33labs

# Recoup

## Security Audit Report

# Recoup Security Audit Report

## Introduction

33Labs performed a focused security review of the **Recoup** smart contracts, a self-repaying lending protocol for DexFi Treasury Bonds on Base. The review examined the lender-pool accounting and withdrawal model, protocol wiring, liquidity-source behavior, fee routing, configuration bounds, and loan-to-value mathematics.

## Audit Team

| Auditor | Role |
|---|---|
| 0x23r0 ↗ | Security Researcher |
| PhantomOz ↗ | Security Researcher |

## Disclaimer

A smart contract security review cannot prove the complete absence of vulnerabilities. This assessment was bounded by the agreed scope, review period, available documentation, and referenced commits. Additional review, deployment verification, monitoring, and a public bug bounty are recommended before production use.

## About 33Labs

33Labs is an independent smart contract security research and development group with six years of experience and over 40 completed audits. We specialize in building and auditing Solidity, Rust, Move, and DeFi systems. Learn more at 33labs.ai or follow us on X @33labs_.

## About Recoup

Recoup is a self-repaying loan protocol for DexFi Treasury Bonds on Base. Borrowers supply bonds and borrow USDC; realized bond yield is applied to debt over time. The reviewed system includes lender-pool liquidity, credit management and wiring, liquidation/workout accounting, treasury liquidity, protocol fee distribution, and risk-parameter configuration. The repository states that Recoup is not deployed on Base mainnet and that its Base Sepolia stack uses mocks and does not match the current source.

## Audit Scope

- **Repository:** `thedelph/recoup-contracts`
- **Audited branch:** `main`
- **Audited commit:** `b66023dc4a2ea2c745f0822769f1f82e8f50ca53`
- **In-scope files:**

- `src/LenderPool.sol`
- `src/CreditWiring.sol`
- `src/TreasuryLiquiditySource.sol`
- `src/ProtocolFeeSplitter.sol`
- `src/Config.sol`
- `src/LtvMath.sol`
- **Review window:** September 7–22, 2026

### Remediation Review

Remediation status was reviewed through `f6893cb21ef981a1798cc4a5dc1b4876dfae2d13` on `main`. The CI run for that commit completed successfully. Statuses in this report reflect current code, regression evidence, `KNOWN_RISKS.md`, and maintainer/reviewer discussion—not the GitHub issue open/closed flag.

### Methodology

The review combined manual source analysis, adversarial call-path and state-transition reasoning, Foundry proof-of-concept construction, regression-test review, fix-diff analysis, and remediation verification. Findings were evaluated using impact and likelihood, with liveness, loss allocation, migration safety, access control, accounting consistency, and invariant preservation as primary concerns.

## Severity Definitions

| Severity | Definition |
|---|---|
| Critical | Direct, highly likely catastrophic loss or protocol compromise. |
| High | Material loss, insolvency, privilege compromise, or failure of a core protocol function. |
| Medium | Meaningful security, accounting, or availability impact under realistic conditions. |
| Low | Limited-impact issue, constrained edge case, or hardening gap. |
| Informational | Design, maintainability, UX, or defense-in-depth observation. |

## Findings Summary

| ID | Finding | Severity | Status | Source |
|---|---|---|---|---|
| H-01 | Borrower-keyed recovery provenance redirects an earlier workout's recovery | High | Fixed | Issue #45 |
| H-02 | Auction replacement orphans a closed workout's recovery leg | High | Fixed | Issue #46 |
| H-03 | Repeated request service converts more than a requester's pro-rata cash into senior claims | High | Fixed | Issue #47 |
| H-04 | Manager migration strands post-close loss recoveries with no unblocked repair path | High | Fixed | Issue #53 |

| ID | Finding | Severity | Status | Source |
|---|---|---|---|---|
| M-01 | Permissionless yield sweep can confiscate an open workout's borrower yield | Medium | Fixed | Issue #48 |
| M-02 | Realised open-workout yield bypasses the dedicated sweep protection | Medium | Fixed | Issue #49 |
| M-03 | First clean workout close captures shared residual yield | Medium | Fixed | Issue #50 |
| M-04 | Uncapped stream duration plus gross entry pricing lets a timed flush overcharge new lenders | Medium | Fixed | Issue #51 |
| M-05 | A lender-yield backlog above the deposit-cap ceiling can never be delivered | Medium | Fixed | Issue #52 |
| M-06 | A request serviced down to one share-wei keeps the rest of its cash floor (maintainer-reported) | Medium | Acknowledged / Accepted Risk | Issue #64 |
| L-01 | Permissionless settle discards a borrower's sub-unit yield accrual | Low | Fixed | Issue #54 |
| L-02 | After a raw cash loss, the request floors can lock the whole remaining cash until a repayment or a cancel | Low | Acknowledged / Accepted Risk | Issue #61 |
| L-03 | A paused or blacklisting USDC shuts every bond door, because the farm pays its pending USDC inside the same call (maintainer-reported) | Low | Acknowledged / Accepted Risk | Issue #68 |

# High Findings

## H-01 Borrower-keyed recovery provenance redirects an earlier workout's recovery

**Source:** GitHub issue #45
**Source finding ID:** `H-01`
**Report severity:** High

### Description

**What's wrong**

The protocol remembers *who to repay* for a bad debt by borrower name, not by the specific bad debt. A borrower who defaults twice overwrites the first record with the second. When money for the **first** default finally shows up — and collateral redemptions are quoted at "48h+", so late money is the normal case — it is paid to whoever bore the **second** loss. The pool that actually took the hit gets nothing, and the pool that did not gets paid for a loss it never carried.

**How (mechanism)**

Affected code:

- `src/CreditManager.sol:2078-2079` — `writeDownLoss` overwrites `lossBearerOf[borrower]` and `lossFunderOf[borrower]`.
- `src/CreditManager.sol:2203-2223` — `recoverWrittenDownLoss` resolves the recipient from those borrower-keyed mappings.
- `src/LiquidationAuction.sol:1634` — `workoutSettleAfterClose` supplies only `w.borrower`, even though the closed workout has its own recovery state.

### Exploit path

1. Borrower A's first workout is force-closed and Pool A bears a socialised loss.
2. The liquidity source is migrated to Pool B.
3. Borrower A opens and defaults on a second loan. The second `writeDownLoss` replaces the mappings with Pool B.
4. A late tranche is paid into the first closed workout.
5. `recoverWrittenDownLoss` reads the latest borrower mapping and sends the first workout's recovery to Pool B.

The recovery amount and workout ID are preserved by the auction, but the manager's recipient lookup is mutable and borrower-scoped.

### Impact

Pool A permanently loses recoveries belonging to its previously written-down workout. Pool B receives funds for a loss it did not bear, corrupting lender accounting and allowing a later borrower event to redirect material protocol funds.

**Economic Impact:** PoC — the first workout's entire late recovery, **628,750,000 USDC base units** in the fixture, is delivered to Pool B. Pool A, which bore the loss, receives **0**. Real magnitude is the written-down principal of the earlier default.

## Proof of Concept

Paste into `ImpairmentIntegrationTest` (`test/Impairment.integration.t.sol`) and run it. Every helper it calls is already defined in that fixture — nothing else is needed.

```
/// @notice A second default by the same borrower overwrites the first workout's recovery
///         destination. The late recovery for the first workout reaches the new pool.
function test_POC_recoveryDestinationIsOverwrittenByLaterDefault() public {
    (uint256 oldId, uint256 oldLoss) = _forceCloseOntoThePool();
    assertEq(credit.lossBearerOf(alice), address(pool));

    LenderPool poolB = _poolForNewEra(credit);
    oracle.setNav(NAV);
    vm.prank(alice);
    vault.depositBonds(BONDS);

    uint256 newId = _openAuctionAt(_crashedNav());
    skip(Config.AUCTION_DURATION + 1);
    auction.expireToWorkout(newId);
    skip(Config.WORKOUT_MAX_DURATION + 1);
    vm.prank(stranger);
    auction.closeWorkout(newId);
    assertEq(credit.lossBearerOf(alice), address(poolB));

    uint256 oldPoolBefore = usdc.balanceOf(address(pool));
    uint256 newPoolBefore = usdc.balanceOf(address(poolB));
    _fund(payer, oldLoss);
    vm.prank(payer);
    auction.workoutSettleAfterClose(oldId, oldLoss);

    assertEq(usdc.balanceOf(address(pool)) - oldPoolBefore, 0);
    assertEq(usdc.balanceOf(address(poolB)) - newPoolBefore, oldLoss);
}
```

### Run output

```
$ forge test --match-test test_POC_recoveryDestinationIsOverwrittenByLaterDefault -vv
[PASS] test_POC_recoveryDestinationIsOverwrittenByLaterDefault() (gas: 3913501)
Suite result: ok. 1 passed; 0 failed; 0 skipped
```

## Recommendation

Assign an immutable recovery ID to each write-down/workout and key both bearer and funder records by that ID. Pass the same ID from `workoutSettleAfterClose` into `recoverWrittenDownLoss`; never resolve a closed workout's recovery destination from the borrower's latest state.

## Status

**Fixed** — The recovery bearer and funder are now recorded per auction ID, preventing a later default by the same borrower from redirecting an earlier workout recovery. The reviewers verified the fix on commit `68c0c26`; the current remediation snapshot retains the regression coverage.

---

# H-02 Auction replacement orphans a closed workout's recovery leg

**Source:** GitHub issue #46
**Source finding ID:** `H-02`
**Report severity:** High

## Description

**What's wrong**

Swapping the auction contract is a supported migration, and the code has a guard whose whole job is to refuse an unsafe one. That guard checks whether the old auction has any work left — live auctions, open workouts, bonds held — and all three read zero the moment a workout closes. But closing a workout is exactly what *creates* the obligation to repay a written-down loss later. So the guard waves through a migration that permanently strands that repayment: the old auction is the only contract that can deliver it, and after the swap the manager will not listen to the old auction any more.

