Security Review

Cantina Managed

3F: GRUNT

Cantina Managed review by:
M4rio.eth, Lead Security Researcher
Chinmay Farkya, Security Researcher

CANTINA

September 16, 2026

Contents

1 Introduction **2**
1.1 About Cantina . . . 2
1.2 Disclaimer . . . 2
1.3 Risk assessment . . . 2
1.3.1 Severity Classification . . . 2

2 Security Review Summary **3**
2.1 Scope . . . 3

3 Findings **4**
3.1 Medium Risk . . . 4
3.1.1 Liquidation can produce a positive performance-fee basis while PositionManager NAV falls . . . 4
3.2 Low Risk . . . 4
3.2.1 Centrifuge `forceEnd` lets `resolve()` clear a Retargetter operation while the venue request is still pending . . . 4
3.2.2 `Request.mint()` honors a stored mint authorization after the PositionManager principal cap falls . . . 6
3.2.3 A dust basis clears the whole held management-fee deduction . . . 6
3.2.4 Empty-good-debt exit skips held-fee scaling and leaves an oversized deduction . . . 7
3.2.5 Rebalance re-anchors the performance reference at the post-loss NAV, so recovery is charged as performance . . . 8
3.2.6 Offer fill rounding can push LTV past `liquidationLtv` and unlock near-total equity liquidation . . . 9
3.2.7 Partitioning a liquidation target skips the best offer and fills from a worse one . . . 10
3.2.8 Maker callback in `Request.consume()` can invalidate the `maxPrincipal` cap checked by `Retargetter.consume()` . . . 11
3.2.9 `Retargetter.maxPrincipal` can exceed the one-trip repayment headroom of a fully deployed LTV-up operation . . . 12
3.2.10 A supply-only Retargetter rebalance lets holders of existing `PositionManager` shares withdraw Request-funded collateral . . . 13
3.2.11 Held management-fee credit is scaled by a deposit made at zero NAV . . . 14
3.2.12 Splitting one gain across many checkpoints can erase the performance fee . . . 15
3.2.13 Performance reference is rebased after outgoing token transfers, so a token callback can move the basis . . . 17
3.2.14 Retargetter binds to an unauthenticated `PositionManager` address . . . 18
3.2.15 Shared Midas redemption capacity can be exhausted after the Fund pays its bond 19
3.2.16 `FACILITATOR` can burn an intent's entire Midas share balance by overstating the redeem input . . . 20
3.2.17 Permissioned mToken deposits can become stuck when vault greenlisting is disabled 21
3.3 Informational . . . 22
3.3.1 Small hardening and documentation nits . . . 22
3.3.2 A revoked mint authorization permanently starts the Retargetter loan clock . . . 23
3.3.3 One-wei repayment flips a module into the management-fee basis for the whole checkpoint interval . . . 24
3.3.4 Fund adapters assume exact-transfer assets without documenting it . . . 25
3.3.5 Facility balance snapshots credit `PositionManager` fee-mint shares to the resolving intent . . . 25
3.3.6 Zero-share fee accrual is documented only in internal comments, not on the `feeData` interface . . . 26
3.3.7 Held management fee accumulator tracks charged assets, not minted shares, and this is undocumented . . . 27
3.3.8 Retargetter's flash-loan callback binds neither the payload nor the delivered principal 27
3.3.9 Standing `BorrowOffers` have no cap on liquidation bonus, seized equity, or close factor . . . 28
3.3.10 Offer-walk `Stop` / `Skip` semantics and raw-unit rounding are not documented on the external liquidation surface . . . 29
3.3.11 Minimum tick is payable out of `PositionManager` while committed Request principal stays unused . . . 29

3.3.12 `remediationDelta` counts expected collateral yield twice 30
3.3.13 Rebalance cooldown is enforced per call, so a `SYNC` payload with two `PositionManager` rebalances always reverts 31
3.3.14 Rebalance post-check reverts on modules the payload never touched 31
3.3.15 `pullRequestFunds(0)` is an undocumented close of the Retargetter funding round 32
3.3.16 Retargetter's one-operation-at-a-time guard is per instance, so two Retargetters can each admit the full principal cap 33
3.3.17 Splitting an exit across many transactions rounds `heldManagementFeeAssets` down repeatedly 33
3.3.18 Partial fills can leave all offer slots live but unusable 34
3.3.19 `setConfig` reprices the repayment of an already-started Retargetter operation 34
3.3.20 `CONSUMER_ROLE` can authorize itself to mint YT up to the yield cap 35
3.3.21 Offer proposers cannot cancel their own timelocked offers 36
3.3.22 `PositionManager` exposes no getter for the bad-debt exclusion flag 36
3.3.23 Retargetter ignores the Fund state returned by `cancelOrder` 37
3.3.24 Timelock changes do not retime existing offers 38
3.3.25 `MidasFund` accepts arbitrarily old oracle rounds 38

1 Introduction

1.1 About Cantina

Cantina is a security platform that pairs a world-class network of security researchers with agentic AI to help teams building onchain find and fix real risk. Through managed reviews, competitions, and bounties, Cantina secures protocols before vulnerabilities can be exploited.
Learn more at cantina.security

1.2 Disclaimer

Cantina Managed provides a detailed evaluation of the security posture of the code at a particular moment based on the information available at the time of the review. While Cantina Managed endeavors to identify and disclose all potential security issues, it cannot guarantee that every vulnerability will be detected or that the code will be entirely secure against all possible attacks. The assessment is conducted based on the specific commit and version of the code provided. Any subsequent modifications to the code may introduce new vulnerabilities that were absent during the initial review. Therefore, any changes made to the code require a new security review to ensure that the code remains secure. Please be advised that the Cantina Managed security review is not a replacement for continuous security measures such as penetration testing, vulnerability scanning, and regular code reviews.

1.3 Risk assessment

Severity level	Impact: High	Impact: Medium	Impact: Low
Likelihood: high	Critical	High	Medium
Likelihood: medium	High	Medium	Low
Likelihood: low	Medium	Low	Low

1.3.1 Severity Classification

The severity of security issues found during the security review is categorized based on the above table. Critical findings have a high likelihood of being exploited and must be addressed immediately. High findings are almost certain to occur, easy to perform, or not easy but highly incentivized thus must be fixed as soon as possible.

Medium findings are conditionally possible or incentivized but are still relatively likely to occur and should be addressed. Low findings are a rare combination of circumstances to exploit, or offer little to no incentive to exploit but are recommended to be addressed.

Lastly, some findings might represent objective improvements that should be addressed but do not impact the project's overall security (Gas and Informational findings).

2 Security Review Summary

From Jul 20th to Jul 31st the Cantina team conducted a review of grunt on commit hash 9ae1b3cf. The team identified a total of **43** issues:

Issues Found

Severity	Count	Fixed	Acknowledged
Critical Risk	0	0	0
High Risk	0	0	0
Medium Risk	1	1	0
Low Risk	17	13	4
Gas Optimizations	0	0	0
Informational	25	21	4
Total	**43**	**35**	**8**

The Cantina Managed team reviewed 3F's grunt holistically on commit hash c30b0f1f and concluded that all findings were addressed and no new vulnerabilities were identified.

