0` — true with no credits is the stuck-flag state.
- **PK hygiene.** The `TreasuryFactory` owner deploys every treasury, flips the global `paused` stop, and can call `rescue()`, so it is the highest-value key and belongs on a multisig with hardware signers. Each treasury's `rebalancer` sets weights and executes rebalances — the platform agent key for agentic tokens and the creator for classic — and should use hardware custody with a documented rotation and incident runbook. Both roles already use two-step handover (`Ownable2Step` and `proposeRebalancer` / `acceptRebalancer`) and the system has a global pause, but the access-control model does not bound what a rebalancer can do to the weights inside a single basket, so key custody is the main residual control there.

Technical Overview

Perpgame Contracts (HyperEVM / Solidity 0.8.28)

Three contracts, one stateless library, and one external interface. `AgentTreasury` is upgradeable behind a beacon proxy; `AgentCurve` and `TreasuryFactory` are not upgradeable, and `Treasury`

`Valuation` is a stateless library linked into `AgentTreasury` at deploy time.

- **TreasuryFactory** — owner-only factory that CREATE2-deploys each token's treasury proxy, funded by a USDC permit. Holds the global `paused` kill switch and a `rescue()`, and uses `Ownable2Step` for owner handover.
- **AgentTreasury** — holds the basket's Bounce LTs and runs deploy, withdraw, and rebalance under a single `rebalancer` role. Upgradeable via beacon with append-only storage plus a `__gap`, capped at `MAX_ASSETS = 20` legs, and gated by the factory's global pause. Its read-only valuation and quote logic is delegatecall-linked from the `TreasuryValuation` library to keep the deployed implementation under the EIP-170 code-size limit.
- **TreasuryValuation** — stateless library holding `AgentTreasury`'s read-only valuation helpers (`nav`, `quoteWithdrawUsdc`, held-LT values, NAV-with-pending-redemptions, and the redeem-buffer check). Extracted and DELEGATECALL-linked solely to shrink `AgentTreasury`'s bytecode below the EIP-170 24,576-byte runtime limit, which the in-contract version exceeded. It owns no storage and executes in the treasury's context (`address(this)` is the treasury and the storage pointers address its slots), so behavior is identical to the former in-contract methods.
- **AgentCurve** — ERC-20 share token with a NAV-anchored bonding curve where `buy()` mints at `nav / supply` times a premium and `sell()` burns and redeems pro-rata, with a 1% fee skimmed on each buy and sell to a hardcoded `FEE_RECIPIENT`. The premium add-on is capped at `MAX_EXTRA_PREMIUM = 4e18` (at most 5x), and the treasury link is immutable and set at construction.

Three privileged roles operate the protocol:

- **TreasuryFactory owner** — deploys treasuries, sets the global `paused` stop, and can `rescue()` factory-held tokens. Platform-held, should be a multisig.
- **rebalancer** — per-treasury key that sets target weights and executes rebalance steps, held by the platform agent for agentic tokens and the creator for classic.
- **CREATOR** — receives dust and stranded LT balances swept through `sweepDust` and `migrateLt`, set to the token creator.

Executive summary

Overview

Project Name	PerpGame
Repository	https://github.com/perpgame/contracts
Commit hash	34c47cb45268b823ca80e140d19c38a073948d0f
Remediation	7dcfba6df0294384dd685bee16539b3e0565852e
Methods	Manual review

Timeline

Audit kick-off	09.06.2026
End of audit	12.06.2026
Remediations start	12.06.2026
Remediations end	16.06.2026

Scope

contracts/src/AgentCurve.sol

contracts/src/AgentTreasury.sol

contracts/src/TreasuryFactory.sol

contracts/src/TreasuryValuation.sol

Findings Summary

1 High · 12 Medium · 13 Low

ID	Title	Severity	Status
[H-01]	Duplicate LT across symbols double-counts NAV and lets the rebalancer drain co-investors	High	Fixed
[M-01]	Async redeemed rebalance USDC Is re-deployed by weight on the next buy, undoing the rebalance	Medium	Fixed
[M-02]	Deploy remainder reverts most of the buys	Medium	Fixed
[M-03]	Rebalance buy sizes targets against a stale navAtStart after redeems change the rates	Medium	Fixed
[M-04]	Buy premium is priced at supplyBefore, so one large buy pays less than many small ones	Medium	Acknowledged
[M-05]	One stuck Bounce redemption freezes all buys and sells for the whole fund	Medium	Fixed
[M-06]	Deploy at low minBps leaves buyer capital undeployed as sells drain Idle below the threshold	Medium	Acknowledged
[M-07]	Normal rebalances may revert	Medium	Fixed
[M-08]	`executeRebalanceStep` sell path always triggers ‘lt.redeem‘ does to a wrong comparison	Medium	Fixed
[M-09]	‘_isFactoryLt‘ scan over lt list scales with the Bounce catalog and can DoS portfolio configuration	Medium	Fixed
[M-10]	A single held LT’s reverting price view freezes the entire treasury with no on-chain recovery	Medium	Fixed
[M-11]	Symbols array grows without bound	Medium	Fixed
[M-12]	Unredeemable sell leg: fee skipped or forfeited	Medium	Fixed
[L-01]	Rebalance slippage check uses a pre-checkpoint rate overstating redeemable value	Low	Fixed
[L-02]	‘prepareRedeem‘ path bypasses the minRedeemUsdc slippage check	Low	Acknowledged
[L-03]	minLtMintUsdc manually mirrors Bounce’s minTransactionSize reverting deploys until the rebalancer catches up	Low	Fixed
[L-04]	MAX_EXTRA_PREMIUM Allows 99% Buyer Value Transfer	Low	Fixed

ID	Title	Severity	Status
[L-05]	‘withdrawLtsTo‘ overpays the first seller during the streaming-fee window	Low	Fixed
[L-06]	Missing protocol-wide pause across treasuries	Low	Fixed
[L-07]	Per-Leg slippage floors make multi-leg buys and rebalances revert on a single unstable leg	Low	Acknowledged
[L-08]	‘sell()‘ payout can leave small sellers with LT dust	Low	Acknowledged
[L-09]	‘buy()‘ lets the caller waive mint slippage on a deploy of the entire shared idle pool	Low	Acknowledged
[L-10]	Rebalance is limited by bounce redeem min TX size	Low	Fixed
[L-11]	‘buy‘ can accept payment yet mint 0 shares — no ‘agentOut > 0‘ check and no minimum supply	Low	Fixed
[L-12]	Paused-leg skip in ‘_mintLTs‘ silently ignores the buyer’s per-leg ‘minLtOuts‘ floor	Low	Fixed
[L-13]	Fee pushed to a hardcoded, unchangeable recipient	Low	Fixed

Findings

High Severity

[H-01] Duplicate LT across symbols double-counts NAV and lets the rebalancer drain co-investors

The duplicate guard in `_setTargetPortfolio` only compares entries inside the array passed in the current call. It never checks a new symbol's `lt` against symbols already stored from previous calls, so the same LT can back two live symbols.

```
for (uint256 j = 0; j < i; j++) {
    if (keccak256(bytes(spec.symbol)) ==
        keccak256(bytes(newPortfolio[j].symbol))) {
        revert DuplicateSymbol(spec.symbol);
    }
    if (spec.lt == newPortfolio[j].lt) revert DuplicateLt(spec.lt);   // only
        checks within this array
}
```

If `ltX` is already registered under symbol `"A"`, a second call with a single entry `("B", ltX)` slips through, and `"B"` is appended pointing at the same `ltX`. The old `"A"` is zeroed to `bps = 0` but `_prune ExitedSymbols` keeps it, because it only removes a symbol whose LT balance is zero.