**How (mechanism)**

Affected code:

- `src/CreditWiring.sol:588-617` — `checkAuctionSwap` gates the swap on `liveAuctionCount()`, `openWorkoutCount()` and `vault.bondCount(current)`, and on nothing else.
- `src/CreditWiring.sol:801-808` — `_outgoingAuctionWork` reads only those first two counters.
- `src/LiquidationAuction.sol:1432` — `closeWorkout` writes `w.writtenDown`, sets `WorkoutStatus.Closed` and pops `_openWorkouts`, so `openWorkoutCount()` returns to zero in the same transaction that creates the obligation.
- `src/LiquidationAuction.sol:1782` — `disposeWorkoutLot` clears `w.bondCount`, so `vault.bondCount(outgoing)` also returns to zero.
- `src/LiquidationAuction.sol:1601-1634` — `workoutSettleAfterClose` pays the recorded `w.bearer` via `ICreditManager(cm).recoverWrittenDownLoss(...)`.
- `src/CreditManager.sol:2196` — `recoverWrittenDownLoss` calls `_requireAuction()`.
- `src/CreditManager.sol:657-658` — `_requireAuction` reverts unless `msg.sender == liquidationAuction`, the **live mutable pointer**.

### Relationship to round 22 finding F8

`KNOWN_RISKS.md:244` lists F8 — "`workoutSettleAfterClose` resolving its payee from state `closeWorkout` can empty in the same block" — as **closed by recording a `bearer` at every close**. That remediation is real but covers only one of the two pointers:

- **Manager swapped, auction unchanged.** `w.bearer` is the old manager, whose `liquidationAuction` still points at the old auction, so `_requireAuction` passes. F8's fix works here.
- **Auction swapped, manager unchanged.** `w.bearer` is that same manager, but its `liquidationAuction` has moved. `_requireAuction` fails. F8's fix does nothing here.

The docstring at `src/LiquidationAuction.sol:196-199` measures precisely this revert, then argues at `:201-208` for "a field rather than a setter guard" — reasoning applied to `LiquidationAuction.setCreditManager` and never mirrored onto `CreditManager.setLiquidationAuction`. The symmetric hole is open.

### Exploit path

1. A workout is force-closed. `closeWorkout` writes `w.writtenDown = X` (`X > 0`), records `w.bearer`, sets status `Closed`, and pops `_openWorkouts`. `openWorkoutCount()` is now `0`.

2. The owner calls `disposeWorkoutLot`, moving the parked lot. `vault.bondCount(oldAuction)` is now `0`.
3. The owner migrates: `CreditManager.setLiquidationAuction(newAuction)`. `liveAuctions`, `openWorkouts` and `heldLot` all read `0`, every clause passes, the call succeeds and emits `LiquidationAuctionSet`.
4. A DexFi redemption tranche arrives. The round-21 note at `src/LiquidationAuction.sol:179-184` records that redemption is off-chain and quoted at "48h+", so a late tranche is the ordinary case rather than an edge case.
5. Anyone calls `oldAuction.workoutSettleAfterClose(auctionId, amount)`.
6. It reaches `ICreditManager(w.bearer).recoverWrittenDownLoss(...)`. Inside, `_requireAuction()` compares `msg.sender` (the **old** auction) against `liquidationAuction` (now the **new** auction) and reverts `NotLiquidationAuction()`.
7. The whole transaction reverts. `w.writtenDown` is never reduced, and no other path can reduce it.

## Impact

The pool recorded in `lossBearerOf` permanently forfeits every later recovery on that workout. The write-down stays socialised against lenders who should have been made whole by collateral proceeds that did in fact arrive.

No funds are stolen and none are lost in transit — because the call reverts atomically, the late payer keeps their USDC. The loss is the permanent unavailability of the recovery leg: `writtenDown` is monotonically stuck, and the only state that could clear it is unreachable.

The devs' own measurement of the sibling case (`src/LiquidationAuction.sol:193-196`) puts a single forced close at `628.750000` USDC. Real magnitude scales with the written-down principal at migration time.

Reachability is not adversarial: `setLiquidationAuction` is `onlyOwner`, but the owner is running a documented migration that every guard clause approves. `checkAuctionSwap` exists specifically to refuse unsafe migrations, so a migration it approves that still strands funds is a defect in the guard, not misuse of the key.

**Economic Impact:** PoC — the closed workout's `writtenDown` figure stays at its full value and the bearing pool receives **0**. The devs' own measurement of the sibling case (`src/LiquidationAuction.sol:193-196`) puts a single forced close at **628.750000 USDC**; real magnitude scales with written-down principal at migration time.

## Proof of Concept

Paste into `ImpairmentIntegrationTest` (`test/Impairment.integration.t.sol`) and run it. Every helper it calls is already defined in that fixture — nothing else is needed.

```
/// @notice **[H-02]** Replacing only the auction - the manager stays - strands a closed
///         workout's `writtenDown`. Every `checkAuctionSwap` clause reads zero, so the swap
///         reports success, and the outgoing auction's return leg then reverts forever.
/// @dev Distinct from `test_R22_theReturnLegSurvivesAMigrationToAFreshAuction`, which repoints
///      the *incoming* manager and leaves the recorded bearer still naming the old auction.
///      Here the bearer IS the live manager, and it is the one that repointed.
function test_POC_auctionSwapOrphansPostCloseRecovery() public {
    (uint256 id, uint256 writtenDown) = _forceCloseOntoThePool();
    assertEq(credit.lossBearerOf(alice), address(pool), "fixture: the pool bore the loss");

    LiquidationAuction auctionB = new LiquidationAuction(
        usdc,
        ICollateralVault(address(vault)),
        INAVOracle(address(oracle)),
        IRiskParams(address(riskParams)),
        admin
    );

    // The ordinary auction-only migration. No manager is replaced, so `w.bearer` is `credit`.
    vm.startPrank(admin);
```

```
    auction.disposeWorkoutLot(id, admin); // clears vault.bondCount(outgoing)
    vault.setLiquidationAuction(address(auctionB));
    credit.setLiquidationAuction(address(auctionB)); // every guard clause passes
    auctionB.setCreditManager(address(credit));
    vm.stopPrank();

    assertEq(credit.liquidationAuction(), address(auctionB), "fixture: the swap did not take");

    uint256 poolCashBefore = usdc.balanceOf(address(pool));
    _fund(payer, writtenDown);
    vm.prank(payer);
    vm.expectRevert(CreditManager.NotLiquidationAuction.selector);
    auction.workoutSettleAfterClose(id, writtenDown);

    (,,,,,,, uint256 stillWrittenDown,,,) = auction.workouts(id);
    assertEq(stillWrittenDown, writtenDown, "the written-down figure is now unreachable");
    assertEq(usdc.balanceOf(address(pool)), poolCashBefore, "the bearer was paid nothing");
}
```

### Run output

```
$ forge test --match-test test_POC_auctionSwapOrphansPostCloseRecovery -vv
[PASS] test_POC_auctionSwapOrphansPostCloseRecovery() (gas: 2109234)
Suite result: ok. 1 passed; 0 failed; 0 skipped
```

## Recommendation

Do **not** add a fourth clause refusing the swap while `writtenDown` stands — `src/CreditManager.sol:2165-2169` already rejects that shape, correctly: nothing obliges a redemption to arrive, so the guard would weld the pointer shut forever.

Apply the same remedy the `bearer` field applied to the other pointer — authorize the recovery leg against a recorded identity rather than the live pointer:

```
mapping(address => bool) public wasLiquidationAuction;

function setLiquidationAuction(address liquidationAuction_) external onlyOwner {
    _requireWiringRan(CreditWiring.checkAuctionSwap(liquidationAuction, liquidationAuction_, vault));
    if (liquidationAuction != address(0)) wasLiquidationAuction[liquidationAuction] = true;
    liquidationAuction = liquidationAuction_;
    emit LiquidationAuctionSet(liquidationAuction_);
}

function _requireAuctionOrFormer() private view {
    if (msg.sender != liquidationAuction && !wasLiquidationAuction[msg.sender]) {
        revert NotLiquidationAuction();
    }
}
```

Call `_requireAuctionOrFormer()` from `recoverWrittenDownLoss` only. Every other `_requireAuction()` site stays on the live pointer — a former auction must keep its return leg alive without regaining any authority to open auctions, move bonds, or write down new losses.

## Status

**Fixed** — Former liquidation auctions remain authorized for the recovery leg after an auction migration, while no additional auction-gated privileges are exposed. The reviewers verified the fix on commit `68c0c26`.

---

# H-03 Repeated request service converts more than a requester's pro-rata cash into senior claims

**Source:** GitHub issue #47
**Source finding ID:** `H-03`
**Report severity:** High

## Description

### What's wrong

A lender who queues a withdrawal is supposed to get their fair share of the pool's spare cash. The pool works out that share fresh on every call, against whatever cash is left — so if you take it in many small bites instead of one, each bite is recalculated against a smaller pot and the slices add up to far more than your share. You walk out with cash-backed, first-in-line claims; everyone still in the pool is left holding the lending exposure. If the loan book then goes bad, you already left at full value and the loss lands on the people who did not run the loop.

### How (mechanism)

Affected code:

- `src/LenderPool.sol:2256-2270` — `maxRequestRedeem` computes `requestCash = _executablePoolCash(raw) * requestedShares / totalSupply()` each time it is called.
- `src/LenderPool.sol:2295-2332` — `serviceWithdrawalRequest` burns the serviced shares, decrements `request.shares`, and adds the payout to `claimable[receiver]` / `totalClaimable`. The remainder of the request survives and is immediately serviceable again.
- `src/LenderPool.sol:691-695` — `_poolBalance` subtracts `totalClaimable`, so each serviced payout leaves the executable-cash base.
- `src/LenderPool.sol:553-557` — `_totalAssets` likewise excludes `totalClaimable`, seniorising the payout ahead of remaining shareholders.
- `src/LenderPool.sol:2347-2361` — `_claim` transfers the crystallised claim out, physically removing the captured cash before outstanding principal resolves.