2.1 Scope

The security review had the following components in scope for grunt on commit hash 9ae1b3cf:

```
src
├── borrow
│   ├── BorrowOffersRegistry.sol
│   ├── MorphoBorrowPosition.sol
│   └── MorphoBorrowPositionFactory.sol
├── facility
│   ├── Facility.sol
│   ├── IntentDescriptor.sol
│   └── base
│       ├── FacilityFunds.sol
│       ├── FacilityIntents.sol
│       ├── FacilityLP.sol
│       ├── FacilityPositionManager.sol
│       ├── FacilityRequests.sol
│       ├── FacilityRoles.sol
│       └── FacilitySwap.sol
├── funds
│   ├── centrifuge
│   │   ├── CentrifugeFund.sol
│   │   └── CentrifugeFundFactory.sol
│   ├── midas
│   │   ├── MidasFund.sol
│   │   └── MidasFundFactory.sol
│   ├── pareto
│   │   ├── ParetoFund.sol
│   │   └── ParetoFundFactory.sol
│   └── USCC
│       ├── SuperstateRestrictedWrappedAsset.sol
│       ├── USCCFund.sol
│       └── USCCFundFactory.sol
├── guard
│   ├── TransferGuard.sol
│   └── TransferGuardFactory.sol
├── libs
│   ├── Constants.sol
```

```
│   ├── borrow
│   │   ├── LibBorrowErrors.sol
│   │   ├── LibBorrowOffers.sol
│   │   ├── LibBorrowOffersConstants.sol
│   │   ├── MorphoBalancesLib.sol
│   │   └── SharesMathLib.sol
│   ├── common
│   │   ├── LibChecks.sol
│   │   ├── LibCommonErrors.sol
│   │   ├── LibPause.sol
│   │   └── LibTransientSlot.sol
│   ├── facility
│   │   ├── LibAddress.sol
│   │   ├── LibConstants.sol
│   │   ├── LibFacilityErrors.sol
│   │   ├── LibIntent.sol
│   │   ├── LibStorage.sol
│   │   └── LibTokenBalances.sol
│   ├── funds
│   │   ├── LibFundsErrors.sol
│   │   └── Order.sol
│   ├── manager
│   │   ├── LibConstants.sol
│   │   ├── LibExecutor.sol
│   │   ├── LibManagerErrors.sol
│   │   ├── LibOperations.sol
│   │   ├── LibStorage.sol
│   │   ├── LibView.sol
│   │   └── rebalancer
│   │       ├── LibRetargetterConstants.sol
│   │       ├── LibRetargetterErrors.sol
│   │       └── LibStorage.sol
│   └── request
│       ├── Lib128Fields.sol
│       ├── LibAllowance.sol
│       ├── LibMintAuth.sol
│       ├── LibRequestErrors.sol
│       └── LibTokenController.sol
├── manager
│   ├── PositionManager.sol
│   ├── PositionManagerFactory.sol
│   ├── base
│   │   ├── PositionManagerAdmin.sol
│   │   ├── PositionManagerBase.sol
│   │   ├── PositionManagerLP.sol
│   │   └── PositionManagerRebalancing.sol
│   └── rebalancer
│       ├── MorphoFlashLoanAdapter.sol
│       ├── MorphoRebalancer.sol
│       ├── Retargetter.sol
│       ├── RetargetterFactory.sol
│       └── RetargetterQuoter.sol
└── request
    ├── MorphoFlashLoanRequest.sol
    ├── MorphoFlashLoanRequestFactory.sol
    ├── Request.sol
    ├── RequestFactory.sol
    ├── Vault.sol
    ├── abstract
    │   ├── OfferReceiver.sol
    │   ├── tokens
```

```
│   │   ├── ControlledToken.sol
│   │   └── TokenController.sol
│   └── vault
│       ├── ControlledVault.sol
│       └── VaultController.sol
└── scripts
    ├── SyncAllocatorDeposit.sol
    ├── SyncDeposit.sol
    └── SyncWithdrawal.sol
```

3 Findings

3.1 Medium Risk

3.1.1 Liquidation can produce a positive performance-fee basis while PositionManager NAV falls

Severity: Medium Risk.

Context: MorphoBorrowPosition.sol#L255-L306, PositionManagerBase.sol#L65-L223.

A liquidation changes the borrow position directly. `MorphoBorrowPosition.preLiquidate` (`MorphoBorrowPosition.sol:344`) and native Morpho liquidation both settle on the Morpho market without touching PositionManager state, so the performance reference (`lastTotalAssets`, `lastDebt`) is never rebased in that call. `preLiquidate`'s proportional path is permissionless; the offer/band path additionally needs a standing offer authorized by a `PROPOSER` on `BorrowOffersRegistry`.

The next accrual computes the basis from the stale reference at `PositionManagerBase.sol:188-197`. A liquidation cuts collateral and debt in a ratio that leaves the surviving position below the reference LTV, so `scaledLastDebt > currentDebt` holds and the basis reads positive even though NAV dropped.

```
// src/manager/base/PositionManagerBase.sol:188-197
uint256 lastCollat = _storage.lastTotalAssets + _lastDebt;
// basis = mulDiv(lastDebt, currentCollat, lastCollat) - currentDebt
//       = LTV_ref * currentCollat - currentDebt
// Round up on the minuend (mulDivUp) so the basis is biased larger — favors the
// protocol, consistent with conservative-to-protocol rounding elsewhere.
uint256 scaledLastDebt = _lastDebt.mulDivUp(currentCollat, lastCollat);
if (scaledLastDebt > currentDebt) {
  basis = scaledLastDebt - currentDebt;
  advanceReference = true;
}
```

The performance fee minted on that basis is on top of the ordinary liquidation loss. The minted amount depends on the fee configuration and the management-fee deduction.

Example: USDC debt asset and wUSCC collateral, both 6 decimals, so collateral and debt are directly comparable. Assume a 10% (1000 BPS) performance fee.

1. Reference: 1,250,000 wUSCC collateral (`1.25e12`), 1,000,000 USDC debt (`1e12`). Reference LTV 80%, NAV 250,000 USDC (`2.5e11`).
2. A liquidation repays 500,000 USDC and seizes 525,000 wUSCC. Post-state: 725,000 wUSCC (`7.25e11`), 500,000 USDC debt (`5e11`).
3. NAV falls to 225,000 USDC, a drop of 25,000 USDC. The stale reference gives `scaledLastDebt = 1e12 * 7.25e11 / 1.25e12 = 5.8e11` (580,000 USDC), so `basis = 5.8e11 - 5e11 = 8e10`, i.e. 80,000 USDC.
4. The next accrual mints performance-fee shares worth 8,000 USDC on that basis, and `advanceReference` re-anchors the reference at the post-liquidation state. NAV fell by 25,000 USDC while the fee recipient was paid 8,000 USDC.

Recommendation: Reconcile the performance reference with the borrow-position state after a liquidation before computing the basis, so a NAV decrease cannot yield a positive basis. If the current behavior is intended, document external debt relief (direct Morpho repayment and liquidation recovery with no NAV improvement) as fee-bearing.

3F: Fixed in PR 211: `_pendingFees` caps the levered basis at the zero-floored NAV gain since the reference and holds the reference when the cap binds, with `rebaseSnapshot` converting a seizure deficit into carry so the pre-loss mark survives capital flows (high-water behavior).

Cantina Managed: Fix verified.

3.2 Low Risk

3.2.1 Centrifuge `forceEnd` lets `resolve()` clear a Retargetter operation while the venue request is still pending

Severity: Low Risk.

Context: LibStorage.sol#L138-L147, LibStorage.sol#L264-L275, Retargetter.sol#L298-L313, Retargetter.sol#L354-L379, CentrifugeFund.sol#L358-L399, CentrifugeFund.sol#L445-L513.

`CentrifugeFund.forceEnd` sets `internalState = ENDED` after checking only that there is no claimable fill (`maxMint` / `maxWithdraw`), and it checks cancellation claimables only when the order is already `RECOVERING` (`CentrifugeFund.sol:379-402`). The still-open `pendingDepositRequest` at the vault is never checked, so an order with an outstanding venue request reads `ENDED` .

`LibStorage.checkNoPendingOrder` accepts `ENDED` or `EMPTY` and clears the order (`LibStorage.sol:144-149`), so `Retargetter.resolve` passes its no-pending-order gate and calls `clearOperation` (`Retargetter.sol:344-353`). The operation's link to the outstanding venue request is gone. `unlock()` later mints the fill and pays `order.receiver` (`CentrifugeFund.sol:257-265`), which by then belongs to whichever operation is reusing the Fund. `Centrifuge` aggregates by `(vault, controller)` and exposes request ID 0 (`_PENDING_REQUEST`), so the request ID cannot identify which operation the liability belongs to.

```
// src/funds/centrifuge/CentrifugeFund.sol:376-404
  address _vault = $.vault;

  // Revert if claimable fills exist (must be drained via unlock() first).
  if (order.mode == Mode.DEPOSIT) {
    if (ICentrifugeVault(_vault).maxMint(address(this)) > 0) {
      revert LibFundsErrors.PendingClaimableAssets();
    }
  } else {
    if (ICentrifugeVault(_vault).maxWithdraw(address(this)) > 0) {
      revert LibFundsErrors.PendingClaimableAssets();
    }
  }

  // Revert if recoverable cancel assets exist (must be drained via recover() first)
  if (_internalState == State.RECOVERING) {
    if (order.mode == Mode.DEPOSIT) {
      if (ICentrifugeVault(_vault).claimableCancelDepositRequest(_PENDING_REQUEST,
      ↪  address(this)) > 0) {
        revert LibFundsErrors.PendingClaimableAssets();
      }
    } else {
      if (ICentrifugeVault(_vault).claimableCancelRedeemRequest(_PENDING_REQUEST,
      ↪  address(this)) > 0) {
        revert LibFundsErrors.PendingClaimableAssets();
      }
    }
  }

  $.internalState = State.ENDED;

  emit OrderForceEnded(_orderId, msg.sender);
}
```

```
// src/libs/manager/rebalancer/LibStorage.sol:144-149
function checkNoPendingOrder(RetargetterOperation storage self, address fund)
↪  internal {
  if (!self.orderLive) return;
  State state_ = IFund(fund).state(order(self));
```

```
    if (state_ != State.ENDED && state_ != State.EMPTY) revert
    ↪  LibRetargetterErrors.OrderPending();
    clearOrder(self);
}
```

Centrifuge share/asset conversions and partial unlocks can leave a bounded raw-atom remainder, which stays with the order/controller whose claim was converted. That does not repair cross-order attribution.