`nav()` then walks `symbols[]` and reads `balanceOf` once per symbol, so the one real balance is counted twice:

```
for (uint256 i = 0; i < n; i++) {
    IBounceLT lt = IBounceLT(assets[symbols[i]].lt);
    uint256 bal = lt.balanceOf(address(this));
    if (bal > 0) total += lt.ltToBaseAmount(bal);
}
```

`withdrawLtsTo` walks the same array on a sell, so the inflated NAV inflates the seller's payout and the per-symbol loop transfers LT out of the same balance more than once. Each extra call adds one more symbol on the same LT, so NAV can be inflated to an arbitrary N times the truth.

This breaks two whitepaper invariants: "duplicates are rejected", and "a compromised rebalancer cannot steal principal". In classic mode the on-chain rebalancer is the token creator, an untrusted party, while the curve is open to public buyers.

Example

1. Genesis registers `"A" -> ltX` at 100 percent, creator holds 1000e18 shares
2. A public buyer buys in, supply is 2000e18, treasury holds ltX worth 1000 USDC, both own 50 percent
3. Creator calls `setTargetPortfolio` with `("B" -> ltX, 100%)`, the within-array guard lets it pass
4. `nav()` now counts ltX twice and reports 2000 USDC for 1000 USDC of real backing
5. Creator sells their 50 percent, the payout loop sends ltX twice, so they take 750e18 instead of the fair 500e18
6. Only 250e18 is left for the buyer who still holds 50 percent

IMPACT

A token creator can inflate `nav()` and drain the principal of public buyers in their own curve, approaching 100 percent of the treasury with enough duplicate registrations.

Severity assumes the creator is not trusted toward co-investors who buy into the public curve, consistent with the stated invariant. If creators are intended to be fully trusted, the classic-mode impact downgrades, but the agentic-mode vector still violates the invariant.

RECOMMENDATION

Enforce "each LT appears in `symbols[]` at most once". The duplicate guard must check the persistent set, not just the array passed in the current call — keep a reverse index and clear it on prune:

```
+    /// every LT address currently bound to a registered symbol
+    mapping(address => bool) private ltRegistered;
@@ _setTargetPortfolio -- the new-symbol branch
         if (!asset.registered) {
             if (!_isFactoryLt(factoryLts, spec.lt)) revert LtNotAllowed(
                 spec.lt);
+            if (ltRegistered[spec.lt]) revert DuplicateLt(spec.lt);
+            ltRegistered[spec.lt] = true;
             symbols.push(spec.symbol);
             assets[spec.symbol] = AssetConfig({lt: spec.lt,
                 targetBps: spec.bps, registered: true});
         } else if (asset.lt != spec.lt) {
@@ _pruneExitedSymbols -- when a symbol is removed
+            ltRegistered[a.lt] = false;
             symbols.pop();
             delete assets[sym];
```

Medium Severity

[M-01] Async redeemed rebalance USDC Is re-deployed by weight on the next buy, undoing the rebalance

When a rebalance shrink goes through `prepareRedeem`, the redeemed USDC arrives as idle only after Bounce settles, and the next `buy` deploys that idle through `AgentTreasury::_mintLTs`, which splits it across legs by target weight — not toward the rebalance's target balances. The settled funds re-inflate the over-weight leg instead of topping up the under-weight one.

```
// AgentTreasury.sol -- _mintLTs (reached via buy -> _deployIdle),
// deploys ALL idle by weight
usdcToAllocate = (usdcAmount * uint256(assets[sym].targetBps)) / BPS_DENOM;
```

`executeRebalanceStep` cannot place the funds itself in this case: the grow leg is deferred while settlement is pending.

```
// AgentTreasury.sol -- executeRebalanceStep buy loop
if (growUsdc > usdcBal) {
```

```
    if (rebalanceInFlight) { emit MintDeferredForSettlement(symbols[i],
        growUsdc, usdcBal); continue; }
    ...
}
```

Once the flag clears, the idle is up for grabs: a `buy` deploys it by weight via `_deployIdle`, whereas only a fresh `executeRebalanceStep` would deploy it toward targets.

Scenario — two legs, target 90/10, holding 29,000 / 1,000 (target 27,000 / 3,000):

1. `executeRebalanceStep` redeems 2,000 from leg 1 via `prepareRedeem`; leg 2 grow is deferred, flag set.
2. Settlement delivers ~2,000 idle; `settleRebalance` clears the flag.
3. A `buy` deploys the idle 90/10 → leg 1 +1,800, leg 2 +200 → **28,800 / 1,200** (≈96/4), back to the original imbalance.

IMPACT

Any async-redemption rebalance can be silently undone by a single intervening buy, so the rebalancer cannot reliably converge to target weights, and each redeem→redeploy cycle burns the Bounce redemption fee for no net rebalancing. Whether a rebalance works at all depends on the redeem path and who calls next.

RECOMMENDATION

Keep async-redemption proceeds out of the weight-based buy deploy: reserve the in-flight redeemed amount so `_deployIdle` cannot consume it, and deploy it only through the balance-aware rebalance path. This makes settled rebalance funds land on the under-weight legs regardless of who transacts next.

[M-02] Deploy remainder reverts most of the buys

`AgentTreasury::_mintLTs` assigns the leftover USDC to the last `symbols[]` entry without checking that leg's `targetBps`, so when the last entry is a zeroed leg pointing at a live LT it receives only rounding dust, and `lt.mint` reverts on `minTransactionSize`.

```
// AgentTreasury.sol -- _mintLTs
if (i == n - 1) {
    usdcToAllocate = usdcAmount - deployed;   // remainder, regardless of this
        leg bps
} else {
    usdcToAllocate = (usdcAmount * uint256(assets[sym].targetBps)) / BPS_DENOM;
    deployed += usdcToAllocate;
}

if (usdcToAllocate == 0) {                    // dust is NOT zero,
    so this does not catch it
    if (minLtOuts[i] != 0) revert SlippageExceeded();
    continue;
}
...
uint256 ltOut = lt.mint(address(this), usdcToAllocate, minLtOuts[i]);  //
```

```
    reverts if dust < minTransactionSize
```

Active weights sum to `BPS_DENOM`, so the active legs (all at non-last indices) consume `deployed ≈ usdcAmount`. A zeroed leg (`targetBps == 0`) that persists in `symbols[]` and lands at index `n-1` then gets `usdcAmount - deployed` = a few wei of flooring dust. That dust is non-zero, so the `usdcToAllocate == 0` guard skips it, and `lt.mint` reverts `BelowMinTransactionSize`.

`minDeployUsdc()` does not prevent this: it is derived from `minBps` (smallest non-zero weight) and never accounts for the remainder landing on a zero-weight leg.

IMPACT

Every deploy-triggering buy reverts while a `targetBps == 0` symbol with a live, non-paused LT occupies the last slot of `symbols[]` — a state reachable through normal basket rotation, since zeroed legs are not removed from `symbols[]`. Buys stay bricked until the array order changes.

RECOMMENDATION

Treat the remainder leg like any other: skip it when its `targetBps == 0`, and route the rounding remainder to an idle.