The entitlement is recomputed, never accumulated. Writing `C` for executable cash, `R` for the request's remaining shares and `S` for total supply, each call pays `a_i = C_i · R_i / S_i` and leaves `C_{i+1} = C_i − a_i`. Total capture across `n` calls is therefore

```
Σ a_i = C · (1 − Π (1 − f_i)),   f_i = R_i / S_i
```

which tends toward the **entire** executable cash balance, not the fair single-shot share `C · f_1`.

Worked example — `C = 100` cash, `P = 100` outstanding principal, `S = 100` supply, `R = 50` attacker shares. Fair pro-rata cash is `50`:

| call | payout | cash remaining |
|---|---|---|
| 1 | 50.00 | 50.00 |
| 2 | 16.67 | 33.33 |
| 3 | 8.33 | 25.00 |

| call | payout | cash remaining |
|---|---|---|
| 4 | 5.00 | 20.00 |
| … | → | → 0 |

The fair share is already exceeded on call 2, and the series continues toward capturing all of `C`.

### Exploit path

1. Attacker holds `R` shares of a pool with non-zero `outstandingPrincipal`.
2. `requestWithdrawal(R, attacker)` escrows the shares.
3. Loop `serviceWithdrawalRequest(attacker, maxRequestRedeem(attacker), 0)` until `maxRequestRedeem` returns 0.
4. `claim()` collects the crystallised USDC.

`nonReentrant` blocks nested re-entry but not sequential calls, so steps 3–4 run from a contract in a single transaction.

Per-share pricing is honest throughout — `previewRedeem` is not mispriced, and the attacker is not paid more per share than anyone else. The defect is compositional: the attacker exits into **senior, fixed, cash-backed claims** while the remaining lenders are left holding the junior principal exposure. If the credit book later takes a loss, the attacker has already exited at par and the loss concentrates on everyone else.

The cap at `src/LenderPool.sol:2265-2269` (`burnable = supply − MIN_SUPPLY_FOR_YIELD` while `outstandingPrincipal != 0`) bounds the total shares burned but does not interrupt the iteration or restore a cumulative entitlement.

## Impact

- **Remaining lenders** absorb an outsized share of `outstandingPrincipal`: a requester with fraction `f` of supply can convert well beyond `f · C` of idle cash into senior claims, so the pool's cash-to-principal ratio degrades for everyone who did not run the loop.
- **Any subsequent credit write-down** falls on a smaller, more concentrated junior base, magnifying each remaining lender's realised loss.
- Requires no privileged role, no flash loan, and no external market conditions — only an existing share balance and a pool with non-zero outstanding principal.

**Economic Impact:** PoC — fair single-shot pro-rata cash **4,874.250000 USDC**; captured by repeated service **4,999.999999 USDC**. Excess **125.749999 USDC, ~2.58%** of the request, converted from shared junior cash into a senior fixed claim. The series tends toward the *entire* executable cash balance as share fraction and call count rise.

## Proof of Concept

Paste into `ImpairmentIntegrationTest` (`test/Impairment.integration.t.sol`) and run it. Every helper it calls is already defined in that fixture — nothing else is needed.

```
/// @notice **[H-03]** `maxRequestRedeem` recomputes a request's cash slice against a base the
///         previous service already shrank, so servicing in steps captures strictly more than
///         one pro-rata share of executable cash - the rest of the pool keeps the principal.
function test_POC_repeatedRequestServiceCapturesSharedCash() public {
    address attacker = makeAddr("attacker");
    uint256 stake = 5_000e6; // remaining headroom under the deposit cap
```

```
    _lend(attacker, stake);

    // Principal at risk is what makes a cash exit senior rather than neutral.
    uint256 debt = _maxBorrowAtCeiling();
    vm.prank(alice);
    credit.borrow(debt);
    assertGt(pool.outstandingPrincipal(), 0, "fixture: no principal at risk");

    // Read before the prank: a staticcall in argument position spends it (see `_openAuctionAt`).
    uint256 attackerShares = pool.balanceOf(attacker);
    vm.prank(attacker);
    pool.requestWithdrawal(attackerShares, attacker);

    // The honest entitlement: one pro-rata slice of executable cash, taken in a single call.
    uint256 fairOnce = pool.previewRedeem(pool.maxRequestRedeem(attacker));
    assertGt(fairOnce, 0, "fixture: nothing was serviceable");

    uint256 captured;
    for (uint256 i = 0; i < 64; i++) {
        uint256 serviceable = pool.maxRequestRedeem(attacker);
        if (serviceable == 0) break;
        vm.prank(attacker);
        captured += pool.serviceWithdrawalRequest(attacker, serviceable, 0);
    }

    emit log_named_uint("MEASURED fair single-shot pro-rata cash", fairOnce);
    emit log_named_uint("MEASURED captured by repeated service   ", captured);
    assertGt(captured, fairOnce, "repeated service captured no more than the fair slice");
}
```

### Run output

```
$ forge test --match-test test_POC_repeatedRequestServiceCapturesSharedCash -vv
[PASS] test_POC_repeatedRequestServiceCapturesSharedCash() (gas: 1794326)
Logs:
  MEASURED fair single-shot pro-rata cash: 4874250000
  MEASURED captured by repeated service   : 4999999999
Suite result: ok. 1 passed; 0 failed; 0 skipped
```

## Recommendation

Fix the request's cash entitlement once, rather than recomputing it against a shrinking base. Record the entitlement when the request is created and spend it down as the request is serviced:

```
struct WithdrawalRequest {
    uint256 requestId;
    address receiver;
    uint256 shares;
    uint256 cashEntitlement; // new: set at requestWithdrawal, decremented on service
}
```

Set `cashEntitlement = _executablePoolCash(_rawBalance()) * shares / totalSupply()` in `requestWithdrawal`, cap `maxRequestRedeem` at `convertToShares(request.cashEntitlement)`, and decrement it by `assetsOut` in `serviceWithdrawalRequest`. A requester then draws at most one pro-rata slice of the cash that existed when they queued, however many calls they split it across.

### Status

**Fixed** — Per-controller request-draw memory closes repeated-service amplification, and a per-request cash floor closes the remaining synchronous and cross-controller drain routes. The reviewers verified the original High-severity routes on commit `68c0c26`. The post-loss liveness cost of the cash floor is tracked separately as report finding L-02 / GitHub issue #61.

---

## H-04 Manager migration strands post-close loss recoveries with no unblocked repair path

**Source:** GitHub issue #53
**Source finding ID:** `L-01`
**Report severity:** High

### Description

#### What's wrong

The pool will only accept loss repayments from the manager it currently points at. Swapping to a new manager is allowed even while an old loss is still waiting to be repaid — so afterwards the old manager routes the repayment correctly, the pool refuses it, and the money bounces. The code acknowledges this and says to just point the pool back. But pointing back is itself refused once the new manager has lent anything, so the suggested repair is unavailable exactly when it is needed.

#### How (mechanism)

Affected code:

- `src/LenderPool.sol:475-491` — `setCreditManager` refuses only on `outstandingPrincipal != 0` and `totalImpairment != 0`. Neither term reflects a former manager's unrecovered write-downs.
- `src/LenderPool.sol:2137` — `recoverLoss` reverts `NotCreditManager` unless `msg.sender == creditManager`, the live mutable pointer.
- `src/CreditManager.sol:2203-2209` — `recoverWrittenDownLoss` makes a bare `ILenderPool(pool).recoverLoss(amount)` call to the recorded `lossBearerOf[borrower]`, with no parking fallback.
- `src/CreditManager.sol:2220-2228` — the sibling *funder* branch **does** park, via `owedToSource[funder] += amount`. The pool branch has no equivalent.
- `src/CreditManager.sol:625` — `recoverWrittenDownLoss` deliberately omits `whileAttached`, confirming post-detachment recovery is intended.

#### Why this is High, not the Low it was filed as

`src/CreditManager.sol:2165-2185` documents the trade explicitly. The devs considered and rejected refusing the repoint while a write-off stands ("welds both pointers shut for good, since nothing obliges a redemption to ever arrive"), and state the resulting stance plainly: *"a pool that no longer recognises this manager will refuse `recoverLoss`, and the caller keeps their USDC and can retry once the pointer is repaired."*

**The Low rating rested on that retry being available. It is not, and our own PoC proves it.** The second half of the reproduction below shows the repair blocked: once the successor manager has made a single borrow, `setCreditManager` back to the bearing manager reverts `PrincipalOutstanding`. So the correct reading is a **permanent refusal**, not a recoverable delay — and permanently undeliverable recovery of a socialised loss is High, not Low.

We under-rated this by filing the "funds are not lost in transit" observation as the whole impact. That is true and irrelevant: the payer keeping their USDC does not make the bearing pool's lenders whole. The maintainer had already rated it High internally and had retracted the "retry once repaired" sentence in source for the same reason — there is no state for the retry to wait for.

### Exploit path

1. A workout closes; Pool A bears a socialised write-down, recorded in `lossBearerOf`.
2. Principal winds down; `outstandingPrincipal` and `totalImpairment` reach zero.
3. The owner migrates the pool to Manager B via `setCreditManager`. Both guards pass.
4. A late tranche reaches the old manager, which routes to Pool A per the recorded bearer.
5. `Pool A.recoverLoss` sees `msg.sender != creditManager` (now Manager B) and reverts.

## Impact

- **Pool A's lenders** cannot receive recoveries on losses they bore until the pool is repointed back, which may require fully unwinding the successor manager's book.
- No fund loss and no theft: the tranche stays with its payer and can be redelivered once pointers align.
- Owner-gated and operationally repairable, so this is a liveness and process concern rather than a solvency one.

**Economic Impact:** PoC — the late tranche reverts `NotCreditManager` and the bearing pool receives **0**. No funds are lost in transit: the payer keeps their USDC. The loss is availability of the recovery leg for as long as the successor manager holds credit.