Example: Assets are USDC (debt, 6 decimals) and wUSCC (collateral, 6 decimals), quoted 1:1 for the arithmetic.

1. Operation 1 sends 5,000 USDC (`5e9`) into a pending Centrifuge deposit request.
2. A Fund `OPERATOR` calls `forceEnd()` ; no fill is claimable yet, so the check passes and the order reads `ENDED` .
3. Repayment (or the 90-day `REPAYMENT_DEADLINE_OFFSET` expiry sync) lets `resolve()` run and clear Retargetter state.
4. Operation 2 starts and reuses the same Fund.
5. Centrifuge fulfills late. Operation 2 calls `unlock()` and receives Operation 1's wrapped-share output.

The unmatched 5,000 wUSCC (`5e9`) lands as collateral with no shares minted, so PositionManager NAV goes from 6,000 to 11,000 USDC against an unchanged supply of 6,000e18 shares. `burn()` pays `collateral = totalCollateral * shares / (totalSupply + virtualShareOffset)` with `virtualShareOffset = 1e12` (`PositionManager.sol:74-75`), so burning all 6,000e18 shares against 11,000 wUSCC (`1.1e10`) returns `1.1e10 * 6e21 / (6e21 + 1e12) = 10,999,999,998` base units, i.e. 10,999.999998 wUSCC, leaving 2 base units behind. That is 4,999.999998 wUSCC over the holder's own 6,000, and Operation 1's controller is short the full 5,000 USDC. The path needs Request terminality plus exit authority; repayment leaves lenders whole, and a direct loss requires default or under-repayment.

Recommendation: Preserve each controller's pending-claim ownership across `forceEnd()` and `clearOperation()` until late venue claims settle. Failing that, document in the operator runbook that old-controller ownership is preserved and Fund reuse is blocked until the venue request and all claimables are terminal.

3F: Acknowledged, no code change: after a Centrifuge `forceEnd` the Fund is not reused until the venue request and its claimables are terminal, enforced as an operator-runbook rule.

Cantina Managed: Acknowledged.

3.2.2 `Request.mint()` honors a stored mint authorization after the PositionManager principal cap falls

Severity: Low Risk.

Context: Retargetter.sol#L230-L248, LibStorage.sol#L149-L175, Request.sol#L328-L339.

`Retargetter.authorizeMinting()` checks the principal cap once, at authorization time: `Retargetter.sol:277` passes `maxPrincipal(operation_.positionManager)` into `LibStorage.checkOffer()` , which reverts with `PrincipalCapExceeded` when PT supply plus outstanding authorizations plus the new amount exceeds it (`LibStorage.sol:174`). `maxPrincipal()` is derived from live PositionManager collateral and debt (`Retargetter.sol:688-695`), so a public liquidation against the PositionManager reduces it afterwards.

`Request.mint()` reads the stored authorization and checks only the request state and the caller's slippage bounds. It transfers the authorized principal and mints PT/YT without reading the current

cap, so the broker can mint the old amount after the cap falls. A later Retargetter settlement can require the PositionManager to pay the minimum tick on that excess capital.

```
// src/request/Request.sol:378-389
function mint(uint128 maxPt, uint128 minYt) external nonReentrant {
  if (_syncWithdrawalStatus()) revert LibRequestErrors.AlreadyRepaid();
  (uint128 ptMintAuth, uint128 ytMintAuth) = msg.sender.mintAuth();
  // Early return when no authorization — prevents griefing where a zero-authorized
  ↪  caller
  // repeatedly calls mint to bump lastMintTimestamp and permanently delay
  ↪  setRepaid().
  if (ptMintAuth == 0 && ytMintAuth == 0) return;
  if (ptMintAuth > maxPt || ytMintAuth < minYt) revert
  ↪  LibRequestErrors.SlippageExceeded();
  msg.sender.updateMintAuth(0, 0);
  _asset().safeTransferFrom(msg.sender, address(this), ptMintAuth);
  _mint(msg.sender, ptMintAuth, ytMintAuth);
  _requestStorage().lastMintTimestamp = uint40(block.timestamp);
}
```

Sequence:

1. Authorize broker principal `P` while `P <= maxPrincipal`.
2. Execute an independent public liquidation that reduces `maxPrincipal` below `P`.
3. Call `Request.mint()` with the stored authorization.
4. The broker deposits `P` and receives PT/YT under the original commitment. On settlement it recovers its principal and earns the promised tick.

The client treats the stored authorization as a firm capital commitment, so honoring the grant after PositionManager state changes is the intended policy; an exercise-time cap check would be a different policy. The broker and liquidator act independently, and the liquidation bonus belongs to the standalone liquidation path. The durable authorization adds the committed Request liability and its tick. A loss occurs only if the PositionManager cannot back the commitment at settlement.

The amounts are integers: `Request.consume()` floors proportional YT once for the consumed nonce (`Request.sol:422`, `mulDiv`), while settlement rounds accrued YT up (`RetargetterQuoter.sol:205`, `ytSupply.fullMulDivUp(duration, horizon)`). The ceiling is a conservative debt-asset obligation for the firm commitment.

Recommendation: Define the authorization's backing policy explicitly and apply it consistently when `Request.mint()` exercises the stored commitment. If unreserved firm commitments are intentional, document the policy and its backing requirements.

3F: Acknowledged as intended firm-commitment policy and documented on `authorizeMinting` in PR 213; exercise-time cap re-checks were tightened separately under findings "Maker callback in `Request.consume()` can invalidate the `maxPrincipal` cap checked by `Retargetter.consume()`" and "`Retargetter.maxPrincipal` can exceed the one-trip repayment headroom of a fully deployed LTV-up operation".

Cantina Managed: Acknowledged.

3.2.3 A dust basis clears the whole held management-fee deduction

Severity: Low Risk.

Context: PositionManagerBase.sol#L65-223. A permissionless caller can create the dust basis through Morpho, but the caller receives no native payout. A later PositionManager flow and an aligned fee recipient are required for fee dilution.

While the performance reference is held, `heldManagementFeeAssets` should offset the next positive performance basis. `_pendingFees()` computes the basis as `scaledLastDebt - currentDebt` and

sets `advanceReference = true` on any positive value (`PositionManagerBase.sol:194-197`), then zeroes `heldManagementFees_` on the advance path (`PositionManagerBase.sol:229`).

The performance-fee branch still requires `basis > managementFeeAssets + heldManagementFees_` (`PositionManagerBase.sol:223`), so a basis too small to mint performance-fee shares clears the full accumulated deduction.

```
// src/manager/base/PositionManagerBase.sol:193-197
uint256 scaledLastDebt = _lastDebt.mulDivUp(currentCollat, lastCollat);
if (scaledLastDebt > currentDebt) {
  basis = scaledLastDebt - currentDebt;
  advanceReference = true;
}
```

```
// src/manager/base/PositionManagerBase.sol:223-229
if (fd.performanceFee > 0 && basis > managementFeeAssets + heldManagementFees_) {
  totalFeeAssets += (basis - managementFeeAssets -
  ↪  heldManagementFees_).mulDiv(fd.performanceFee, BPS);
}
// While the reference is held, the current interval's management fee joins the
↪  accumulator
// so the next crystallization deducts it; on advance the pending deduction is
↪  consumed (or,
// for any excess above the basis, forgiven) and the accumulator restarts.
heldManagementFees_ = advanceReference ? 0 : heldManagementFees_ +
↪  managementFeeAssets;
```

Example: Debt asset USDC (6 decimals), collateral wUSCC (6 decimals), 2% management fee, 20% performance fee, and a hold window long enough for the accumulator to reach 50,000 USDC.

1. `heldManagementFeeAssets` reaches 50,000 USDC (5e10) while a zero-interest Morpho position keeps the basis at zero.
2. An external caller repays 1 base unit, 0.000001 USDC, through `Morpho.repay(..., onBehalf)` . Collateral is unchanged, so the next basis is 1.
3. `basis > managementFeeAssets + heldManagementFees_` is `1 > 5e10 + current interval charge` , false, so no performance-fee shares mint. `advanceReference` is still true, so line 229 sets the accumulator to 0 and the 50,000 USDC deduction is written off.
4. A later gain with a basis of 500,000 USDC (5e11) is then charged on the full amount instead of 450,000, so the fee recipient mints 100,000 USDC of fee shares instead of 90,000. The extra 10,000 USDC (50,000 * 20%) dilutes LPs. The unrelated caller receives no direct payout.