[M-03] Rebalance buy sizes targets against a stale navAtStart after redeems change the rates

`AgentTreasury::executeRebalanceStep` snapshots `navAtStart` once at the top, then uses it to compute `targetUsdcValue` for every leg — but the sell loop that runs in between calls `redeem`/`prepareRedeem`, which pay redemption fees out of the LT and trigger Bounce's `_checkpoint` (streaming fee). Both permanently lower the true nav and the per-LT `exchangeRate`, so the buy loop targets a nav that no longer exists.

```
// AgentTreasury.sol -- executeRebalanceStep
uint256 navAtStart = nav();                                    // snapshot:
    pre-fee, un-checkpointed rates

// sell loop: each redeem/prepareRedeem pays a redemption fee out of the LT
// and runs _checkpoint(), which streams a fee out of totalAssets ->
// exchangeRate drops
uint256 actualBaseOut = lt.redeem(address(this), shrinkLt, minRedeemUsdc[i]);
...
// buy loop:
uint256 currentUsdcValue = lt.ltToBaseAmount(lt.balanceOf(address(
    this)));  // fresh, LOWER rates
uint256 targetUsdcValue = (navAtStart * uint256(assets[symbols[i]].targetBps))
    / BPS_DENOM;  // stale, HIGHER nav
uint256 growUsdc = targetUsdcValue - currentUsdcValue;
```

In Bounce, `redeem` removes gross value proportional to burned supply, but the redemption fee leaves the contract and `_checkpoint` streams a fee out of `totalAssets` without burning supply, so `exchangeRate = totalAssets / totalSupply` falls. The view `nav()` used for `navAtStart` counts assets that the sell loop then bleeds away, so by the buy loop the portfolio is genuinely worth less than `navAtStart` while each leg's `targetUsdcValue` is still a fraction of the inflated `navAtStart`.

IMPACT

The buy loop over-sizes grow targets: it allocates fractions of a nav that has already shrunk by the fees realized during selling, so legs it processes are minted past their true target weight and the portfolio drifts from the intended `targetBps` proportions.

RECOMMENDATION

Recompute the nav reference after the sell loop completes, so both `targetUsdcValue` and `currentUsdcValue` are measured against the same post-redeem rates.

[M-04] Buy premium is priced at supplyBefore, so one large buy pays less than many small ones

`AgentCurve::buy` charges the premium using `premium(supplyBefore)` — a single value read from the supply *before* the buy — and applies it to the whole `usdcIn`, no matter how far that one buy pushes the supply.

```
// AgentCurve.sol -- buy / _quoteBuy
uint256 supplyBefore = totalSupply();
uint256 navBefore = TREASURY.nav();
agentOut = _quoteBuy(usdcIn, supplyBefore, navBefore);
...
uint256 p = premium(supplyBefore);                // one premium for the
    entire buy
return (usdcIn * supplyBefore * ONE) / (navBefore * p);
```

`premium()` is meant to rise as supply approaches `PREMIUM_CAP_SUPPLY`, so each AGENT minted closer to the cap should cost more. But because the premium is fixed at the *starting* supply, a buyer who jumps the whole distance in one transaction pays the low starting premium on every AGENT, while a buyer who moves the same distance in small steps pays the premium that rises under their feet.

The protocol's intended rule — AGENT minted nearer the cap pays a higher premium — holds only for small buys; a single large buy bypasses it.

Both buyers want the same result: push supply from 950 to the cap at 1,000, minting 50 AGENT, at a fair price of 1 USDC per AGENT before premium. `premium(supply)` climbs as supply nears the cap, so the cost of each AGENT depends on the supply at the moment it is minted:

AGENT minted at supply...	`premium(supply)` there	One large buy charges	50 small buys charge
950 (start)	91.25×	91.25×	91.25×
975 (halfway)	96.1×	91.25×	96.1×
999 (last one)	100.8×	91.25×	100.8×
Total for all 50 AGENT		**~4,562 USDC**	**~4,799 USDC**

The large buy freezes the premium at the start value (91.25×) for every AGENT, even the ones it mints right at the cap; A clear example of the problem: imagine a buyer who purchases the entire supplyCap at once—they would pay almost no premium.

IMPACT

The premium curve is the protocol's tool for charging late buyers more and routing that surplus to existing holders. A buyer who concentrates demand into one large transaction underpays that premium versus buyers who arrive incrementally, so existing holders collect less than the curve dictates.

RECOMMENDATION

The problem here lies in the formula itself—a more appropriate pricing formula would be a virtual constant product amm. This formula specifically prevents the trade size from affecting the amount of commissions paid.

At the very least, this behavior should simply be documented.

[M-05] One stuck Bounce redemption freezes all buys and sells for the whole fund

When a LT cannot be redeemed atomically, `executeRebalanceStep` calls `lt.prepareRedeem` (which escrows the LT and records `userCredit` on that LT) and sets a single global `rebalanceInFlight = true`.

`deployUsdc` and `withdrawLtsTo` both revert `RebalancePending` while the flag is set, and it clears only via `_clearInFlightIfSettled`, which stays set while any held LT still has `userCredit(treasury) > 0`.

On the Bounce side redemptions are LT independent, and `_executeRedemption` has two conditions that each return early without zeroing `userCredit`. The second one is where the problem is: if a credit cannot pay redemption fees which has flat parameter - we cannot execute our redemption. Moreover, LeverageToken executor may, intentionally or unintentionally, simply fail to execute our withdrawal.

https://github.com/bounce-tech/bounce-smart-contracts/blob/main/src/LeveragedToken.sol#L238-L240

```
if (baseAssetBalance() < baseAmount_) return false;
...
if (baseAmount_ < redemptionFee_) return false;
```

The flag is global and keyed on `userCredit`, so one stuck leg blocks buys and sells for the whole basket, even legs Bounce already settled. A price drop never zeroes `userCredit`, so the treasury can hold almost none of a leg yet stay frozen by its dust credit. There is no `cancelRedeem` path and no emergency exit, and `settleRebalance()` just re-checks `userCredit > 0`.

RECOMMENDATION

Add a function that calls `cancelRedeem()` on a pending leg after Bounce's 1h delay. It pulls the possibly depreciated LT back into the treasury's balance, zeroes `userCredit`, lets `_clearInFlightIfSettled` and clear the flag:

```
+    function cancelRedeem(string calldata symbol) external onlyRebalancer {
+        IBounceLT(assets[symbol].lt).cancelRedeem();
+        _clearInFlightIfSettled();
+    }
```

[M-06] Deploy at low minBps leaves buyer capital undeployed as sells drain Idle below the threshold

`AgentTreasury::_deployIdle` defers the **entire** deploy whenever idle USDC is below `minDeployUsdc()`, and that threshold scales inversely with `minBps` (the smallest non-zero target weight). Because `withdrawLtsTo` pays sells out of idle first, idle is continuously drained and rarely climbs to a threshold set high by a small weight — so buyer USDC is minted into LTs only sporadically, if ever.

```
// AgentTreasury.sol -- _deployIdle
uint256 idle = USDC.balanceOf(address(this));
uint256 threshold = minDeployUsdc();
if (idle < threshold) {
    emit DeployDeferred(idle, threshold);
    return;            // ALL idle stays undeployed, not just the sub-min
        remainder
}
_mintLTs(idle, minLtOuts);
```

```
// AgentTreasury.sol -- minDeployUsdc: threshold grows as minBps shrinks
return (minLtMintUsdc * BPS_DENOM + uint256(minBps) - 1) / uint256(minBps);
```

Every sell pulls USDC straight out of idle before touching LTs, so idle is pushed back down between buys:

```
// AgentTreasury.sol -- withdrawLtsTo
//

1. Pay from USDC idle first
uint256 usdcOut = notional <= idle ? notional : idle;
if (usdcOut > 0) USDC.safeTransfer(recipient, usdcOut);
```

With `minLtMintUsdc = 10e6`, the smallest weight the rebalancer assigns sets how much idle must accumulate — in one buy or across buys — before any deploy happens:

Smallest target weight (`minBps`)	`minDeployUsdc`
1000 (10%)	100 USDC
100 (1%)	1,000 USDC
10 (0.1%)	10,000 USDC

At a 0.1% leg, idle must reach 10,000 USDC and survive intervening sells before a single deploy fires. Under two-sided flow the threshold is rarely crossed, so capital stays idle.