### Severity — real-world impact × likelihood

- **Impact: High.** The lenders who bore a socialised write-down never receive its recovery. The money exists, arrives, and is correctly routed — and is refused forever. No owner action clears it once the successor has lent, so this is permanent loss of a recovery leg, not a delay.
- **Likelihood: Medium.** `setCreditManager` is owner-only, but every guard it does enforce passes in this state, so an ordinary honest migration reaches it without any misuse of the key. One borrow through the replacement makes it irreversible.
- **High × Medium → High.**

## Proof of Concept

Paste into `ImpairmentIntegrationTest` (`test/Impairment.integration.t.sol`) and run it. Every helper it calls is already defined in that fixture — nothing else is needed.

```
/// @notice **[L-01]** Repointing the pool to a new manager leaves the former manager unable to
///         deliver a recovery it correctly routed: `recoverLoss` authorises only the live
///         pointer. Acknowledged at `CreditManager.sol:2182-2185`; asserted here.
function test_POC_managerMigrationStrandsPostCloseRecovery() public {
    (uint256 id, uint256 writtenDown) = _forceCloseOntoThePool();
    assertEq(credit.lossBearerOf(alice), address(pool), "fixture: the pool bore the loss");

    CreditManager incoming = _migrate();
    vm.startPrank(admin);
    pool.setCreditManager(address(incoming)); // the pool now names the successor
    incoming.setLiquiditySource(address(pool));
    incoming.setLenderPool(address(pool));
    vm.stopPrank();

    uint256 poolCashBefore = usdc.balanceOf(address(pool));
    _fund(payer, writtenDown);
```

```
    vm.prank(payer);
    vm.expectRevert(LenderPool.NotCreditManager.selector);
    auction.workoutSettleAfterClose(id, writtenDown);

    assertEq(usdc.balanceOf(address(pool)), poolCashBefore, "the bearer received nothing");

    // The acknowledged repair - repoint back - is unavailable while the successor has credit.
    oracle.setNav(NAV);
    vm.prank(alice);
    vault.depositBonds(BONDS);
    vm.prank(alice);
    incoming.borrow(500e6); // inside the 25% LTV ceiling
    uint256 live = pool.outstandingPrincipal(); // read before the prank
    vm.prank(admin);
    vm.expectRevert(abi.encodeWithSelector(LenderPool.PrincipalOutstanding.selector, live));
    pool.setCreditManager(address(credit));
}
```

### Run output

```
$ forge test --match-test test_POC_managerMigrationStrandsPostCloseRecovery -vv
[PASS] test_POC_managerMigrationStrandsPostCloseRecovery() (gas: 2755326)
Suite result: ok. 1 passed; 0 failed; 0 skipped
```

## Recommendation

Give the pool branch the parking fallback the funder branch already has, so a mismatched pointer defers the payment instead of reverting it:

```
address pool = lossBearerOf[borrower];
if (pool != address(0)) {
    if (ILenderPool(pool).creditManager() == address(this)) {
        _approveUsdc(pool, amount);
        ILenderPool(pool).recoverLoss(amount);
        _approveUsdc(pool, 0);
    } else {
        owedToPool[pool] += amount;   // mirrors owedToSource
        emit RecoveryParked(pool, amount);
    }
    emit WrittenDownLossRecovered(borrower, amount, pool);
    return;
}
```

Pair it with a permissionless `flushRecoveryTo(address pool)` that delivers a parked balance once the pointer agrees — the same shape as `flushPrincipalTo` for `owedToSource` and `EpochHarvester.owedToPool`. This matches the stated design intent at `CreditManager.sol:2170-2172` ("the outgoing pool keeps a claim on this manager for as long as this manager exists"), which the current reverting call does not actually implement.

## Status

**Fixed** — This issue was re-rated from Low to High after the permanent-stranding path was confirmed. Former credit managers can now deliver recoveries to the recorded pool after a legal migration without gaining access to other manager-gated functions.

# Medium Findings

## M-01 Permissionless yield sweep can confiscate an open workout's borrower yield

**Source:** GitHub issue #48
**Source finding ID:** `M-01`
**Report severity:** Medium

### Description

#### What's wrong

While a borrower is working out a defaulted loan, their collateral keeps earning yield that belongs to them if they repay cleanly. Anyone at all can push that yield into the insurance fund before the workout finishes. The borrower then repays in full, closes cleanly with no loss recorded — and finds their earned yield is gone, because the money backing it was already moved somewhere they cannot reach.

#### How (mechanism)

At `src/LiquidationAuction.sol:1668-1678`, the sweep protects only `totalUnclaimedRewards` and `totalWorkoutYieldOwed`. Yield belonging to open workouts is not included in either amount. Any caller can therefore sweep the auction's currently realised claim into `CreditManager.insuranceFund()`.

### Exploit path

1. Wait until a workout lot accrues yield.
2. Call `sweepWorkoutYieldToInsurance()` as an unrelated address.
3. Let the borrower repay the workout in full.
4. Observe the clean close record no borrower yield because the backing was already swept.

### Impact

The borrower loses earned yield despite repaying the workout without a write-down. The yield is irreversibly credited to insurance, creating a material accounting loss for the identifiable borrower.

**Economic Impact:** PoC — a `1,000.000000 USDC` streamed pot is swept to insurance by an unrelated address; the borrower then repays in full, closes with `owed == 0`, and `claimWorkoutYield` reverts `NothingToClaim`. Loss is the full accrued yield of the open workout.

### Severity — real-world impact × likelihood

- **Impact: Medium.** A clean-repaying borrower loses their full earned workout yield, irreversibly credited to the insurance fund. No protocol insolvency and no attacker profit — the value moves to insurance, not to the caller.
- **Likelihood: High.** `sweepWorkoutYieldToInsurance` is fully permissionless, needs no capital and no special state beyond an open workout that has accrued yield, which is the ordinary condition during any workout.
- **Medium × High → Medium.**

### Proof of Concept

Paste into `ImpairmentIntegrationTest` (`test/Impairment.integration.t.sol`) and run it. Every helper it calls is already defined in that fixture — nothing else is needed.

```
/// @notice A stranger can sweep an open workout's earned yield before the workout closes.
function test_POC_openWorkoutYieldCanBeSweptToInsurance() public {
    uint256 id = _openAuctionAt(_crashedNav());
    skip(Config.AUCTION_DURATION + 1);
    auction.expireToWorkout(id);
    _streamYieldTo(1_000e6);

    uint256 insuranceBefore = credit.insuranceFund();
    vm.prank(stranger);
    auction.sweepWorkoutYieldToInsurance();
    assertGt(credit.insuranceFund() - insuranceBefore, 0);

    uint256 payment = credit.currentDebtOf(alice) * 2;
    _fund(payer, payment);
    vm.prank(payer);
    auction.workoutSettle(id, payment);
    auction.closeWorkout(id);

    (,,,,,,, uint256 writtenDown,,, uint256 owed) = auction.workouts(id);
    assertEq(writtenDown, 0);
    assertEq(owed, 0);
    vm.expectRevert(LiquidationAuction.NothingToClaim.selector);
    auction.claimWorkoutYield(id);
}
```

#### Run output

```
$ forge test --match-test test_POC_openWorkoutYieldCanBeSweptToInsurance -vv
[PASS] test_POC_openWorkoutYieldCanBeSweptToInsurance() (gas: 2416297)
Suite result: ok. 1 passed; 0 failed; 0 skipped
```

### Recommendation

Do not sweep workout yield while any workout is open. The smallest guard is to revert when `_openWorkouts.length != 0`; a more granular design can track and reserve each open workout's entitlement until its terminal transition.

### Status

**Fixed** — Both pre-close sweep routes now reserve open-workout accrual through `_fundInsuranceWithFree`, preventing permissionless confiscation. The reviewers verified the fix on commit `68c0c26`.

---

## M-02 Realised open-workout yield bypasses the dedicated sweep protection

**Source:** GitHub issue #49
**Source finding ID:** `M-02`
**Report severity:** Medium

## Description

### What's wrong

This is the same theft as M-01 through a different door. Even if the obvious sweep is locked, an attacker can first *pull* the open workout's yield into the auction using a permissionless call, and then use a **second** sweep — one that only reserves closed workouts and liquidations — to move it to insurance. Guarding the first door alone does not close this one.

### How (mechanism)

`CreditManager.claimSurplusFor(address)` is permissionless and accepts the auction as its recipient. At `src/LiquidationAuction.sol:1750-1759`, `sweepFreeBalanceToInsurance` treats the resulting balance as free because it subtracts only closed-workout and liquidation claims. Open-workout yield is not reserved.

### Exploit path

1. An open workout accrues yield.
2. An arbitrary caller invokes `credit.claimSurplusFor(address(auction))`.
3. The caller invokes `auction.sweepFreeBalanceToInsurance()`.
4. The realised open-workout yield is transferred to insurance before the workout closes.

## Impact

The open workout's earned yield is removed from the auction's borrower-owned accounting and becomes insurance funds. This leaves the borrower unable to claim the yield even though the claim was successfully realised on-chain.

**Economic Impact:** PoC — the auction's realised balance is moved to insurance in full: `insuranceFund` rises by exactly the realised open-workout yield. Same magnitude as M-01, reached without touching `sweepWorkoutYieldToInsurance`.