Scope note: this is the one-atom positive-basis transition where a floor mints zero fee shares while still advancing the reference and clearing the held deduction, distinct from zero-share management-fee conversion and zero-NAV scaling.

Recommendation: Prevent a positive basis smaller than the held deduction from advancing the reference and clearing `heldManagementFeeAssets` . If forgiving the deduction on any advance is intentional, keep the rule and document that permissionless repayment can trigger it.

3F: Fixed in PR 214: a crystallizing advance now consumes the held deduction only up to the basis and carries the excess past the new mark, so a one-atom basis can no longer clear it.

Cantina Managed: Fix verified.

3.2.4 Empty-good-debt exit skips held-fee scaling and leaves an oversized deduction

Severity: Low Risk.

Context: LibStorage.sol#L166-L207 contains the affected code. The configured fee recipient misses fee shares. Remaining or later LP cohorts receive the excess fee shield.

`heldManagementFeeAssets` holds the management fees charged since the performance reference last advanced, and `PositionManagerBase.sol:223-224` deducts it from the next positive performance basis.

`LibStorage.rebaseSnapshot()` returns early at `LibStorage.sol:270` when `refDebt > 0 && newCollat == 0`, preserving the reference during a bad-debt episode. That return also skips the `newSupply / prevSupply` scaling of `heldManagementFeeAssets` at `LibStorage.sol:285-290`. The deduction belongs to the shares that incurred it, so the early return leaves the exited LPs' slice attached to the remaining shares.

```
// src/libs/manager/LibStorage.sol:270
if (refDebt > 0 && newCollat == 0) return;
```

```
// src/libs/manager/LibStorage.sol:285-290
if (newSupply != prevSupply && prevCollat > 0) {
  uint256 heldManagementFeeAssets = self.heldManagementFeeAssets;
  if (heldManagementFeeAssets > 0) {
    self.heldManagementFeeAssets = heldManagementFeeAssets.mulDiv(newSupply,
    ↪  prevSupply);
  }
}
```

Example: Debt asset USDC and collateral wUSCC are both 6 decimals, so `virtualShareOffset` is `10 ** (18 - 6) = 1e12` and shares are 18 decimals.

1. LPs deposited 1,000,000 USDC (1e12) against a 1e24 share supply, and `heldManagementFeeAssets` has reached 10,000 USDC (1e10), roughly six months at the 200 BPS management fee cap.
2. An exit burns 2e23 shares (20% of supply) and drains the last healthy position, so `newCollat == 0` while underwater positions and 8e23 shares remain.
3. The early return leaves the accumulator at 10,000 USDC (1e10) instead of scaling it to 8,000 USDC (8e9), which is `1e10 * 8e23 / 1e24`, for the remaining shares.
4. A later positive basis consumes the stale deduction, so 2,000 USDC of basis is shielded that should have been charged. At a 2000 BPS performance fee the fee recipient loses 400 USDC (4e8); at the 5000 BPS `MAX_PERFORMANCE_FEE` it is 1,000 USDC (1e9).

A partial exit is enough, zero supply is not required. A zero-supply exit has the largest effect because newly issued shares can inherit the exited LPs' full deduction. User principal is unaffected, and the client may accept fee under-collection as policy.

Recommendation: Scale `heldManagementFeeAssets` by the remaining share ratio before the empty-good-debt early return, so exited shares cannot leave their deduction with later shares. Otherwise document and accept that remaining or later LPs inherit the exited LPs' deduction and pay reduced future performance fees.

3F: Fixed in PR 214: `rebaseSnapshot` now zeroes `heldManagementFeeAssets` on the flow that burns the last share, so a departed cohort's credit can no longer shield a future cohort.

Cantina Managed: Fix verified.

3.2.5 Rebalance re-anchors the performance reference at the post-loss NAV, so recovery is charged as performance

Severity: Low Risk.

Context: PositionManagerRebalancing.sol#L45-L119, PositionManagerLP.sol#L190-L236, LibStorage.sol#L166-L207. The configured fee recipient receives the direct fee.

`rebalance()` accrues fees and captures NAV at `PositionManagerRebalancing.sol:83`, runs the operations, then calls `_rebaseReference()` at `PositionManagerRebalancing.sol:113` with the post-operation state. The `maxRebalanceLoss` check at `PositionManagerRebalancing.sol:116-123` runs after that rebase and only bounds the loss, it does not record it.

`LibStorage.rebaseSnapshot()` derives `prevCarry` from the pre-call aggregates (`LibStorage.sol:274-275`) and writes the new reference as `newDebt - carry` against the post-call collateral (`LibStorage.sol:292-295`). A rebalance is supply-neutral, so `carry == prevCarry` and the NAV lost during the call never enters it. The reference lands at the post-loss trough, and recovery back to the pre-rebalance NAV reads as a gain.

```
// src/manager/base/PositionManagerRebalancing.sol:110-123
// Rebase the performance reference to the post-rebalance state. A rebalance is a
↪  flow, not
// a gain: the share supply is unchanged, so any carried pending basis is preserved
↪  as-is and
// the rebalance neither crystallizes a performance fee nor writes off accrued debt
↪  carry.
uint256 totalAssetsAfter = _rebaseReference(totalAssetsBefore, debtBefore,
↪  ERC20.totalSupply());

// Check that totalAssets didn't decrease by more than maxRebalanceLoss
if (totalAssetsAfter < totalAssetsBefore) {
  uint256 loss = totalAssetsBefore - totalAssetsAfter;
  // loss * BPS / totalAssetsBefore > maxRebalanceLoss
  // Rearranged to avoid division: loss * BPS > maxRebalanceLoss * totalAssetsBefore
  if (loss * BPS > uint256(_storage.rebalanceConfig.maxRebalanceLoss) *
  ↪  totalAssetsBefore) {
    revert LibManagerErrors.RebalanceLossExceedsMax();
  }
}
```

```
// src/libs/manager/LibStorage.sol:271-295
uint256 carry;
if (refDebt > 0 && prevSupply > 0 && newSupply > 0) {
  uint256 refCollat = self.lastTotalAssets + refDebt;
  uint256 scaledRefDebt = refDebt.mulDivUp(prevCollat, refCollat);
  uint256 prevCarry = FixedPointMathLib.zeroFloorSub(prevDebt, scaledRefDebt);
  // Preserve the per-share carry across the supply change.
  carry = prevCarry.mulDiv(newSupply, prevSupply);
  if (carry > newDebt) carry = newDebt;
  // Preserve the per-share pending management fee deduction the same way. Skipped
  ↪  when the
  // pre-flow good-debt universe is empty (a rescue flow out of a full bad-debt
  ↪  episode):
  // shares are then minted against a zero asset base, so the supply ratio is
  ↪  unmoored from
  // any price and scaling would inflate the deduction far beyond the fees ever
  ↪  charged. The
  // accumulator stays nominal instead, matching its definition (fees charged since
  ↪  the last
  // advance).
  if (newSupply != prevSupply && prevCollat > 0) {
    uint256 heldManagementFeeAssets = self.heldManagementFeeAssets;
    if (heldManagementFeeAssets > 0) {
      self.heldManagementFeeAssets = heldManagementFeeAssets.mulDiv(newSupply,
      ↪  prevSupply);
    }
  }
}
```

```
uint256 newRefDebt = newDebt - carry;
self.lastDebt = newRefDebt;
// Good-debt aggregation guarantees newCollat >= newDebt >= newRefDebt.
self.lastTotalAssets = newCollat - newRefDebt;
```

LPs pay performance fees on recovery of value they already held. The loss can come from bridge fees, venue slippage, or rebalance rounding, so an ordinary rebalance triggers it without owner action. Scope covers the ordinary path; no separate split-flow path is asserted.

Example: Assumes a 2000 BPS performance fee, no management fee, and `maxRebalanceLoss = 100` BPS.

1. PositionManager holds 10,000,000 USDC of quoted wUSCC collateral (1e13) against 5,000,000 USDC debt (5e12), NAV 5,000,000 USDC. Reference is `lastDebt = 5e12`, `lastTotalAssets = 5e12`.
2. A rebalance loses 25,000 USDC of collateral, 50 BPS of NAV, inside the cap. `prevCarry` is 0, so `rebaseSnapshot()` re-anchors at `lastDebt = 5e12`, `lastTotalAssets = 9,975,000 - 5,000,000 = 4,975,000 USDC (4.975e12)`.
3. Collateral recovers to 10,000,000 USDC and NAV returns to 5,000,000 USDC, no new high.
4. The next accrual reads `basis = mulDivUp(5e12, 1e13, 9.975e12) - 5e12 = 12,531.328321 USDC` and mints the recipient fee shares worth 2,506.265664 USDC. The charge is the levered slice of the recovery, not the full 25,000 USDC.