IMPACT

The protocol's core function — deploying buyer capital into the target leveraged-token portfolio — is effectively disabled when the rebalancer assigns any small (but legitimate) weight. Buyers receive AGENT backed by idle USDC at face value rather than the intended leveraged exposure, and `nav` drifts from the target portfolio. No funds are lost, but the product does not perform its stated purpose, and a rebalancer can trigger this silently by shrinking one leg's weight.

RECOMMENDATION

Stop gating the whole deploy on a single global threshold. Deploy per leg: for each symbol whose computed allocation meets the per-leg minimum, mint it; leave only the legs whose allocation falls below the minimum sitting in idle. Rebalancer can rebalance this balances after.

Mininal deploy size should not be so strictly dependent on minBPS.

At the very least, this behavior needs to be documented so that the rebalancer understands how the distribution affects the protocol's future behavior.

[M-07] Normal rebalances may revert

A rebalance sells one token to buy another. It plans the buy from the portfolio value **before** the redemption fee, but receives cash after the fee — so it is short by exactly the fee and reverts.

Scenario. Treasury = 2000 USDC, fully invested: 1000 in A, 1000 in B

1. Goal: re-weight 50/50 → 20/80 ⇒ **sell 600 of A**, **buy 600 of B**.
2. One transaction: sell loop, then buy loop.
3. **Sell 600 of A** → Bounce's 1.5% fee = 9 USDC → treasury receives **591**, not 600.
4. **Buy step** still wants **600** (sized up front from full NAV), but only **591** is on hand.
5. **591 < 600 → the whole tx reverts** `InsufficientUsdcForMint(600e6, 591e6)`. Nothing moves.

Root cause is the buy amount is sized from value before the fee, but the cash that arrives is after the fee; the gap is exactly the fee. With no idle cash, every "sell A to fund B" rebalance reverts.

IMPACT

Rebalancing is the protocol's core operation, and the natural parameters an off-chain agent emits revert whenever idle USDC ≈ 0 (the normal state). The caller can't reliably pre-set a cap, because the shortfall depends on `redemptionFee × targetLeverage` and the live `exchangeRate`.

RECOMMENDATION

On the atomic path, mint what the proceeds actually cover (mirror the deferred branch) instead of reverting:

```
  if (growUsdc > usdcBal) {
      if (rebalanceInFlight) { emit MintDeferredForSettlement(...); continue; }
-     revert InsufficientUsdcForMint(symbols[i], growUsdc, usdcBal);
+     if (usdcBal == 0) continue;    // nothing realized for this leg this step
+     growUsdc = usdcBal;            // buy the realized net; don't over-target
    the fee
  }
```

[M-08] `executeRebalanceStep` sell path always triggers `lt.redeem` does to a wrong comparison

In the rebalance sell loop, the treasury decides whether to redeem atomically or queue it with `prepareRedeem`:

```
uint256 expectedBaseOut = lt.ltToBaseAmount(shrinkLt);
if (expectedBaseOut < minRedeemUsdc[i]) revert SlippageExceeded();

// it LT has enough USDC in buffer, use atomic redeem(), else async path
int256 bufferRaw = LT_HELPER.getLeveragedTokenBufferAssetValue(address(lt));
uint256 buffer = bufferRaw < 0 ? 0 : uint256(bufferRaw);
if (expectedBaseOut <= buffer) {
    uint256 actualBaseOut = lt.redeem(address(this), shrinkLt,
        minRedeemUsdc[i]);
    emit AtomicRedeem(symbols[i], address(lt), shrinkLt, actualBaseOut);
} else {
```

However the two operands are on different scales. `expectedBaseOut = lt.ltToBaseAmount(shrinkLt)` is 6 decimal USDC, because the real Bounce `ltToBaseAmount` ends in `scaleTo(6)`.

However `buffer` comes from `getLeveragedTokenBufferAssetValue`, which is 18 decimal because it uses `baseAssetBalance().scaleFrom(6)`.

The guard compares them directly with no rescaling, so `buffer` is about 1e12 times too large and `expectedBaseOut <= buffer` is essentially always true. The atomic branch is always taken and the async `prepareRedeem` fallback in the `else` is dead code.

The treasury then calls `lt.redeem()`, which reverts when the requested amount exceeds the LT's idle EVM-side USDC:

```
uint256 baseAmount_ = ltToBaseAmount(ltAmount_);
if (baseAmount_ > baseAssetBalance()) revert InsufficientBalance();
```

RECOMMENDATION

Scale `expectedBaseOut` up from 6 dec USDC to 1e18 before comparing (or `buffer` to 6):

```
+    /// USDC (6-dec) to 1e18 scale, matching the buffer helper's scaleFrom(6).
+    uint256 private constant USDC_TO_WAD = 1e12;
```

```
            int256 bufferRaw = LT_HELPER.getLeveragedTokenBufferAssetValue(
                address(lt));
            uint256 buffer = bufferRaw < 0 ? 0 : uint256(bufferRaw);

-           if (expectedBaseOut <= buffer) {
+           if (expectedBaseOut * USDC_TO_WAD <= buffer) {
```

[M-09] `_isFactoryLt` scan over lt list scales with the Bounce catalog and can DoS portfolio configuration

`AgentTreasury::_setTargetPortfolio` validates each newly-registered spec by copying the *entire* `BOUNCE_FACTORY.lts()` array into memory and linear-scanning it via `_isFactoryLt`, even though Bounce's `Factory` exposes an O(1) `ltExists(address)` mapping returning the same boolean

```
// contracts/src/AgentTreasury.sol -- _setTargetPortfolio (excerpt)
address[] memory factoryLts = BOUNCE_FACTORY.lts();   // O(N_bounce) SLOADs
    every call

for (uint256 i = 0; i < newPortfolio.length; i++) {
    ...
    if (!asset.registered) {
        if (!_isFactoryLt(factoryLts, spec.lt)) revert LtNotAllowed(spec.lt);
        ...
    }
}
```

As Bounce's catalog grows, both `AgentTreasury::initialize` and `AgentTreasury::setTarget Portfolio` hit a hard ceiling that perpgame does not control. It leads to DoS in the worst scenario.