### Severity — real-world impact × likelihood

- **Impact: Medium.** Identical to M-01 — the borrower's realised yield becomes insurance funds and is unclaimable.
- **Likelihood: High.** Both legs are permissionless (`claimSurplusFor`, `sweepFreeBalanceToInsurance`) and need no capital. Critically, this path stays open even after M-01's obvious guard is applied.
- **Medium × High → Medium.**

## Proof of Concept

Paste into `ImpairmentIntegrationTest` (`test/Impairment.integration.t.sol`) and run it. Every helper it calls is already defined in that fixture — nothing else is needed.

```
/// @notice Realising an open workout's yield into the auction first exposes the alternate
///         free-balance sweep path.
function test_POC_realisedOpenWorkoutYieldCanBeSweptToInsurance() public {
    uint256 id = _openAuctionAt(_crashedNav());
    skip(Config.AUCTION_DURATION + 1);
    auction.expireToWorkout(id);
    _streamYieldTo(1_000e6);

    vm.prank(stranger);
    credit.claimSurplusFor(address(auction));
    uint256 auctionBalance = usdc.balanceOf(address(auction));
    assertGt(auctionBalance, 0);
```

```
    uint256 insuranceBefore = credit.insuranceFund();
    vm.prank(stranger);
    auction.sweepFreeBalanceToInsurance();
    assertEq(credit.insuranceFund() - insuranceBefore, auctionBalance);
}
```

### Run output

```
$ forge test --match-test test_POC_realisedOpenWorkoutYieldCanBeSweptToInsurance -vv
[PASS] test_POC_realisedOpenWorkoutYieldCanBeSweptToInsurance() (gas: 1869227)
Suite result: ok. 1 passed; 0 failed; 0 skipped
```

## Recommendation

Apply the same live-workout reservation rule to `sweepFreeBalanceToInsurance`: reject the sweep while an open workout exists, or maintain per-workout reserved entitlements that are excluded from free-balance calculations.

## Status

**Fixed** — The reserve is derived from the workout accumulator rather than the current cash location, so realizing the yield first no longer bypasses sweep protection. Current-source regression tests cover the sibling route.

---

# M-03 First clean workout close captures shared residual yield

**Source:** GitHub issue #50
**Source finding ID:** `M-03`
**Report severity:** Medium

## Description

### What's wrong

When two borrowers are both working out defaults, they share one pot of accrued yield. The pot is not divided between them — each closing workout takes as much as it can from the shared balance. So whoever closes first takes the remainder and the second borrower gets nothing, even though both earned part of it. Who gets paid is decided by close ordering, which anyone can influence.

### How (mechanism)

At `src/LiquidationAuction.sol:1491-1536`, each workout recomputes its historical entitlement from `yieldIndexAtOpen`, then caps that entitlement against one aggregate `reachable` balance. The function does not track which portion of the remaining balance belongs to each open workout.

### Exploit path

1. Two workouts are open and an earlier yield pot is swept.
2. A later epoch produces a shared residual balance.
3. Both borrowers repay cleanly.
4. The first workout is closed and consumes the residual balance.

5. The second workout closes with zero yield despite having earned part of that epoch.

## Impact

Close ordering determines which borrower receives the remaining yield. The second borrower suffers a direct loss while the first borrower receives funds that include the second workout's share.

**Economic Impact:** PoC — two clean workouts, both entitled; the first records positive `yieldOwed` and the second records **0**. Loss to the second borrower is its whole share of the shared residual epoch.

### Severity — real-world impact × likelihood

- **Impact: Medium.** One borrower's earned yield is redirected to another. Bounded by the shared residual epoch, and value stays with a borrower rather than leaving the system.
- **Likelihood: Medium.** Needs two concurrent workouts and a prior sweep. Close ordering is permissionless, so the deciding step is freely reachable once that state exists.
- **Medium × Medium → Medium.**

## Proof of Concept

Paste into `ImpairmentIntegrationTest` (`test/Impairment.integration.t.sol`) and run it. Every helper it calls is already defined in that fixture — nothing else is needed.

```
/// @notice After a prior sweep, the first clean workout close consumes the remaining shared
///         pot and the second clean workout receives nothing.
function test_POC_firstCloseCapturesSharedResidualYield() public {
    (uint256 first, uint256 second) = _twoCleanCloses(makeAddr("secondBorrower"));
    (,,,,,,,,,, uint256 firstOwed) = auction.workouts(first);
    (,,,,,,,,,, uint256 secondOwed) = auction.workouts(second);

    assertGt(firstOwed, 0);
    assertEq(secondOwed, 0);
}
```

### Run output

```
$ forge test --match-test test_POC_firstCloseCapturesSharedResidualYield -vv
[PASS] test_POC_firstCloseCapturesSharedResidualYield() (gas: 4561084)
Suite result: ok. 1 passed; 0 failed; 0 skipped
```

## Recommendation

Track unswept yield entitlement per workout and debit each workout's entitlement whenever yield is redirected. At close, book only that workout's remaining entitlement rather than capping historical entitlements against one shared balance.

## Status

**Fixed** — Open-workout accrual is reserved and the owed ledger is maintained per bearer, so one clean close can no longer capture residual yield owed to another workout.

---

# M-04 Uncapped stream duration plus gross entry pricing lets a timed flush overcharge new lenders

**Source:** GitHub issue #51
**Source finding ID:** `M-04`
**Report severity:** Medium

## Description

### What's wrong

When a new lender deposits, they are charged for yield the pool has collected but not yet handed out — they pay the full sticker price up front, then receive it gradually over a vesting window. Exiting early forfeits the part that has not vested. The window is set to however long it has been since the last delivery, with a floor but **no ceiling**, and the call that starts it is open to anyone. After a long quiet spell, anyone can fire that call in the block right before a deposit lands, so the newcomer pays in full for a pot that now vests over months. Leave early and the difference stays with the lenders who were already there.

### How (mechanism)

Affected code:

- `src/LenderPool.sol:1044-1047` — `_entryAssets` is `_totalAssets(raw)` **plus** `_effectiveUnreleasedYield(raw)`.
- `src/LenderPool.sol:1124-1128` — `_exitAssets` is `_totalAssets(raw)` **minus** `exitReserve()`.
- `src/LenderPool.sol:553-557` — `_totalAssets` excludes `_effectiveUnreleasedYield` outright.
- `src/LenderPool.sol:1886-1887` — `_rateStream` sets `duration = elapsed > Config.YIELD_STREAM_DURATION ? elapsed : Config.YIELD_STREAM_DURATION`, where `elapsed = block.timestamp - lastYieldDistributeAt`. The floor is five days; there is **no ceiling**.
- `src/EpochHarvester.sol:420-428` — `harvest()` is `external nonReentrant` with no access modifier, throttled only by `Config.MIN_EPOCH_GAP` since `lastHarvestAt`.

The three legs compose:

1. **Entry pays gross.** A depositor is priced against `_totalAssets + unreleasedYield`, so they buy into the whole undelivered pot at face value.
2. **Exit pays net.** Redemption is priced against `_totalAssets`, which excludes that pot. Value is realised only as the stream releases.
3. **The window is attacker-timed and unbounded.** After a long gap the pot vests over that entire historical gap, and a permissionless `harvest()` picks the block the gap is measured in.

## Exploit path

1. Harvest delivery lapses for an extended period (keeper outage, a run of declined zero-yield epochs, or the gap before the pool is wired — all three named in the in-code rationale at `src/LenderPool.sol:1877-1886`).
2. A victim's `deposit` sits in the mempool.
3. An unprivileged caller front-runs it with `EpochHarvester.harvest()`. `_rateStream(streamable, true)` sets `duration = elapsed`, an arbitrarily long window, and the whole pot becomes unreleased yield.
4. The victim's deposit prices against `_entryAssets`, paying gross for the full pot.
5. The victim exits before the stream completes. Their share of the still-unreleased remainder stays in the pool and accrues to the cohort that held shares before the delivery — including the caller from step 3.

The long duration is deliberate and the rationale at `:1877-1886` is sound: paying a sixty-day pot over five days would allow same-block capture, so `duration ≥ elapsed` is the correct direction. What is missing is a ceiling. The design bounds the window from below (`YIELD_STREAM_DURATION`) and leaves it unbounded from above, while the entry side charges the full pot immediately.

## Impact

- **New depositors** pay for yield on a vesting horizon they may not be able to hold, and forfeit the unreleased remainder if they exit early. The forfeit is proportional to their stake and to how much of the window remains.
- **The pre-delivery cohort** receives that forfeited value — so the transfer has a beneficiary, and an attacker holding shares can position for it deliberately by choosing the flush block.
- No protocol-level insolvency and no theft from the pool: this redistributes value between lender cohorts. Severity rests on the size of the delivered pot and the length of the stale gap, both of which grow with outage duration.

**Economic Impact:** PoC — after a 180-day gap the flush sets a **15,552,000-second (180-day)** vesting window against a five-day floor. A lender depositing **3,000.000000 USDC** can immediately redeem only **2,875.000000** — **125.000000 USDC, ~4.17%**, under water on arrival, forfeited to the pre-delivery cohort if they exit before the stream completes.

### Severity — real-world impact × likelihood

- **Impact: Medium.** Value is redistributed between lender cohorts — no protocol insolvency and no theft from the pool. Magnitude scales with pot size and outage length; the PoC measures 4.17% on arrival.
- **Likelihood: Medium.** Requires a genuine harvest outage, which the in-code rationale names three ordinary causes for. Once present, the flush is permissionless and the block is chosen, so the attacker picks the moment.
- **Medium × Medium → Medium.**

## Proof of Concept

Paste into `ImpairmentIntegrationTest` (`test/Impairment.integration.t.sol`) and run it. Every helper it calls is already defined in that fixture — nothing else is needed.

```
/// @notice **[M-04]** After a long delivery gap a permissionless flush rates the whole pot over
///         that entire gap, while entry pricing charges the newcomer for it gross. A lender who
///         enters straight after is immediately under water on `previewRedeem`.
function test_POC_staleEpochClockOverchargesANewLender() public {
    skip(180 days); // harvest outage: keeper down, or a run of declined zero-yield epochs

    uint256 pot = 1_000e6;
    usdc.mint(harvester, pot);
    vm.startPrank(harvester);
    usdc.approve(address(pool), pot);
    pool.distributeYield(pot); // the epoch leg: sets duration = elapsed since last delivery
    vm.stopPrank();

    assertGt(pool.unreleasedYield(), 0, "fixture: nothing was left unreleased");
    emit log_named_uint("MEASURED stream window (seconds)", pool.yieldStreamEndsAt() - block.timestamp);
    assertGt(
        pool.yieldStreamEndsAt() - block.timestamp,
        Config.YIELD_STREAM_DURATION,
        "the window did not stretch past the floor"
    );

    address victim = makeAddr("victim");
    uint256 stake = 3_000e6; // inside the remaining deposit-cap headroom
    _lend(victim, stake);

    uint256 out = pool.previewRedeem(pool.balanceOf(victim));
```

```
    emit log_named_uint("MEASURED deposited           ", stake);
    emit log_named_uint("MEASURED redeemable at once  ", out);
    assertLt(out, stake, "the newcomer did not pay for the unreleased pot");
}
```