Recommendation: Preserve the pre-rebalance performance reference across an allowed NAV loss, so recovery to that reference mints no performance-fee shares. If charging recovery as performance is the intended policy, document it.

3F: Acknowledged and documented on `rebalance()` in PR 211: the loss is bounded by the owner-set `maxRebalanceLoss`, rebalancing is role-gated, and the fee mints only on genuine upward movement from the post-loss state.

Cantina Managed: Acknowledged.

3.2.6 Offer fill rounding can push LTV past `liquidationLtv` and unlock near-total equity liquidation

Severity: Low Risk.

Context: LibBorrowOffers.sol#L287-L408, MorphoBorrowPosition.sol#L319-L356, MorphoBorrowPosition.sol#L438-L458.

The offer band is permissionless to consume while `safeLtv < LTV <= liquidationLtv` (`MorphoBorrowPosition.sol:360-369`). `_consumeOffers` snapshots Morpho's pre-repayment totals (`MorphoBorrowPosition.sol:446-459`), and `_priceAction` evaluates `strictlyLowersLtv` against that snapshot with collateral values rounded down and debt values rounded up. On a virtual-share boundary the fill credits a debt-asset unit that disappears once Morpho updates `totalBorrowAssets` / `totalBorrowShares`: remaining debt rounds to the same value while collateral drops, so post-repayment LTV is flat or higher even though the pre-state check accepted the fill. Independently floored collateral and ceil'd share terms give the same outcome by a different route, off by one raw loan atom.

Once LTV crosses `liquidationLtv` but stays under market `LLTV`, `preLiquidate` dispatches to the proportional path, which can repay the remaining shares and seize the remaining collateral. The liquidator reaches that in one transaction via the `onPreLiquidate` callback, or with a second call after the first fill. `liquidationLtv == LLTV` blocks the escalation (the reviewed fleet snapshot used that config), so no current deployment exposure was established.

```
// src/libs/borrow/LibBorrowOffers.sol:436-446
uint256 seizedValue = fillCollateral.mulDiv(inp.price, ORACLE_PRICE_SCALE);
uint256 repaidDebtValue = fillShares.toAssetsUp(inp.totalBorrowAssets,
↪  inp.totalBorrowShares);
if (!isProfitableAboveBonusFloor(seizedValue, repaidDebtValue,
↪  inp.minOfferBonusBps)) return FillAction.Skip;

uint256 remainingDebtValue =
↪  remainingPositionShares.toAssetsUp(inp.totalBorrowAssets,
↪  inp.totalBorrowShares);
uint256 remainingCollateralValue = remainingPositionCollateral.mulDiv(inp.price,
↪  ORACLE_PRICE_SCALE);
if (!strictlyLowersLtv(seizedValue, repaidDebtValue, remainingDebtValue,
↪  remainingCollateralValue)) {
  return FillAction.Stop;
}

return FillAction.Consume;
```

Example: wUSCC/USDC market at 1:1 oracle price, `safeLtv = 45%`, `liquidationLtv = 50%`, `LLTV = 62.5%`.

1. Position holds 9,973,308.160223 wUSCC collateral against 4,986,654.080111 USDC debt (4,326,631,081,525,540,850 borrow shares). Equity is 4,986,654.080112 USDC and LTV sits inside the band.
2. Proposer posts an offer of 2 collateral atoms for 1 debt share.
3. Liquidator calls `preLiquidate(position, 2, 0, data)`. The fill seizes 0.000002 wUSCC and repays 1 share; rounded position debt is 4,986,654.080111 USDC before and after Morpho's update, so LTV rises past 50%.
4. Inside `onPreLiquidate`, the liquidator re-enters `preLiquidate` in shares mode for all remaining shares. The proportional path repays the rest and seizes the remaining 9,973,308.160221 wUSCC.
5. Liquidator spends 4,986,654.080112 USDC and receives 9,973,308.160223 wUSCC: net 4,986,654.080111 USDC of the 4,986,654.080112 USDC starting equity, one raw atom short of the whole position.

The collateral-floor variant reaches the same cascade while making debt progress and captures the full 150.000000 USDC quoted equity.

Recommendation: After Morpho updates the debt totals, re-check the position's final balances and require LTV to be below both entry LTV and `liquidationLtv`.

3F: Fixed in PR 210: `_offerPreLiquidate` now derives the settled position frame from the fill amounts before the repay and reverts `LtvNotReduced` unless the LTV strictly decreases, computed before control ever reaches a callback (which also closes the `margin == 0` reentrant proportional-liquidation path).

Cantina Managed: Fix verified.

3.2.7 Partitioning a liquidation target skips the best offer and fills from a worse one

Severity: Low Risk.

Context: MorphoBorrowPosition.sol#L255-L329, LibBorrowOffers.sol#L287-L408.

`MorphoBorrowPosition.preLiquidate` lets a permissionless liquidator pick the collateral or debt-share target. In the offer band (`safeLtv < LTV <= liquidationLtv`) `_walk` visits offers cheapest-first for the owner, but a below-floor offer is skipped, not stopped (`LibBorrowOffers.sol:319`).

`_computeFill` rounds `fillShares` up. An offer whose whole fill sits exactly on the configured bonus floor can miss that floor by one raw share once the fill is partial, so `_priceAction` returns `Skip` for the best offer and the walk fills from a worse one. Splitting one target into two calls therefore buys the same collateral for less repaid USDC.

```
// src/libs/borrow/LibBorrowOffers.sol:389-391
if (fillCollateral == 0) return (FillAction.Skip, 0, 0);

uint256 fillShares = fillCollateral.mulDivUp(remainingDebtShares,
↪  remainingCollateral);
```

```
// src/libs/borrow/LibBorrowOffers.sol:436-438
uint256 seizedValue = fillCollateral.mulDiv(inp.price, ORACLE_PRICE_SCALE);
uint256 repaidDebtValue = fillShares.toAssetsUp(inp.totalBorrowAssets,
↪  inp.totalBorrowShares);
if (!isProfitableAboveBonusFloor(seizedValue, repaidDebtValue,
↪  inp.minOfferBonusBps)) return FillAction.Skip;
```

The invariant that must hold:

> For identical seized collateral, partitioning one liquidation into multiple calls must not materially reduce the debt repaid solely because of offer-walk rounding.

Example: Raw units, wUSCC and USDC both 6 decimals, oracle price 1:1 (`price = ORACLE_PRICE_SCALE`), `minOfferBonusBps = 100`, and `totalBorrowAssets == totalBorrowShares` so one share converts to one USDC atom.

Position: 2,000,000 wUSCC collateral, 1,700,000 USDC debt (LTV 85%, inside the band). Offer A (best for owner): 1,010,000 wUSCC for 1,000,000 shares. Offer B (worse): 1,100,000 wUSCC for 1,000,000 shares.

1. Whole target `seizedAssets = 1_010_000_000_000`. Offer A fills whole: `fillShares = 1_000_000_000_000`, excess `10_000_000_000 >= ceil(1e12 * 100 / 10000) = 10_000_000_000`, floor met. Liquidator repays 1,000,000 USDC.
2. Split call 1, `seizedAssets = 505_000_000_001`. On A, `fillShares = ceil(505_000_000_001 * 100 / 101) = 500_000_000_001`, excess `5_000_000_000 < ceil(500_000_000_001 / 100) = 5_000_000_001`. A is skipped; B fills at 1.1, repaying `459_090_909_092`.
3. Split call 2, `seizedAssets = 504_999_999_999`. On A, `fillShares = 500_000_000_000`, excess `4_999_999_999 < 5_000_000_000`. A is skipped again; B repays `459_090_909_090`.
4. Same 1,010,000 wUSCC seized, 918,181.818182 USDC repaid instead of 1,000,000. The owner loses about 81,818 USDC of debt reduction, and offer A stays unconsumed.

Both fills pass `strictlyLowersLtv` (fill ratio 1.1 is below the position's 2.0/1.7), and empty callback data is enough.

Recommendation: Document that offer priority assumes liquidators do not partition targets to skip a better offer.

3F: Fixed in PR 210: the profitability and bonus-floor gate is evaluated once per offer on its whole remaining amounts, so the fill decision is independent of how the liquidator partitions the target.

Cantina Managed: Fix verified.

3.2.8 Maker callback in `Request.consume()` can invalidate the `maxPrincipal` cap checked by `Retargetter.consume()`

Severity: Low Risk.

Context: Retargetter.sol#L211-L223, LibStorage.sol#L149-L175, Request.sol#L345-L369.