RECOMMENDATION

Add `ltExists` to the local interface and use it instead of linear scan:

```
 interface IBounceFactory {
     function lts() external view returns (address[] memory);
+    function ltExists(address lt) external view returns (bool);
 }
```

```
-    address[] memory factoryLts = BOUNCE_FACTORY.lts();
     for (uint256 i = 0; i < newPortfolio.length; i++) {
         AssetSpec memory spec = newPortfolio[i];
         ...
         if (!asset.registered) {
-            if (!_isFactoryLt(factoryLts, spec.lt)) revert LtNotAllowed(
    spec.lt);
+            if (!BOUNCE_FACTORY.ltExists(spec.lt)) revert LtNotAllowed(
    spec.lt);
             ...
         }
     }
```

`_isFactoryLt` becomes dead code and can be removed.

[M-10] A single held LT's reverting price view freezes the entire treasury with no on-chain recovery

`nav()` is the only pricing primitive and it walks `symbols[]` calling each leg.

```
function nav() public view returns (uint256 total) {
    uint256 n = symbols.length;
    for (uint256 i = 0; i < n; i++) {
        IBounceLT lt = IBounceLT(assets[symbols[i]].lt);
        uint256 bal = lt.balanceOf(address(this));
        if (bal > 0) total += lt.ltToBaseAmount(bal);   // no try/catch
    }
    total += USDC.balanceOf(address(this));
```

```
}
```

`nav()` called on every path, so if one leg's `ltToBaseAmount` reverts, all of them revert. On HyperEVM mainnet a Bounce LT prices itself off HyperCore. `ltToBaseAmount` reaches a HyperCore read precompile:

```
AgentTreasury::nav() -> 0x18b8...baBE::ltToBaseAmount(1e18)
  -> ltImplementation() = 0xc92C434a...
  -> 0xc92C434a...::ltToBaseAmount [delegatecall]
     -> staticcall 0x0000000000000000000000000000000000000801   // HyperCore
         spotBalance(account, token)
```

The HyperCore docs state: Precompiles called on invalid inputs such as invalid assets or vault address will return an error and consume all gas passed into the precompile call frame. So if the LT's underlying HyperCore token is delisted / its index becomes invalid `ltToBaseAmount` reverts permanently.

There is no on-chain recovery. Because `_setTargetPortfolio` emits `RebalanceTargetSet(... , nav(), ...)` and `executeRebalanceStep` starts with `navAtStart = nav()`, the rebalancer can neither rotate the bad LT out nor rebalance — both revert at `nav()`. The treasury has no `rescue`/`skim` and the LT cannot be moved out by any other path, so the only fix is a 48h-timelocked beacon-implementation upgrade.

Example

1. A treasury holds `HYPE5L`; buys, sells and rebalances all work normally.
2. Hyperliquid delists the LT's underlying spot/perp asset, so `spotBalance` at `0x...0801` returns an error.
3. `HYPE5L.ltToBaseAmount(bal)` now reverts, so `nav()` reverts.
4. Every holder is locked out: `buy` reverts at `navBefore`, `sell` reverts inside `withdrawLtsTo`.
5. The rebalancer tries to remove the bad LT reverts at the `RebalanceTargetSet(..., nav(), ...)` emit; `executeRebalanceStep` reverts at `navAtStart = nav()`.
6. Funds are frozen until a 48h-timelocked beacon upgrade ships a patched implementation.

IMPACT

A single delisted/unpriceable held LT permanently freezes the entire treasury: no holder can enter or exit, and the rebalancer cannot remove the offending leg or rebalance. Recovery requires a 48h-timelocked beacon upgrade by the multisig.

RECOMMENDATION

This is a rare, high-impact, recoverable event, so a minimal fix is reasonable — the priority is to not make it worse.

Minimal fix (recommended): drop the `nav()` call from the `RebalanceTargetSet` emit — `emit RebalanceTargetSet(newPortfolio, 0, reasoningCid)` — so portfolio rotation is never bricked. Fork-verified caveat: this alone does **not** unfreeze `buy`/`sell` or remove the stuck leg (`nav()` is still on those paths), so recovery from an actually-unpriceable held LT remains the 48h-timelocked beacon upgrade. For a rare event with funds recoverable (not lost), accepting that governance-recovery path is reasonable.

Optional, only if no-upgrade recovery is wanted: - a guardian `forceRemoveLt(symbol)` that evicts an unpriceable/delisted leg without pricing it (restores `nav`/`buy`/`sell`); or - a trustless raw in-kind `emergencyExit` paying `ltBalance * agentShares / totalSupply` per leg with **no `ltToBaseAmount`** call, so holders can exit without an upgrade.

[M-11] Symbols array grows without bound

`AgentTreasury` never reliably shrinks `symbols[]`: `_setTargetPortfolio` only zeroes a dropped leg's `targetBps`, and the sole removal path `_pruneExitedSymbols` deletes a symbol only when `lt.balanceOf(treasury) == 0 && userCredit == 0` — a condition two independent mechanisms keep permanently false.

```
// AgentTreasury.sol -- _setTargetPortfolio drops a leg by zeroing, not removing
for (uint256 i = 0; i < existingN; i++) {
    assets[symbols[i]].targetBps = 0;   // zeroed, still in symbols[]
}

// AgentTreasury.sol -- _pruneExitedSymbols, the only removal path
if (a.targetBps == 0 && lt.balanceOf(address(this)) == 0 && lt.userCredit(
    address(this)) == 0) {
    // ... swap-pop and delete
}
```

Bounce LTs are ordinary ERC-20s, so anyone can send 1 wei of a retired LT to keep `balanceOf != 0` forever. A 1-wei leg also has `ltToBaseAmount == 0`, so the redeem/withdraw loops skip it and the balance never returns to 0. The contract has no `sweep`, so the prune condition can never be satisfied again.

Also, a registered symbol's `lt` is immutable — `_setTargetPortfolio` reverts any change with `SymbolTokenMismatch`:

```
// AgentTreasury.sol -- _setTargetPortfolio
} else if (asset.lt != spec.lt) {
    revert SymbolTokenMismatch(spec.symbol, asset.lt, spec.lt);  // blocks the
        refresh
}
```

Bounce's `Factory::redeployLt` legitimately swaps an LT's contract address while preserving its `(asset, leverage, direction)` and therefore its deterministic symbol. After such a redeploy the rebalancer cannot re-point `assets["HYPE5L"].lt` to the new address and must register a fresh symbol (`HYPE5L_v2`) for the same product. The original entry orphans: its `lt` is the delisted contract, which the treasury can no longer drain through Bounce, so `balanceOf`/`userCredit` can never reach zero and it is unprunable.

Every path iterates the full stored `symbols[]` with no distinction between active and dead legs each dead entry costing an SLOAD plus an external call per iteration. Worse, `buy` and `executeRebalanceStep` force the caller to size per-symbol slippage arrays to the full length: `deployUsdc` reverts `LengthMismatch` unless `minLtOuts.length == symbols.length`, so every buyer must pad zeros for every dead leg, and the rebalancer must pass four such arrays.

IMPACT

The `symbols[]` array expands monotonically under two forces the protocol cannot counter. Every user permanently overpays gas proportional to the dead legs, and every buy/rebalance must carry full-length slippage arrays covering legs that no longer exist. In the worst case this scenario leads to hard DoS of most protocol functions.

RECOMMENDATION

1. Add a `sweepDust`, so retired LT become deletable.
2. Add a rebalancer-callable migration that updates `assets[symbol].lt` in place, so a Bounce redeploy refreshes the existing symbol instead of orphaning it.