### Run output

```
$ forge test --match-test test_POC_staleEpochClockOverchargesANewLender -vv
[PASS] test_POC_staleEpochClockOverchargesANewLender() (gas: 564288)
Logs:
  MEASURED stream window (seconds): 15552000
  MEASURED deposited           : 3000000000
  MEASURED redeemable at once  : 2875000000
Suite result: ok. 1 passed; 0 failed; 0 skipped
```

## Recommendation

Keep `duration ≥ elapsed` for its anti-capture property but bound it from above, so a stale clock cannot impose an unbounded vesting horizon:

```
uint256 elapsed = block.timestamp - lastYieldDistributeAt;
duration = elapsed > Config.YIELD_STREAM_DURATION ? elapsed : Config.YIELD_STREAM_DURATION;
if (duration > Config.MAX_YIELD_STREAM_DURATION) duration = Config.MAX_YIELD_STREAM_DURATION;
```

Choose `MAX_YIELD_STREAM_DURATION` above any plausible honest outage (30 days is a reasonable starting point) so ordinary operations are unaffected. If the entry-side charge is intended to stay gross, document the resulting minimum sensible holding period; otherwise consider excluding the unreleased pot from `_entryAssets` and defending same-block capture with the stream window alone.

## Status

**Fixed** — The epoch leg of `_rateStream` is capped by `MAX_YIELD_STREAM_DURATION` at 30 days, bounding the stale-epoch window. Gross active-tail entry pricing remains an explicit disclosed design choice rather than part of this remediation.

---

# M-05 A lender-yield backlog above the deposit-cap ceiling can never be delivered

**Source:** GitHub issue #52
**Source finding ID:** `M-05`
**Report severity:** Medium

## Description

### What's wrong

The harvester always offers the pool its **entire** backlog of lender yield in one go. The pool refuses any offer bigger than the money it currently holds. The pool's size is capped by a hard-coded constant that cannot be raised without redeploying. So once the backlog grows past that cap the offer is permanently too big, the offer cannot be made smaller, and the pool cannot be made bigger. The yield is stuck forever, with no admin lever to free it.

**How (mechanism)**

Affected code:

- `src/LenderPool.sol:1816-1820` — `distributeYield` reverts `YieldExceedsCapital(streamable, capital)` when `liquidityDeficitBefore == 0 && streamable > capital`.
- `src/LenderPool.sol:353-362` — `setDepositCap` reverts `DepositCapTooLarge` above `Config.GLOBAL_BORROW_CAP_MAX`.
- `src/Config.sol:117` — `GLOBAL_BORROW_CAP_MAX = 250_000e6`, declared `constant`. Changing it means redeploying.
- `src/LenderPool.sol:553-557` / `:1188-1192` — `_totalAssets` and `depositCapUsage` bound capital by that cap.
- `src/EpochHarvester.sol:646-647` and `:665-666` — both delivery helpers offer `pendingLenderYield` whole, with no clamp.

**The design's own escape hatch assumes something that is not true here.** The guard's rationale is that refusing is safe because the share *"stays in `pendingLenderYield` until there is capital to pay it to."* That presumes capital can eventually arrive. Above the ceiling it cannot. The existing regression test `test_distributeYield_refusesAnEpochLargerThanThePool` (`test/LenderPool.t.sol:1086-1104`) demonstrates recovery by depositing more — which works at its 5,000 USDC fixture and is impossible at 300,000.

**The machinery for partial delivery already exists on both sides, unused.** The harvester measures what actually arrived and does `pendingLenderYield -= delivered`, with a comment explaining why it decrements rather than assigns (`src/EpochHarvester.sol:660` — *"Decrement, do not assign."*). The pool's own empty-pool branch already accepts a **partial** amount, pulling just the claim shortfall and leaving the rest pending (`src/LenderPool.sol:1781-1789`). What is missing is a clamp on the amount offered.

**No other drain exists.** `_push` swallows the revert, so nothing moves and no state changes. `pendingLenderYield` is explicitly excluded from the harvester's sweep — *"the one balance already spoken for"* (`src/EpochHarvester.sol:437`). There is no owner rescue.

### Exploit path

No attacker is required — this is what happens if the backlog simply grows large before the pool's first delivery.

1. `pendingLenderYield` accrues every epoch as the lender share of farm yield, whether or not a pool is live. `README.md` marks the pool "not approved for activation", so there is a pre-activation window during which it accumulates with no drain.
2. Bond collateral is **not** bounded by the 250,000 borrow cap, so the yield accruing on it is not bounded by that cap either.
3. Once `pendingLenderYield > 250,000e6`, every delivery attempt reverts `YieldExceedsCapital`.
4. Capital cannot be raised to meet it: at full funding `maxDeposit` is `0`, and the cap cannot legally be set higher.
5. The state is terminal. No permissionless call, no owner call, and no passage of time changes it.

## Impact

- **Lenders** permanently forfeit the entire accrued lender-yield backlog. It is not lost in transit and nobody else receives it — it is frozen in the harvester with no path out and no owner lever.
- **The trigger needs no privileged role and no adversary.** It is reached by ordinary accrual before first delivery.
- Not a solvency event: no principal is at risk and no attacker profits.

**Economic Impact:** PoC — hard capital ceiling **250,000.000000 USDC** with `maxDeposit` at **0**; a **300,000.000000 USDC** backlog reverts `YieldExceedsCapital` permanently. The same pool accepts a clamped **250,000.000000** offer and streams it in full. Amount at risk is the whole backlog above the ceiling, unbounded above.

### Severity — real-world impact × likelihood

- **Impact: High.** Permanently frozen unclaimed yield with no owner lever and no alternate drain — the bug-bounty severity tables list permanent freezing of unclaimed yield as High on its own.
- **Likelihood: Low.** Requires the backlog to clear a 250,000 USDC ceiling before the pool's first delivery. Stated as an assumption: book size and farm APR were not modelled, so the accrual side reaching that figure is unproven. What is proven is that the state is terminal once reached.
- **High × Low → Medium.**

> Note on the accrual assumption: `KNOWN_RISKS.md` already records `pendingLenderYield` on the live pool reading `124.885415` USDC *"parked with nobody to deliver it to"*. The parking behaviour is real today; only the magnitude required to make it terminal is untested.

## Proof of Concept

Paste into `LenderPoolTest` (`test/LenderPool.t.sol`) and run it. It uses only that fixture's own `pool` / `usdc` / `admin` / `alice` / `bob` / `harvester` members — nothing else is needed.

```
uint256 internal constant CEILING = 250_000e6; // Config.GLOBAL_BORROW_CAP_MAX
uint256 internal constant BACKLOG = 300_000e6;

function _fillPoolToTheCeiling() internal {
    vm.prank(admin);
    pool.setDepositCap(Config.GLOBAL_BORROW_CAP_MAX);
    usdc.mint(alice, CEILING);
    vm.prank(alice);
    pool.deposit(CEILING, alice);
}

/// 1. The ceiling is hard: the cap cannot exceed GLOBAL_BORROW_CAP_MAX, and once the pool is
///    full there is no room left for more capital to arrive.
function test_F11_theCeilingIsHard() public {
    vm.prank(admin);
    vm.expectRevert(
        abi.encodeWithSelector(
            LenderPool.DepositCapTooLarge.selector, Config.GLOBAL_BORROW_CAP_MAX + 1, Config.GLOBAL_BORROW_CAP_MAX
        )
    );
    pool.setDepositCap(Config.GLOBAL_BORROW_CAP_MAX + 1);

    _fillPoolToTheCeiling();
    emit log_named_uint("MEASURED max capital  ", pool.totalAssets());
    emit log_named_uint("MEASURED maxDeposit   ", pool.maxDeposit(bob));
    assertEq(pool.maxDeposit(bob), 0, "more capital can still enter");
}

/// 2. A backlog above that ceiling is refused, and no action can raise capital to meet it.
function test_F11_theBacklogIsPermanentlyStuck() public {
    _fillPoolToTheCeiling();
    uint256 capital = pool.totalAssets();

    usdc.mint(harvester, BACKLOG);
    vm.prank(harvester);
    usdc.approve(address(pool), BACKLOG);

    emit log_named_uint("MEASURED backlog owed ", BACKLOG);
    emit log_named_uint("MEASURED capital      ", capital);

    vm.prank(harvester);
    vm.expectRevert(abi.encodeWithSelector(LenderPool.YieldExceedsCapital.selector, BACKLOG, capital));
    pool.distributeYield(BACKLOG);
```

```
}

/// 3. The money is fine; only the offer size is wrong. The same pool, in the same state,
///    accepts a clamped offer and streams it in full. The fix is a clamp, not a redesign.
function test_F11_aClampedOfferIsAcceptedInFull() public {
    _fillPoolToTheCeiling();
    uint256 capital = pool.totalAssets();

    usdc.mint(harvester, BACKLOG);
    vm.prank(harvester);
    usdc.approve(address(pool), BACKLOG);

    vm.prank(harvester);
    pool.distributeYield(capital); // clamped to exactly what fits

    emit log_named_uint("MEASURED clamped offer accepted", capital);
    assertEq(pool.unreleasedYield(), capital, "the clamped epoch did not stream");
}
```

### Run output

```
$ forge test --match-test test_F11 -vv
[PASS] test_F11_aClampedOfferIsAcceptedInFull() (gas: 548442)
Logs:
  MEASURED clamped offer accepted: 250000000000

[PASS] test_F11_theBacklogIsPermanentlyStuck() (gas: 451448)
Logs:
  MEASURED backlog owed : 300000000000
  MEASURED capital      : 250000000000

[PASS] test_F11_theCeilingIsHard() (gas: 368426)
Logs:
  MEASURED max capital  : 250000000000
  MEASURED maxDeposit   : 0

Suite result: ok. 3 passed; 0 failed; 0 skipped
```

Test 3 is the one that matters: the money is acceptable, only the offer size is wrong. The fix is a clamp, not a redesign.