`Retargetter.consume()` reads `maxPrincipal(positionManager)` and passes it to `LibStorage.checkOffer`, which reverts with `PrincipalCapExceeded` when PT supply plus outstanding authorizations plus `ptAmount` exceeds the cap (`LibStorage.sol:174-176`). It then calls `Request.consume()`, which invokes the maker's `onRequestConsumed` before pulling principal and minting.

`maxPrincipal` is derived from live `collateralAmountQuoted()` and `debtAmount()` (`Retargetter.sol:688-711`). A callback that liquidates the PositionManager (BorrowOffer, proportional `preLiquidate`, or native Morpho liquidation) shrinks the cap. Control returns to `Request.consume()`, which transfers PT and mints PT/YT against a cap that no longer holds, so the operation takes on a Request liability plus its minimum tick of yield priced against capacity that is gone. The liquidation loss and bonus exist independently and are excluded from this issue's impact.

```
// src/manager/rebalancer/Retargetter.sol:247-255
{
  RetargetterOperation storage operation_ = LibStorage.operationStorage();
  address request = operation_.checkRequest();
  operation_.checkOffer(
    request, offer.amount, offer.expectedReturn, ptAmount,
    ↪  maxPrincipal(operation_.positionManager)
  );
  ytAmount = IRequest(request).consume(offer, signature, ptAmount);
  emit OfferConsumed(request, offer.maker, ptAmount, ytAmount);
}
```

```
// src/request/Request.sol:419-429
if (_syncWithdrawalStatus()) revert LibRequestErrors.AlreadyRepaid();
if (ptAmount == 0 || ptAmount > offer.amount) revert
↪  LibRequestErrors.InvalidPtAmount();
_validateOffer(offer, signature);
ytAmount = offer.expectedReturn.mulDiv(ptAmount, offer.amount);
if (offer.useCallback) {
  IRequestCallback(offer.maker).onRequestConsumed(offer, signature, ptAmount,
  ↪  ytAmount);
}
_asset().safeTransferFrom(offer.maker, address(this), ptAmount);
_mint(offer.maker, ptAmount, ytAmount);
_requestStorage().lastMintTimestamp = uint40(block.timestamp);
```

Example:

1. ASYNC operation on a PositionManager holding wUSCC collateral against USDC debt; `maxPrincipal` reads 500,000e6 USDC, PT supply 0.
2. Consumer calls `Retargetter.consume()` for an offer of `ptAmount = 400,000e6` USDC with `useCallback = true`. `checkOffer` passes (400,000e6 ≤ 500,000e6).
3. Inside `onRequestConsumed`, the maker liquidates the position; collateral drops and the recomputed `maxPrincipal` falls to 150,000e6 USDC.
4. `Request.consume()` continues and mints 400,000e6 PT plus the proportional YT, 250,000e6 USDC of principal above the live cap.

Recommendation: Re-read `maxPrincipal(operation_.positionManager)` after `IRequest(request).consume()` returns and revert if the committed principal exceeds the updated cap.

3F: Fixed in PR 213: the principal-cap gate moved after `Request.consume()` returns, reading PT supply plus pending authorizations against the cap, so a maker callback can no longer invalidate the capacity the mint lands against.

Cantina Managed: Fix verified.

3.2.9 `Retargetter.maxPrincipal` can exceed the one-trip repayment headroom of a fully deployed LTV-up operation

Severity: Low Risk.

Context: Retargetter.sol#L392-L455, Retargetter.sol#L625-L647, Retargetter.sol#L765-L776.

An ASYNC LTV-up operation caps Request funding at `maxPrincipal`, which is the quoter's ideal principal scaled up by `principalBufferBps` (`Retargetter.sol:711`). Settlement prices the Request repayment from actual PT/YT supply and tick-quantized elapsed time (`Retargetter.sol:832-844`), while the debt leg that funds it is bounded by the PositionManager target LTV. The buffer inflates the amount the operation can owe but not the amount it can borrow back, so a cap-sized, fully deployed operation can owe more than the post-supply borrow capacity.

For a non-owner caller the direction guard rejects the settlement rebalance when the resulting LTV is above target and did not improve (`Retargetter.sol:495-508`), so the shortfall surfaces as a revert, not as an over-target position.

```
// src/manager/rebalancer/Retargetter.sol:700-712
  uint256 principal = IRetargetterQuoter(_QUOTER)
    .retargetPrincipal(
      collateralQuoted,
      debt,
      target,
      estimates.requestYieldRate,
      estimates.borrowRate,
      estimates.collateralYieldRate,
      estimates.subscriptionDuration,
      estimates.redemptionDuration
    );
  return principal * (BPS + config_.principalBufferBps) / BPS;
}
```

```
// src/manager/rebalancer/Retargetter.sol:495-499
if (msg.sender != owner()) {
  uint256 ltvAfter = _positionManagerLtv(positionManager);
  if (ltvAfter > target && ltvAfter >= ltvBefore) {
    revert LibRetargetterErrors.AboveTargetLtv(ltvAfter, ltvBefore, target);
  }
```

Example: Position: 10,000,000 USDC of quoted wUSCC collateral, 5,000,000 USDC debt, target LTV 0.7e18, `principalBufferBps = 100`, `maxYieldBps = 1000`, `horizon = 365 days`, `tickDuration = 1 days`, zero rate estimates.

1. Ideal principal: `(7,000,000 - 5,000,000) * 1e18 / 0.3e18 = 6,666,666.666666` USDC.
2. `maxPrincipal` applies the buffer: `6,666,666.666666 * 10100 / 10000 = 6,733,333.333332` USDC.
3. The full cap is subscribed and the wUSCC lands as collateral: quoted collateral becomes 16,733,333.33 USDC, so borrow capacity at target is `16,733,333.33 * 0.7 - 5,000,000 = 6,713,333.33` USDC.
4. After one tick, `repaymentOwed` = PT 6,733,333.333332 + `ceil(673,333.333333 * 1 / 365)` = 6,735,178.08 USDC.
5. Shortfall: 6,735,178.08 - 6,713,333.33 = **21,844.75 USDC**. `repay()` cannot be funded from the position in one trip, and the borrow that would cover it trips the direction guard.

Recommendation: Size `maxPrincipal` against the worst permitted `_owed` repayment (PT plus the full `maxYieldBps` yield) and the post-supply target-LTV borrow capacity, so a cap-sized operation keeps enough one-trip settlement headroom. If the shortfall is accepted instead, document that cap-sized operations require reserved headroom, a top-up, or a second Fund trip before settlement.

3F: Fixed in PR 213: `maxPrincipal` is now bounded by `min(buffered, ltvUpOneTripPrincipal)`, the largest principal whose worst permitted repayment is still borrowable at target.

Cantina Managed: Fix verified.

3.2.10 A supply-only Retargetter rebalance lets holders of existing `PositionManager` shares withdraw Request-funded collateral

Severity: Low Risk.

Context: Retargetter.sol#L392-L455, PositionManagerRebalancing.sol#L45-L149, PositionManagerLP.sol#L41-L178, FacilityPositionManager.sol#L106-L134.

A funded asynchronous Retargetter operation can convert Request principal into collateral. The operation can supply that collateral to a PositionManager without borrowing the debt asset needed to repay the Request.

`Retargetter.rebalance()` accepts the operation list from an owner or `REBALANCER_ROLE` holder. `Retargetter.rebalance()` resolves the requested amounts and forwards the list to the PositionManager:

```
if (resolved.collateral > 0) {
  collateralAsset.safeApproveWithRetry(positionManager, resolved.collateral);
}
IPositionManager(positionManager).rebalance(resolved, address(this));
```

The PositionManager dispatches each operation independently. A `SUPPLY` operation transfers collateral into a borrow module. It does not mint PositionManager shares for that Request-funded supply or record a liability to Request lenders. Unrelated fee accrual can still mint fee shares during the same call:

```
if (operationType == RebalancingOperationType.SUPPLY) {
  position.supply(_collateralAsset, amount);
}
```

This supply increases `totalAssets()` without corresponding share issuance for the supplied value. Fee accrual may mint unrelated fee shares, but no shares represent the Request-funded collateral. The rebalance loss check only handles a decrease in `totalAssets()`. The increase passes:

```
if (totalAssetsAfter < totalAssetsBefore) {
  uint256 loss = totalAssetsBefore - totalAssetsAfter;
  // Check loss against maxRebalanceLoss.
}
```

Existing PositionManager shares now include the Request-funded collateral in their burn value. `PositionManager.burn()` calculates collateral from total module collateral and the shares to burn. `PositionManager.burn()` does not reserve collateral for the active Retargetter operation.

A normal Facility exit can therefore transfer the staged Request collateral to existing Facility LPs. The `FACILITATOR_ROLE` holder calls `burnManager()` and resolves the intent. The LP then calls `claim()`.