[M-12] Unredeemable sell leg: fee skipped or forfeited

On a sell, when a leg can't be instant-redeemed, `withdrawLtsTo` branches on `returnLts`, and the 1% fee is taken only on the USDC that actually redeemed:

```
} catch { if (returnLts) IERC20(address(lt)).safeTransfer(recipient,
    ltOut); }  // else: leg left in treasury
...
uint256 fee = (usdcOut * SELL_FEE_BPS) / BPS_DENOM;   // fee only on redeemed
    USDC
```

IMPACT

With `returnLts=true` the LT goes to the seller untaxed, so the 1% on that leg isn't collected (if nothing redeems, the fee is skipped entirely). With `returnLts=false` the leg is skipped and stays in the treasury — the seller burned all their shares but is paid only for what redeemed, so the rest effectively passes to the other holders, and since `quoteSellUsdc` excludes that leg the shortfall is easy to miss (seller ~$495, ~$500 left behind). Real instant buffers are small (ETH5L ≈ $198), so this is the common path on a larger sell rather than a rare edge.

RECOMMENDATION

In the `catch`, always hand the seller the LT for the unredeemable part, minus a 1% fee in LT, regardless of `returnLts` — this covers both:

```
} catch {
    uint256 ltFee = (ltOut * SELL_FEE_BPS) / BPS_DENOM;
    if (ltFee > 0) IERC20(address(lt)).safeTransfer(FEE_RECIPIENT, ltFee);
    IERC20(address(lt)).safeTransfer(recipient, ltOut - ltFee);
}
```

Low Severity

[L-01] Rebalance slippage check uses a pre-checkpoint rate overstating redeemable value

`AgentTreasury::executeRebalanceStep` checks `minRedeemUsdc[i]` against `expectedBaseOut = lt.ltToBaseAmount(shrinkLt)`, computed at the current `exchangeRate()` — but `redeem`/`prepareRedeem` each run `_checkpoint()` first, which streams the management fee out of `totalAssets` and lowers `exchangeRate`, so the value the redeem actually computes is smaller than `expectedBaseOut`.

```
// AgentTreasury.sol -- executeRebalanceStep
uint256 expectedBaseOut = lt.ltToBaseAmount(shrinkLt);          //
    pre-checkpoint rate
if (expectedBaseOut < minRedeemUsdc[i]) revert SlippageExceeded();
```

```
// LeveragedToken.sol -- redeem
_checkpoint();                                                  // streams fee,
    exchangeRate drops
uint256 baseAmount_ = ltToBaseAmount(ltAmount_);               // recomputed at
    the LOWER rate
```

RECOMMENDATION

Do slippage check for actual redeemed base amount.

[L-02] `prepareRedeem` path bypasses the minRedeemUsdc slippage check

`AgentTreasury::executeRebalanceStep` validates `minRedeemUsdc[i]` against `expectedBaseOut` once, then on the async branch calls `prepareRedeem`, which carries no minimum and settles later at a different rate — so the check guards nothing for that path.

```
// AgentTreasury.sol -- executeRebalanceStep sell loop
uint256 expectedBaseOut = lt.ltToBaseAmount(shrinkLt);
if (expectedBaseOut < minRedeemUsdc[i]) revert SlippageExceeded(
    );   // checked at current rate
...
if (expectedBaseOut <= buffer) {
    lt.redeem(address(this), shrinkLt, minRedeemUsdc[i]);           // atomic:
        min enforced on realized output
} else {
    lt.prepareRedeem(shrinkLt);                                     // async:
        no min, settles later at unknown rate
}
```

`prepareRedeem` only records `userCredit`; the USDC payout happens through Bounce's external keeper settlement at whatever `exchangeRate()` prevails then. Nothing ties `minRedeemUsdc[i]` to that settlement.

IMPACT

If `exchangeRate()` falls between `prepareRedeem` and settlement, the treasury receives less than `minRedeemUsdc[i]` and nothing reverts or flags it. Conversely, if the exchange rate rises between

settlement and preparation, we won't miss a valid withdrawal.

RECOMMENDATION

A true atomic revert isn't possible since settlement is external, and best what can be done without massive changes - simply document this behaviour. Alternatively, you can skip the `prepare_redeem` step entirely and redeem only the LTs available for instant redemption.

[L-03] minLtMintUsdc manually mirrors Bounce's minTransactionSize reverting deploys until the rebalancer catches up

`AgentTreasury::minDeployUsdc` sizes the deploy threshold from `minLtMintUsdc`, a value the rebalancer must manually keep in sync with Bounce's `minTransactionSize`. Bounce can change `minTransactionSize` at any time, and until the rebalancer mirrors it, leg allocations sized against the stale `minLtMintUsdc` fall below Bounce's minimum and `lt.mint` reverts.

```
// AgentTreasury.sol -- minLtMintUsdc is a hand-set mirror, not a live read
function setMinLtMintUsdc(uint256 next) external onlyRebalancer {
    if (next > MAX_MIN_LT_MINT_USDC) revert MinLtMintUsdcTooHigh(next,
        MAX_MIN_LT_MINT_USDC);
    minLtMintUsdc = next;
    ...
}
```

```
// LeveragedToken.sol -- the live bound the deploy must clear
if (baseAmount_ < $.globalStorage.minTransactionSize()) revert
    BelowMinTransactionSize();
```

The deploy threshold guarantees only that each leg receives at least `minLtMintUsdc`; once Bounce's `minTransactionSize` exceeds that, the smallest leg's mint reverts.

IMPACT

Buys and rebalance steps revert whenever Bounce raises `minTransactionSize` above the current `minLtMintUsdc`, and stay broken until the rebalancer notices and calls `setMinLtMintUsdc` — forcing continuous off-chain monitoring of a third party's parameter.

RECOMMENDATION

Read `minTransactionSize` from Bounce directly when sizing the deploy threshold instead of mirroring it in a manually-set variable, so the contract always tracks the live bound and no rebalancer action is required.

[L-04] MAX_EXTRA_PREMIUM Allows 99% Buyer Value Transfer

`AgentCurve` bounds the per-launch premium add-on at `MAX_EXTRA_PREMIUM = 100e18`, which permits a premium multiplier of 101×.

```
// AgentCurve.sol
uint256 private constant MAX_EXTRA_PREMIUM = 100e18;
...
if (extraPremium_ > MAX_EXTRA_PREMIUM) revert
    ExtraPremiumTooHigh(extraPremium_, MAX_EXTRA_PREMIUM);
...
return (usdcIn * supplyBefore * ONE) / (navBefore * p); // p up to 101e18
```

At `EXTRA_PREMIUM = 100e18` and `supply >= PREMIUM_CAP_SUPPLY`, a buyer mints `1/101` of the NAV-pro-rata amount: `100/101 ~ 99%` of every USDC deposit accrues to existing holders, and since sells settle at the NAV floor, the buyer can recover at most ~1% of it by exiting.

IMPACT

A launch configured at the cap is economically confiscatory for buyers past `PREMIUM_CAP_SUPPLY` while passing all constructor validation.

RECOMMENDATION

Lower `MAX_EXTRA_PREMIUM` to a bound consistent with a buy/sell spread (e.g. `1e18`, capping the premium at 2×).

[L-05] `withdrawLtsTo` overpays the first seller during the streaming-fee window

Bounce LTs charge a streaming fee that accrues every second but is only deducted from `exchangeRate` at the next `mint`/`redeem`/`prepareRedeem` of that LT. Between those checkpoints `nav()` is stale-high by the size of the un-deducted fee.