## Recommendation

Clamp the offer. This does not weaken the guard — its stated purpose is that *"an epoch that is larger than everything the pool holds is not an epoch this pool earned"*, and delivering exactly `capital` honours that. Each delivery raises capital, so the backlog drains over successive flushes instead of never.

1. **Clamp pool-side**, at `src/LenderPool.sol:1817`:

```
// was: if (liquidityDeficitBefore == 0 && streamable > capital) revert YieldExceedsCapital(...);
if (liquidityDeficitBefore == 0 && streamable > capital) {
    streamable = capital;        // take what fits; the rest stays pending
    amount = covered + streamable;
}
```

1. **Or clamp harvester-side**, which is arguably cleaner since the harvester already measures delivery: expose the pool's acceptable maximum as a view and offer `min(pendingLenderYield, that)`.

Either way, add a regression test at a backlog above `GLOBAL_BORROW_CAP_MAX` — the existing one recovers by depositing more, which cannot scale past the ceiling.

### Status

**Fixed** — At the hard deposit-cap ceiling, `distributeYield` clamps delivery to available capital and leaves the remainder pending for later flushes. Oversized delivery below the terminal ceiling remains refused, preserving the anti-capture invariant.

---

## M-06 A request serviced down to one share-wei keeps the rest of its cash floor (maintainer-reported)

**Source:** GitHub issue #64
**Source finding ID:** `Issue #64`
**Report severity:** Medium

### Description

`serviceWithdrawalRequest` releases a request's cash floor in full only when the final share is serviced. If a request's floor exceeds the post-impairment value of its remaining shares, servicing almost all shares can pay the controller everything a complete service would have paid while leaving the excess floor reserved against one share-wei. That reservation reduces every other request cap, synchronous withdrawal capacity, borrowing availability, and `available()`, yet only the controller or its request operator can release it.

The condition does not require a realised loss. A request filed while the book is idle can become exposed after later lending, and the temporary whole-debt impairment mark during a routine liquidation can create the mismatch even when the debt is subsequently repaid in full. An asset-denominated integration can leave the dust accidentally by round-tripping `previewRedeem` through `previewWithdraw`; one full `maxRequestRedeem` call still completes safely.

### Impact

Third-party lenders can have otherwise available cash locked behind withdrawal doors that read zero, with no owner lever and no time-based release. New deposits can merely move the lock onto the depositor. The controller cannot extract more than the pool holds and gains no profit; the amount locked is bounded by the controller's floor minus the value of its remaining shares.

The finding is Medium because the trigger can arise during routine liquidation marking without any realised loss, and ordinary asset-denominated integration logic can activate it accidentally. It remains below High because no principal is stolen, there is no leveraged profit, a stationary book does not produce the mismatch, and correct full-share servicing completes.

### Proof of Concept

The maintainer added three current-source reproduction suites through PRs #65 and #66:

- `test/R63A3_DrawMemoryDust.t.sol`: five bare-pool tests, including 3,333.333333 USDC paid either way, 1,666.666667 USDC retained on one share-wei, and the other lender's doors and `available()` at zero.
- `test/R63S61_DustHeldFloorWiredGraph.t.sol`: four wired-graph tests covering the same retained floor through protocol paths.
- `test/R64A4_DustFloorCurve.t.sol`: thirteen tests covering the threshold curve, a temporary impairment mark that is later fully released, and asset-denominated integration rounding.

33Labs locally executed all three suites on remediation snapshot `f6893cb`: **22 passed, 0 failed, 0 skipped**. The separately described no-loss mark route on the complete four-contract graph was not yet committed as a repository test at the report date.

## Recommendation

Ensure any partial service cannot leave a reserved floor materially above the value of the request's remaining shares. One direct design is to cap the remaining floor after service at the assets represented by the remaining shares, while preserving an explicit minimum only if justified.

Treat this as a joint design decision with issue #61: writing floors down during partial service also softens the accepted post-raw-loss floor guarantee and changes existing regression figures. Add regression coverage for full-share and asset-denominated integrations, the temporary impairment-mark route, multiple requesters, cancellation, and post-loss behavior before merging a fix.

## Status

**Acknowledged / Accepted Risk** — The withdrawal-floor lock remains reproducible on remediation snapshot `f6893cb`: 33Labs locally executed the three repository suites covering the bare-pool, wired-graph, raw-loss, temporary-mark, integration-rounding, and threshold-curve variants, with 22 tests passing and none failing. The finding is Medium because routine liquidation marking and ordinary asset-denominated integrations can trigger an indefinite third-party withdrawal lock. The protocol team acknowledges this behavior and accepts the residual risk without a code change.

---

# Low Findings

## L-01 Permissionless settle discards a borrower's sub-unit yield accrual

**Source:** GitHub issue #54
**Source finding ID:** `L-02`
**Report severity:** Low

### Description

#### What's wrong

Settling a borrower's yield stamps their bookmark forward *before* checking whether there was anything to pay. If the amount rounds down to zero, the bookmark still moves and that fraction is gone for good. Anyone can trigger a settle for anyone else, so a small borrower can be settled often enough that every accrual rounds to zero and their yield never arrives. Since that yield is what repays their loan, suppressing it keeps them in debt.

#### How (mechanism)

Affected code:

- `src/CreditManager.sol:1338` — `settle(address borrower)` is `public whileAttached`, callable by anyone for any borrower.
- `src/CreditManager.sol:2855-2858` — the index write precedes the early return:

```
uint256 acc = accYieldPerBond;
uint256 owed = _pending(borrower, bonds);
yieldIndexOf[borrower] = acc;   // written unconditionally
if (owed == 0) return;          // fractional accrual is now unrecoverable
```

- `src/CreditManager.sol:2954` — `_pending` derives `delta = _projectedAcc() - yieldIndexOf[borrower]` and floors the product, so whenever `bonds * delta / 1e18 == 0` the accrual is zero while the index still moves to `acc`.

### Why this is Low

The attack is uneconomic by a wide margin. Each call destroys strictly less than one USDC base unit (`< $0.000001`) and costs a full transaction in gas. Denying a position roughly `$50` of annual yield requires on the order of millions of calls — thousands of dollars of gas to destroy tens of dollars of value, with no profit accruing to the attacker. It is griefing with cost far exceeding damage, and Cantina AI-6 caps dust and rounding impact at Low.

### Exploit path

1. A borrower holds a small bond position.
2. An unrelated caller invokes `settle(borrower)` at an interval short enough that `bonds * delta / 1e18` floors to zero.
3. `yieldIndexOf[borrower]` advances to the current accumulator; `owed` is zero; the fraction between the old and new index is gone.
4. Repeating this before the accrual can ever reach one base unit suppresses the borrower's yield indefinitely.

Because the borrower's yield is the mechanism that repays their loan, sustained suppression prevents the intended debt reduction and could leave a position liquidatable that uninterrupted accrual would have cured.

The unconditional stamp is deliberate for a different case — the comment at `:2845-2848` explains that stamping at `bonds == 0` is what stops a first-time depositor claiming the whole historical accumulator. That rationale covers the zero-**bonds** case; it does not cover a position with non-zero bonds whose accrual merely rounds down.

## Impact

- **Targeted small borrowers** can have yield accrual suppressed, blocking the self-repaying-loan mechanism and, at the margin, keeping a position liquidatable.
- No attacker profit; damage is bounded per call at one base unit and the aggregate cost to the attacker exceeds the aggregate loss to the victim under any realistic gas price.

**Economic Impact:** PoC — **41,254** base units credited by a single settle over a 1,800-second window against **39,600** when settled every second: **1,654 base units (≈ $0.001654) destroyed**. Per-call loss is bounded below one base unit; the gas to repeat it dwarfs the value destroyed.

### Severity — real-world impact × likelihood

- **Impact: Low.** Per call, strictly less than one USDC base unit is destroyed. Sustained suppression could keep a small position liquidatable that uninterrupted accrual would have cured.
- **Likelihood: Low.** Uneconomic by a wide margin — denying roughly $50 of annual yield needs on the order of millions of transactions, thousands of dollars of gas to destroy tens of dollars of value, with zero profit to the attacker. Cantina AI-6 caps dust and rounding impact at Low.
- **Low × Low → Low.**

## Proof of Concept

Paste into `ImpairmentIntegrationTest` (`test/Impairment.integration.t.sol`) and run it. Every helper it calls is already defined in that fixture — nothing else is needed.

```
/// @notice **[L-02]** `_settle` stamps `yieldIndexOf` before its `owed == 0` return, so every
///         settle discards the sub-unit remainder of the accrual. Against a small holder beside
///         a large one the remainder is non-zero, and a stranger settling often enough credits
///         the victim strictly less than a single settle over the same window.
/// @dev The per-call loss is bounded below one USDC base unit, which is why this is Low: the
///      gas to repeat it dwarfs the value destroyed. The defect is real, the economics are not.
function test_POC_permissionlessSettleErasesSubUnitYield() public {
    // A small holder beside alice's 100 bonds: `bonds * delta / 1e18` no longer divides evenly.
    address victim = makeAddr("victim");
    bond.mint(victim, 10);
    vm.startPrank(victim);
    bond.setApprovalForAll(address(vault), true);
    vault.depositBonds(1);
    vm.stopPrank();

    uint256 pot = 1_000e6;
    usdc.mint(harvester, pot);
    vm.startPrank(harvester);
    usdc.approve(address(credit), pot);
    credit.receiveYield(pot);
    credit.distributeYield(pot);
    vm.stopPrank();

    credit.settle(victim); // level the starting index for both arms
    uint256 claimBefore = credit.claimableOf(victim);
```

```
    uint256 window = 1_800;

    uint256 snap = vm.snapshotState();

    skip(window);
    credit.settle(victim);
    uint256 control = credit.claimableOf(victim) - claimBefore;

    vm.revertToState(snap);

    for (uint256 i = 0; i < window; i++) {
        skip(1);
        vm.prank(stranger);
        credit.settle(victim);
    }
    uint256 griefed = credit.claimableOf(victim) - claimBefore;

    emit log_named_uint("MEASURED credited by a single settle  ", control);
    emit log_named_uint("MEASURED credited when settled each sec", griefed);
    emit log_named_uint("MEASURED destroyed (USDC base units)   ", control - griefed);
    assertLt(griefed, control, "repeated settling destroyed no accrual");
}
```