1. A Facility LP deposits 10,000 collateral into an intent.
2. `Facility.depositManager()` creates a PositionManager position with 10,000 collateral and 4,000 debt.
3. The PositionManager has 6,000 NAV. The Facility holds the PositionManager shares.

4. A Retargetter operation converts 5,000 of Request principal into collateral.
5. The Retargetter rebalancer executes one `SUPPLY` operation for 5,000 collateral and no `BORROW` operation.
6. PositionManager NAV increases from 6,000 to 11,000, but no shares are minted to represent the Request-funded increase.
7. The Facility calls `burnManager()` with its existing shares and tracked debt balance.
8. The Facility resolves the intent. The LP calls `claim()`.

The exit transfers `14,999.999999999999999997` collateral to the LP. This amount includes `4,999.999999999999999997` of the 5,000 Request-funded units. Only 3 wei remain in the PositionManager.

The Request still records the repayment obligation. After the Facility exit, the PositionManager can lack enough collateral to borrow the debt asset needed to repay the Request. Request lenders then depend on an external top-up or other recovery.

Recommendation: This has not a clear fix yet, the invariant that must be preserved: `A SUPPLY funded by Request capital must be matched by an equivalent BORROW of debt-asset back out of the PM, so the borrowed debt can repay the Request.`

Initial fix: Fixed in PR 209: `rebalance()` now snapshots the position's net value over every module and reverts `PositionValueIncreased` if it grew while the operation's Request is unrepaid, so bridge capital can only enter paired with an equivalent output leg.

Second issue: We might have another issue now.

The new value-conservation gate only stays active while the Request is considered outstanding. After the deadline, `_bridgeOutstanding()` calls `syncRepaidStatus()`, which marks the Request as repaid even when Retargetter has already pulled lender principal.

For example:

- Request receives 3,000 units from the lender.
- Retargetter pulls 2,000 units.
- The Request expires.
- `syncRepaidStatus()` marks it repaid.
- The value gate is disabled.
- A rebalance folds the 2,000 pulled units into PositionManager NAV.
- The maker can recover only the 1,000 units still held by the Request.

The expiry path therefore treats "deadline passed" as "principal repaid." Should the value gate should remain active until all pulled principal is returned or explicitly written off through a governed default process?

3F: Fixed in PR 220.: the value gate now keys on a Retargetter-local repaid flag set only by repay()/forceRepay() (the only possible setRepaid callers, since the Retargetter owns the Request), so deadline expiry never disarms it, and forceRepay now tolerates an expired Request (bounds enforced locally) so pulled principal can still be delivered back to lenders and the operation resolved.

One acknowledged caveat is documented: post-expiry redemptions price on the live Request balance, so holders should not burn their PT/YT before an expected late delivery lands.

Cantina Managed: Fix verified.

3.2.11 Held management-fee credit is scaled by a deposit made at zero NAV

Severity: Low Risk.

Context: LibView.sol#L65-L83, LibStorage.sol#L166-L207, PositionManagerLP.sol#L41-L83.

`heldManagementFeeAssets` is the management fee already charged, repaid out of the next positive performance basis before any performance fee mints. On every supply change `rebaseSnapshot` scales it by the supply ratio, guarded on `prevCollat > 0`:

```
// src/libs/manager/LibStorage.sol:285-290
if (newSupply != prevSupply && prevCollat > 0) {
  uint256 heldManagementFeeAssets = self.heldManagementFeeAssets;
  if (heldManagementFeeAssets > 0) {
    self.heldManagementFeeAssets = heldManagementFeeAssets.mulDiv(newSupply,
    ↪  prevSupply);
  }
}
```

The comment at `LibStorage.sol:220-224` states the guard exists to skip scaling when shares mint against a zero asset base, because the supply ratio no longer tracks any price. It tests gross collateral, and the caller passes `prevCollat = totalAssetsBefore + debtBefore` (`PositionManagerLP.sol:236`). A module at `collateral == debt` is still included by `LibView.totalAssets`, whose filter is `collateral >= debt`, so NAV is zero while `prevCollat = debt > 0` and the guard passes.

```
// src/libs/manager/LibView.sol:88-96
uint256 collateral = IBorrowPosition(modules[i]).totalCollateralQuoted();
uint256 debt = IBorrowPosition(modules[i]).totalBorrowed();
if (collateral >= debt) {
  amount += collateral - debt;
```

With `totalAssetsBefore == 0` the mint denominator collapses to `VIRTUAL_ASSETS = 1`, so a 1-unit deposit mints roughly the entire existing supply again:

```
// src/manager/base/PositionManagerLP.sol:202-203
uint256 assetsAdded = totalAssetsAfter - totalAssetsBefore;
uint256 sharesToMint = assetsAdded.convertToShares(_totalSupply, totalAssetsBefore,
↪  virtualShareOffset_, false);
```

The doubled supply ratio doubles the credit, and the extra credit is subtracted from the performance basis at `PositionManagerBase.sol:223-224`, cutting the fee recipient's take.

Example: USDC debt asset and wUSCC collateral, both 6 decimals (`USCCFund.sol:50`), so `virtualShareOffset = 1e12`. Shares are 18 decimals. Reference held at 80% LTV during a drawdown:

```
collateral  5,000,000 USDC of wUSCC (5e12)   debt  5,000,000 USDC (5e12)   NAV 0
supply      1e24 (1,000,000 shares)
heldManagementFeeAssets  10,000 USDC (1e10)
lastDebt 4e12, lastTotalAssets 1e12  (lastCollat 5e12, LTV_ref 80%)
```

1. `deposit(1, 0)` supplies 1 base unit of wUSCC (0.000001 wUSCC). `totalAssetsAfter = 1`, so `assetsAdded = 1` and `sharesToMint = 1 * (1e24 + 1e12) / (0 + 1) = 1e24 + 1e12`. Supply goes to `2e24 + 1e12`.
2. `rebaseSnapshot` runs with `prevCollat = 0 + 5e12 = 5e12`, so the guard passes: `heldManagementFeeAssets = 1e10 * (2e24 + 1e12) / 1e24 = 2e10`, i.e. **10,000 USDC becomes 20,000 USDC**. Carry scales the same way (`1e12` to `2e12 + 1`), re-anchoring the reference at `lastDebt = 3e12 - 1`, `lastCollat = 5e12 + 1`, about 60% LTV.

3. WUSCC appreciates to 10,000,000 USDC of collateral against the same 5,000,000 USDC debt. Basis (`PositionManagerBase.sol:193-195`): `mulDivUp(3e12 - 1, 1e13, 5e12 + 1) - 5e12 = 1e12` , a 1,000,000 USDC basis.
4. Performance fee is charged on `basis - managementFeeAssets - heldManagementFees_` . The inflated credit shrinks that base by 10,000 USDC, so at `MAX_PERFORMANCE_FEE = 5000` BPS (`LibConstants.sol:41`) the fee recipient loses **5,000 USDC**.

The deposit must stay small. A larger one pushes `carry` above `newDebt` , which clamps `lastDebt` to the bootstrap sentinel and makes the next accrual clear the accumulator.

Recommendation: Gate the scaling on pre-flow NAV instead of gross collateral:

```
if (newSupply != prevSupply && prevCollat > prevDebt) {
```

The ratio still detaches from price just above zero NAV, so also cap the scaled value at the pre-flow credit when post-flow NAV per share falls below pre-flow.

3F: Fixed in PR 214: the supply-ratio rescaling was removed entirely, so `heldManagementFeeAssets` stays nominal across all flows and the zero-NAV deposit lever (in both directions) is gone.

Cantina Managed: Fix verified.

3.2.12 Splitting one gain across many checkpoints can erase the performance fee

Severity: Low Risk.

Context: PositionManagerBase.sol#L180-L225, PositionManagerBase.sol#L257-L305, PositionManagerRebalancing.sol#L49-L106.

A positive performance basis sets `advanceReference = true` before the performance fee is represented as shares. The fee rounds down once during the BPS multiplication (`PositionManagerBase.sol:223-224`) and again during `convertToShares(..., false)` (`PositionManagerBase.sol:257-266`).

```
uint256 scaledLastDebt = _lastDebt.mulDivUp(currentCollat, lastCollat);
if (scaledLastDebt > currentDebt) {
  basis = scaledLastDebt - currentDebt;
  advanceReference = true;  // <-- set here, before fee share conversion
}
```

First round down:

```
// PositionManagerBase.sol:223-224
if (fd.performanceFee > 0 && basis > managementFeeAssets + heldManagementFees_) {
  totalFeeAssets += (basis - managementFeeAssets -
  ↪  heldManagementFees_).mulDiv(fd.performanceFee, BPS);
}
```

Second:

```
// PositionManagerBase.sol:260-261
performanceFeeShares =
  totalFeeAssets.convertToShares(totalSupply_, totalAssets_ - totalFeeAssets,
  ↪  _storage.virtualShareOffset, false);
```

`_accrueFees()` mints only when the resulting `feeShares` is nonzero, but it still advances `lastTotalAssets` and `lastDebt` whenever `advanceReference` is true (`PositionManagerBase.sol:289-305`). A gain whose fee rounded to zero shares is therefore consumed by the new reference and cannot contribute to a later fee.