`withdrawLtsTo` sizes the seller's payout off that stale `nav()` and pays out of idle USDC first:

```
uint256 navTotal = idle + totalLtValue;             // totalLtValue uses the
    STALE rate
uint256 notional = (navTotal * agentShares) / totalShares;
...
//

1. Pay from USDC idle first
uint256 usdcOut = notional <= idle ? notional : idle;
if (usdcOut > 0) USDC.safeTransfer(recipient, usdcOut);
```

If `notional` fits in idle, the seller walks away with USDC at the overstated mark and zero LT. The treasury still holds the entire LT position. When the checkpoint eventually fires, the streaming fee on the treasury's LT slice falls entirely on whoever is still holding curve shares; external LT holders pay their own separate fee.

The in-kind branch does not leak — the seller carries an LT slice out and pays their share of the fee at their own later checkpoint. Resident idle is reachable without any privileged action: paused legs (`_mintLTs continue`s them), buys below `minDeployUsdc`, async-rebalance settlement windows, and `bps = 0` retired legs all park USDC idle.

Example

One-leg treasury, 5x long, 2%/yr streaming fee. Effective LT decay = `2% * 5 = 10%/yr`.

1. Alice seeds 100k, fully deployed. Owner pauses the leg
2. Bob buys 60k. The mint is skipped, 60k stays idle. `nav() = 60k + 100k = 160k`. Bob owns 1/6 of supply
3. 30 days pass, no checkpoint on this LT. The treasury's 100k LT slice has silently accrued ~822 USDC of streaming fee (`100k * 10%/yr * 30/365`)
4. Bob sells. `notional = 160k * 1/6 = 26,666.67`, fits in idle. He gets 26,666.67 USDC and zero LT
5. A later mint/redeem fires `_checkpoint`. The 822 USDC fee on the treasury's slice now falls entirely on Alice — Bob is gone with cash. Bob's fair share would have been `1/6 * 822 = 137`; Alice eats it instead

```
Bob cash out (stale mark):    26,666.666666
Bob fair (true-rate):         26,529.680365
Bob excess:                      136.986301
Alice deficit:                   136.986300
```

`Bob excess == Alice deficit` to 1 wei on real Bounce bytecode. External LT holders pay their own separate fee.

RECOMMENDATION

Either pay idle USDC pro-rata or document this limitation so sellers know the cash branch can extract stale-fee value from remaining holders.

```
- //

1. Pay from USDC idle first
- uint256 usdcOut = notional <= idle ? notional : idle;
+ uint256 usdcOut = (idle * agentShares) / totalShares;
  if (usdcOut > 0) USDC.safeTransfer(recipient, usdcOut);
  uint256 remaining = notional - usdcOut;
```

[L-06] Missing protocol-wide pause across treasuries

`AgentTreasury` has no pause mechanism.

`TreasuryFactory` deploys an unbounded set of treasuries behind a shared beacon, but there is no operator-level control to halt them during an incident (e.g. LT mispricing on Bounce, which directly corrupts `nav()` and therefore buy/sell pricing in every treasury). The only protocol-wide lever is a beacon implementation upgrade, and the only per-treasury lever is each `rebalancer` deliberately keeping a redemption in flight — neither is a usable emergency stop.

RECOMMENDATION

Add a shared pause registry (e.g. a `paused()` flag on `TreasuryFactory` or a dedicated guardian contract, stored as a reference in the treasury implementation) and check it in `deployUsdc` and `withdrawLtsTo`.

[L-07] Per-Leg slippage floors make multi-leg buys and rebalances revert on a single unstable leg

`AgentTreasury::_mintLTs` enforces a separate slippage floor for every LT leg via `minLtOuts[i]`, so a single leg breaching its floor reverts the entire buy — even though the sell path already protects users with one aggregate floor.

```
// AgentTreasury.sol -- _mintLTs (buy/deploy path)
USDC.forceApprove(address(lt), usdcToAllocate);
uint256 ltOut = lt.mint(address(this), usdcToAllocate, minLtOuts[i]);
if (ltOut < minLtOuts[i]) revert SlippageExceeded(); // per-leg AND-gate
```

The same per-leg pattern gates `executeRebalanceStep` through `minRedeemUsdc[i]` and `minMintLt[i]`. Because these are leveraged tokens whose exchange rates move quickly, the buyer must set N floors that all hold simultaneously at execution time; the chance of the whole transaction reverting grows with each added leg.

By contrast, `withdrawLtsTo` guards the sell with a single aggregate bound and lets the legs net out against each other:

```
// AgentTreasury.sol -- withdrawLtsTo (sell path)
uint256 notional = (navTotal * agentShares) / totalShares;
if (notional < minUsdcOut) revert SlippageExceeded(); // one global floor
```

IMPACT

Multi-leg buys and rebalance steps revert whenever any one leg slips below its individual floor, with revert probability compounding as the portfolio adds legs. No funds are lost, but execution reliability degrades for exactly the fast-moving assets the protocol is built around, and a buyer cannot let an outperforming leg offset an underperforming one.

RECOMMENDATION

Replace the per-leg floors on the buy/deploy and rebalance paths with a single aggregate slippage bound on the operation's net LT value acquired, mirroring `minUsdcOut` in `withdrawLtsTo`. One global floor lets legs net against each other and reverts only when the aggregate execution is genuinely worse than the user accepts.

[L-08] `sell()` payout can leave small sellers with LT dust

`withdrawLtsTo` pays `min(notional, idle)` in USDC first, then covers the deficit with raw LT tokens, a proportional slice of every leg:

```
//

1. Pay from USDC idle first
uint256 usdcOut = notional <= idle ? notional : idle;
if (usdcOut > 0) USDC.safeTransfer(recipient, usdcOut);

//

2. Cover any deficit with LTs, proportional to each leg's USD value.
```

```
uint256 remaining = notional - usdcOut;
if (remaining > 0) {
    for (uint256 i = 0; i < n; i++) {
        ...
        uint256 ltOut = (bal * remaining) / totalLtValue;
        if (ltOut == 0) continue;
        lt.safeTransfer(recipient, ltOut);
    }
}
```

NAV values LTs at gross `ltToBaseAmount(bal)` with no fee deduction, so `notional` treats USDC and LT face value as interchangeable when their realizable values are not. An LT recipient must later call `lt.redeem` or `lt.prepareRedeem` on Bounce, paying `redemptionFee × targetLeverage` plus the executor fee on the async path. This creates two problems:

1. Two sellers burning the same share count get the same face value but different realizable USDC. Whoever drains the idle USDC walks away clean. The next seller eats the redemption fee on their LT slice.
2. Idle USDC is normally near zero after deploys, so the LT path is the common case. For a small sell with portfolio with a lot of LTs, each slice can be worth only a few dollars, below Bounce's `minTransactionSize`. Bounce's `redeem` and `prepareRedeem` both revert `BelowMinTransactionSize` under that floor, so the seller cannot convert the dust LTs to USDC at all.

RECOMMENDATION

Pay sellers in USDC rather than raw LTs. Either redeem the required LTs to USDC inside `withdrawLtsTo` and distribute net USDC, or schedule the withdrawal like Bounce so the seller receives USDC once it settles.