### Run output

```
$ forge test --match-test test_POC_permissionlessSettleErasesSubUnitYield -vv
[PASS] test_POC_permissionlessSettleErasesSubUnitYield() (gas: 145457902)
Logs:
  MEASURED credited by a single settle  : 41254
  MEASURED credited when settled each sec: 39600
  MEASURED destroyed (USDC base units)   : 1654
Suite result: ok. 1 passed; 0 failed; 0 skipped
```

## Recommendation

Advance the index only when it cannot destroy an entitlement — that is, when the position holds no bonds (the case the existing rationale protects) or when a non-zero amount is actually paid:

```
uint256 acc = accYieldPerBond;
uint256 owed = _pending(borrower, bonds);
if (bonds == 0 || owed != 0) {
    yieldIndexOf[borrower] = acc;   // stamp for the first-deposit case, or on real payment
}
if (owed == 0) return;
```

A position with non-zero bonds then keeps its residual index until the accrual reaches one base unit, and the first-time-depositor defence at `:2845-2848` is preserved unchanged. Alternatively, carry the floored remainder forward in a per-borrower dust accumulator rather than discarding it.

## Status

**Fixed** — Non-count-changing settlement paths advance the index only by the amount actually paid, preserving sub-unit accrual until it becomes payable. Count-changing paths retain the full checkpoint behavior needed to prevent re-pricing.

---

# L-02 After a raw cash loss, the request floors can lock the whole remaining cash until a repayment or a cancel

**Source:** GitHub issue #61
**Source finding ID:** `L-03`
**Report severity:** Low

## Description

Since `68c0c26` a queued lender is owed a cash floor, quoted at request time out of the cash not already owed to earlier requests and held in the reserve. The floors are not written down when a raw loss of executable cash lands, so after a loss their sum can exceed the cash. Each request is then capped at the executable cash the OTHER live requests are not owed, and nobody is paid beyond what exists. The cash no request can reach is locked, and with several floors that can be the whole remaining cash rather than the amount lost.

### Recovery Conditions

Two doors, neither privileged and neither timed.

1. A floor holder cancels (`cancelWithdrawalRequest`): exactly that holder's floor is released from the reserve, so the other requests' caps rise by it. What the next holder can then **draw** is still limited by what its escrowed shares are worth at the post-loss price: in the three-floor case a cancel of a 100.000000 floor lets the next holder take **66.666666** in one call, leaving 66.666667 for the third.
2. A borrower repays (`repayPrincipal`) enough to lift the executable cash above the other floors for some request, after which that request's live slice pays: measured, a repayment of 30.000000 reopens three 50.000000-floor doors to 29.999999 each.

The repair doors do not reach it: `claimSolvencyDeficit()` and `entryPriceDeficit()` read 0 and `coverClaimDeficit(1)` / `coverEntryPriceDeficit(1)` revert, because the locked cash is recognised shareholder cash promised twice rather than a deficit either door is built for, and the owner has no lever over a floor. **There is no guaranteed recovery time**: the lock lasts until a borrower repays or a holder cancels.

## Impact

A loss of executable cash can make every queued request temporarily unserviceable and can lock the whole remaining cash rather than only the amount lost. Unqueued exits and borrowing can also read zero while the lock persists.

### Preconditions

A raw loss of executable cash with more than one request queued. The tests simulate that loss by moving cash out of the pool; they do not demonstrate an unprivileged caller causing it.

## Proof of Concept

in `test/R60S2_H03LockBound.t.sol` (added by #60): the reviewers' two rows verbatim, three queued holders of 100.000000 USDC each with 100.000000 lost leave **200.000000 locked**, one hundred holders leave **9,900.000000 locked**; in both, `unreservedIdle()`, `available()`, every `maxRequestRedeem` and both repair deficits read 0. The over-reservation (the floors' sum less the cash, 100.000000 in both cases) measures the **shortfall, not the amount locked**. A fuzz over 2 to 40 equal floors and any loss inside the book states the cap identity and the total-lock equivalence; there is no closed form for the partially drained regime, because a drawn request converts its cap through the gross conversion and is paid through `previewRedeem`.

The two-floor shape the floor's own test pins (floors of 2,000.000000 and 1,000.000000 with 2,500.000000 lost of 5,000.000000) locks 500.000002.

### Recommendation

Retain prominent disclosure of the lock, preserve the multiple-request regression and fuzz coverage, and operationally monitor raw cash losses and queued request floors. Reconsider proportional floor write-down or explicit loss-allocation semantics before accepting third-party capital if indefinite recovery remains unacceptable.

#### Design Trade-off

A reservation that is an amount has to cost something when the cash it was quoted from is destroyed. The alternative measured on a copy of this source, writing the floors down on a raw loss (+285 runtime bytes on `LenderPool`), takes the lock to 0.000001 but makes every queued lender permanently lose a proportional part of her quote while any floor lives, and a cancel then restores 0 to the other request rather than the floor; the aggregate reconstructions measured earlier (+532 and +533 bytes) over-promise after a loss. The lock is disclosed in `KNOWN_RISKS.md` under material residual risks and pinned by the tests above.

### Status

**Acknowledged / Accepted Risk** — A raw cash loss can leave multiple request floors collectively above executable cash, locking the remaining cash until a borrower repayment or a floor-holder cancellation. The behavior is documented, regression-tested, and intentionally retained; neither recovery condition has a guaranteed time.

---

## L-03 A paused or blacklisting USDC shuts every bond door, because the farm pays its pending USDC inside the same call (maintainer-reported)

**Source:** GitHub issue #68
**Source finding ID:** `Issue #68`
**Report severity:** Low

### Description

The adapter's own USDC sweep during `unstake` is best-effort, but the live DexFi farm settles pending USDC inside its `deposit` and `withdraw` calls. Those reward transfers are not wrapped by the adapter. Once any reward is pending, a Circle USDC pause or blacklist of the adapter therefore reverts the farm call itself and closes every bond-movement door that depends on it.

Measured affected paths include `withdrawBonds`, `depositBonds`, and `harvestYield`; source review identifies the same dependency in `seize`, `disposeTo`, `restakeLoose`, and mint paths. The owner-only `emergencyUnstake` forfeits rewards and moves every bond to one address, leaving custody insolvent until bonds are returned, so it is an emergency runbook rather than per-position liveness.

### Impact

Borrowers cannot add bonds to cure a falling NAV and debt-free holders cannot withdraw while the external token restriction persists. Liquidation and workout clocks continue running during the outage. No USDC or debt is stolen, and Circle has never paused USDC.

The finding is Low because the trigger is an external Circle pause or blacklist that no protocol participant can induce. The impact under that condition is nevertheless material: exit and cure paths fail together, and the only available response is a protocol-wide owner action that temporarily breaks custody solvency.

### Proof of Concept

`test/R64A1_RealFarmPausedUsdcFork.t.sol`, merged through PR #66, runs against the real Base USDC and DexFi farm. The maintainer measured 4 passed, 0 failed, and 1 skipped at block 51,619,708. An independent reviewer reran it at block 51,641,633 and reproduced all door failures: paused `withdrawBonds(5)`, `depositBonds(5)`, and `harvestYield` reverted `Pausable: paused`; after emergency unstake they passed with no pending rewards, then `withdrawBonds` reverted again one second later with 3 wei pending. The blacklisted-adapter variant reverted `Blacklistable: account is blacklisted`.

The fork suite is opt-in and is not executed by repository CI. The sibling source-reviewed paths were not claimed as fork-executed.

### Recommendation

Maintain and publish an operational pause/blacklist runbook covering emergency unstake, reward forfeiture, custody reconciliation, bond return, same-block restaking or exits where possible, and borrower protection while liquidation clocks continue. Model reward payment by token transfer in local mocks so CI can cover the dependency.

A complete mechanism-level mitigation requires a farm or custody path that can move bonds without forcing reward transfer, or protocol logic that pauses cure-sensitive liquidation clocks during an external settlement-token outage. Adapter-side `try/catch` around its own sweep does not isolate a revert inside the farm's bond-movement call.

### Status

**Acknowledged / Accepted Risk** — The independent reviewer reproduced the paused-USDC and blacklisted-adapter variants against the live Base contracts. Commit `f6893cb` corrects `AUDITS.md` and records the custody and running-clock residuals in `KNOWN_RISKS.md`; it does not change the mechanism. The risk is classified Low because it depends on an external Circle pause or blacklist that protocol actors cannot induce.

---

## Conclusion

The review contains **13 findings**: **4 High**, **6 Medium**, and **3 Low**. Remediation review classified **10 findings as Fixed** and **3 as Acknowledged / Accepted Risk**.

M-06 is an accepted risk: a request can be serviced to share dust while retaining a materially oversized cash floor, indefinitely blocking other lenders without a victim-side or owner-side release. L-02 is the accepted post-raw-loss withdrawal-floor lock, and L-03 documents the accepted external paused/blacklisted-USDC custody dependency. Before accepting third-party capital, 33Labs recommends revisiting M-06 together with the L-02 design trade-off, publishing the L-03 incident runbook, retaining the activation gate, and verifying deployment/source parity.