[L-09] `buy()` lets the caller waive mint slippage on a deploy of the entire shared idle pool

When calling `buy()` users supply `minLtOuts` to `deployUsdc`, which hands it to `_deployIdle` and then `_mintLTs`, where it becomes a slippage check on Bounce LTs:

```
USDC.forceApprove(address(lt), usdcToAllocate);
uint256 ltOut = lt.mint(address(this), usdcToAllocate, minLtOuts[i]);
if (ltOut < minLtOuts[i]) revert SlippageExceeded();
```

The contract applies no floor or sanity check on these values, so any caller can pass zeros and waive the treasury's mint slippage protection entirely.

The important part is that `_deployIdle` deploys the treasury's full idle balance, not just the caller's `usdcIn`:

```
uint256 idle = USDC.balanceOf(address(this));
uint256 threshold = minDeployUsdc();
if (idle < threshold) {
    emit DeployDeferred(idle, threshold);
    return;
}
_mintLTs(idle, minLtOuts);
```

RECOMMENDATION

Consider having a protocol fixed slippage percentage, or just manage it in `executeRebalanceStep` instead of deposits.

[L-10] Rebalance is limited by bounce redeem min TX size

`AgentTreasury::executeRebalanceStep` computes `shrinkLt` per leg and forwards it to either `lt.redeem` or `lt.prepareRedeem`. Bounce's `LeveragedToken` enforces `baseAmount_ >= globalStorage.minTransactionSize()` on both paths, but the treasury never compares `expectedBaseOut` against that threshold:

```
// AgentTreasury.sol -- executeRebalanceStep sell loop
uint256 expectedBaseOut = lt.ltToBaseAmount(shrinkLt);
if (expectedBaseOut < minRedeemUsdc[i]) revert SlippageExceeded();
// no check against lt.globalStorage.minTransactionSize()

int256 bufferRaw = LT_HELPER.getLeveragedTokenBufferAssetValue(address(lt));
uint256 buffer = bufferRaw < 0 ? 0 : uint256(bufferRaw);
if (expectedBaseOut <= buffer) {
    uint256 actualBaseOut = lt.redeem(address(this), shrinkLt,
        minRedeemUsdc[i]); // reverts BelowMinTransactionSize
    ...
} else {
    ...
    lt.prepareRedeem(shrinkLt); // reverts BelowMinTransactionSize
    ...
}
```

When `expectedBaseOut < minTransactionSize`, the entire `executeRebalanceStep` reverts — one undersized leg blocks all the others in the same call. The rebalancer can only sidestep it by setting `maxRedeemLt[i] = 0` (and `minRedeemUsdc[i] = 0`) for that leg, which then leaves the leg permanently above target.

IMPACT

Rebalance is bricked any time a leg's required shrink falls below Bounce's `minTransactionSize`. The rebalancer must manually pre-compute every leg's `shrinkUsdc` off-chain and zero out the offending caps to make the call go through.

RECOMMENDATION

Document this behaviour. Rebalancer need to choose `maxRedeemLt` values very carefully to avoid this problem.

[L-11] buy can accept payment yet mint 0 shares — no `agentOut > 0` check and no minimum supply

`buy` pulls the user's USDC, deploys it, and mints `_quoteBuy`'s output without checking it is non-zero:

```
agentOut = _quoteBuy(usdcIn, supplyBefore, navBefore);
if (agentOut < minAgentOut) revert SlippageExceeded();   // only the
    user-supplied bound
TREASURY.deployUsdc(usdcIn, minLtOuts);                  // payment taken
_mint(recipient, agentOut);                              // may mint 0
```

A non-zero payment can mint 0 AGENT (value in, nothing out). Normally unreachable — the `1e12` scaling keeps the quote above 1 — but `sell` checks `supplyBefore == 0` *before* the burn, so a holder can drive supply to 1, where `premium(1) == 1e18` collapses the quote to `floor(usdcIn / nav)`. Since `nav()` also counts donations, a sole holder can then make a `minAgentOut == 0` buyer mint 0 and pocket their deposit. Heavily gated: any non-zero `minAgentOut` reverts, needs a sole-holder/genesis state, loss is the buyer's own deposit. `SupplyCollapseInflationPoC.t.sol` confirms both on the HyperEVM fork (no mocks).

IMPACT

Severity: Low — real invariant defect, but unreachable for ordinary buyers and fully prevented by any non-zero `minAgentOut`. Fix as defense-in-depth.

RECOMMENDATION

`require(agentOut > 0)` in `buy` — a paid buy must never mint 0.

[L-12] Paused-leg skip in `_mintLTs` silently ignores the buyer's per-leg `minLtOuts` floor

`_mintLTs` has two branches that produce zero LT for a leg, and it handles them inconsistently. The `usdcToAllocate == 0` branch honors the caller's per-leg floor; the `lt.mintPaused()` branch silently skips the leg without reading it:

```
if (usdcToAllocate == 0) {
    if (minLtOuts[i] != 0) revert SlippageExceeded();   // floor honored
    continue;
}
if (lt.mintPaused()) {
    emit MintSkippedPaused(sym, usdcToAllocate);
    continue;                                           // floor IGNORED
}
USDC.forceApprove(address(lt), usdcToAllocate);
uint256 ltOut = lt.mint(address(this), usdcToAllocate, minLtOuts[i]);
if (ltOut < minLtOuts[i]) revert SlippageExceeded();
```

A buyer who passes `minLtOuts[i] != 0` is explicitly demanding "mint at least N units of leg i or revert". If that leg's LT is paused, the buy nonetheless succeeds with zero units minted and the leg's USDC parked idle — the caller's slippage/exposure floor is silently void. That the sibling branch *does* revert on the same `minLtOuts[i]` shows the floor is meant to be enforced, so this is an inconsistency, not intended behavior.

Example

1. Treasury runs 50/50 `HYPE5L`/`BTC5L`; both legs deployed.
2. `HYPE5L` mint is paused (`mintPaused() == true`, `bps` still 5000).

3. A buyer calls `buy(..., minLtOuts = [1, 0])` — demanding ≥1 unit of `HYPE5L`.
4. The buy **succeeds**: zero `HYPE5L` minted, that leg's USDC sits idle, and `minLtOuts[0] = 1` is silently ignored

IMPACT

The buyer's per-leg exposure floor is unenforceable on any paused leg: they receive unlevered idle-USDC backing instead of the leveraged exposure they required, with no transaction-level signal to abort.

RECOMMENDATION

Mirror the `usdcToAllocate == 0` branch so the floor is honored before skipping a paused leg:

```
  if (lt.mintPaused()) {
+     if (minLtOuts[i] != 0) revert SlippageExceeded();
      emit MintSkippedPaused(sym, usdcToAllocate);
      continue;
  }
```

This reverts only when the buyer demanded exposure to the paused leg; callers who pass `minLtOuts[i] == 0` keep the intended skip-paused resilience.

[L-13] Fee pushed to a hardcoded, unchangeable recipient

`AgentCurve.buy()` and `AgentTreasury.withdrawLtsTo()` send the 1% fee to a hardcoded `FEE_RECIPIENT` on every trade, with no setter:

```
address private constant FEE_RECIPIENT =
    0xb2feD3aCf6e30e0f1902A2b190C88C9a0a68eDC3;
...
if (fee > 0) USDC.safeTransfer(FEE_RECIPIENT, fee);
```

IMPACT

If `FEE_RECIPIENT` is USDC-blacklisted, the transfer reverts and every buy/sell reverts with it. Being a `constant` with no setter, it can't be redirected — only a full redeploy fixes it.

RECOMMENDATION

Make the recipient/bps configurable via a setter, and/or accrue fees for pull-withdrawal so a failing transfer can't block trading.