`rebalance()` calls `_accrueFees()` before processing its funding and operation list. A `REBALANCER_ROLE` holder can consequently submit zero-funding, zero-operation rebalances after successive gain observations, subject to the configured cooldown. If every interval produces less than one raw fee share, every checkpoint mints zero even though one checkpoint over the same aggregate gain would mint a material fee.

Example: Assume one cumulative gain would entitle the fee recipient to 10 raw fee shares if accrued once. If the same gain is observed across 20 equal intervals, each interval economically corresponds to 0.5 raw fee share and `convertToShares(..., false)` rounds it down to zero. Every checkpoint then advances the performance reference and discards that fractional entitlement. After all 20 checkpoints, the fee recipient has received zero shares, whereas one checkpoint over the same cumulative gain would have minted 10 shares.

The standalone impact is lost performance-fee revenue. Direct profit requires the checkpoint actor to be aligned with the existing LP cohort; otherwise the action primarily griefs the fee recipient. The path also requires a nonzero performance fee, high NAV per raw share, and a feasible checkpoint cadence. It does not expose unrelated user principal.

Recommendation: Carry fractional performance-fee entitlement across checkpoints in fee-asset or higher-precision share units. Do not advance the performance reference until any rounded remainder has been conserved. As defense in depth, restrict economically empty checkpoints and revoke or epoch operational roles during ownership handoff.

Initial Fix: Fixed in PR 211: the reference is held when a nonzero performance entitlement converts to zero fee shares, so the basis accumulates across checkpoints until it mints at least one share.

Second Issue: I think we might have added a new issue if i see it correctly:

A deposit scales a held performance entitlement into a fee on fresh principal

Context: LibStorage.sol#L336-L348, PositionManagerLP.sol#L41-L75, PositionManagerLP.sol#L194-L239.

The fix correctly holds the performance reference when a positive entitlement rounds down to zero fee assets or zero fee shares. This prevents repeated checkpoints from consuming an entitlement that was never minted.

When a capital flow subsequently reaches `rebaseSnapshot`, however, the held positive gain is multiplied by the change in share supply:

```
if (prevCarry == 0) {
  gain = FixedPointMathLib.zeroFloorSub(scaledRefDebt, prevDebt)
    .min(FixedPointMathLib.zeroFloorSub(prevCollat - prevDebt,
    ↪  self.lastTotalAssets));

  gain = gain.mulDiv(newSupply, prevSupply);
}
```

`gain` is an asset-denominated performance basis. Increasing share supply through a deposit does not represent profit and should not increase the total fee entitlement.

`deposit()` accrues fees before transferring the new capital. If the existing entitlement rounds to zero, the reference remains held and no fee shares are minted. The deposit then mints shares and calls `rebaseSnapshot`, where the new supply ratio scales the old gain. The next accrual can consequently mint a performance fee against the incoming principal.

Example

Assume a PositionManager has three atoms of NAV and one atom of debt. A third party repays one debt atom through Morpho, producing a one-atom positive basis. At a 15% performance fee, the entitlement rounds to zero fee assets, so the reference is held as intended.

A subsequent pure deposit of `1e18` increases the share supply. The current rebase scales the one-atom basis to:

```
266666666666666667
```

The next accrual mints:

```
33333333333333333 fee shares
```

The fresh depositor's proportional claim falls from:

```
999999999999999999
```

to:

```
959999999999999999
```

This is a loss of `40000000000000000`, or 4% of the deposited principal, despite no profit occurring after the deposit.

Will double check.

Second fix: Fixed in PR 218: the held entitlement is now kept nominal across flow rebases (like the held management-fee accumulator, the supply ratio being a value-detached lever in both directions), falling back to the supply-scaled read only once the gain outgrows half the post-flow NAV so a draining exit cannot truncate the high-water mark to zero and re-read the residual NAV as basis.

Third Issue: I think now we might have another issue, we now stop deposits from scaling a held performance gain into a fee on fresh principal. However, the new half-NAV fallback can still scale that gain down during an exit.

Example:

- NAV starts at 10.
- Share supply starts at 3.
- Held performance gain is 7.
- A deposit of 11 changes NAV and supply to 21 and 7, while the gain stays at 7.
- Withdrawing the same 11 restores NAV and supply to 10 and 3, but reduces the gain to 3.

The deposit and withdrawal restore the vault state, but the held fee entitlement loses 4 units. The remaining LPs benefit because less performance fee will be minted later.

Third Fix: Acknowledged and documented in PR 218. the round trip can only shed fee-recipient value (the reference is read by fee accrual alone, and grinding the gain raises the mark), the zero-share hold bounds any genuine entitlement in the above-half-NAV region below one raw share's worth of fees (dust outside atom-scale vaults), and the window gains large enough to reach the fallback never mint on re-entry anyway, so we prefer this residual over a half-NAV cap that would charge the stayers for the departed shares' slice.

3.2.13 Performance reference is rebased after outgoing token transfers, so a token callback can move the basis

Severity: Low Risk.

Context: PositionManagerLP.sol#L111-L157, PositionManagerRebalancing.sol#L71-L108.

`PositionManagerLP.burn()` sends collateral to the caller at `PositionManagerLP.sol:171` and only then calls `_rebaseReference()` at `PositionManagerLP.sol:176`. `PositionManagerRebalancing.rebalance()` does the same: excess collateral and debt go to `receiver` at `PositionManagerRebalancing.sol:100-101`, rebase at `PositionManagerRebalancing.sol:113`. `_rebaseReference`

() reads live aggregates via _storage.totalAssets() (PositionManagerBase.sol:330), so anything that changes module collateral or debt between the transfer and the rebase lands in the new reference.

A callback-enabled token hands control to the receiver during those transfers. nonReentrant blocks reentry into the PositionManager, but not a direct call to Morpho. Morpho supply/repay are permissionless on behalf of a module, so the receiver can raise collateral or cut debt for a PositionManager module without touching the PositionManager, and the rebase then snapshots that state as the new performance basis. This shifts the basis for a later performance fee up or down.

```
// src/manager/base/PositionManagerLP.sol:169-176
// Send collateral to caller
if (collateral > 0) {
  _storage.metadata.collateralAsset.safeTransfer(msg.sender, collateral);
}

// Rebase the performance reference across the flow so the exiting shares take their
// proportional slice of any carried pending basis with them.
_rebaseReference(totalAssetsBefore, debtBefore, _totalSupply);
```

```
// src/manager/base/PositionManagerRebalancing.sol:100-113
collateralExcess = _collateralAsset.safeTransferAll(receiver);
debtExcess = _debtAsset.safeTransferAll(receiver);

// Record rebalance timestamp for cooldown enforcement
// Safe: block.timestamp fits in uint40 for ~35,000 years
// forge-lint: disable-next-line(unsafe-typecast)
_storage.rebalanceConfig.lastRebalanceTimestamp = uint40(block.timestamp);

emit Rebalanced(receiver, data.collateral, data.debt, collateralExcess, debtExcess);

// Rebase the performance reference to the post-rebalance state. A rebalance is a
↪  flow, not
// a gain: the share supply is unchanged, so any carried pending basis is preserved
↪  as-is and
// the rebalance neither crystallizes a performance fee nor writes off accrued debt
↪  carry.
uint256 totalAssetsAfter = _rebaseReference(totalAssetsBefore, debtBefore,
↪  ERC20.totalSupply());
```

Recommendation: Snapshot the final module aggregates before the outgoing transfers and rebase the reference from that snapshot.

3F: Fixed in PR 211: _rebaseReference (and in rebalance() the maxRebalanceLoss check) now run before the outgoing transfers, so a token-callback donation lands after the reference and is fee-bearing like any external repay.

Cantina Managed: Fix verified.

3.2.14 Retargetter binds to an unauthenticated PositionManager address

Severity: Low Risk.

Context: Retargetter.sol#L197-L234, Retargetter.sol#L786-L826, Retargetter.sol#L440-L512.

startRetargetting takes the positionManager address as a caller-supplied argument and stores it as the operation's binding (Retargetter.sol:197-231). _checkStart is the only gate, and it never checks that the address is a real PositionManager:

```
// src/manager/rebalancer/Retargetter.sol:791-800
function _checkStart(RetargetterOperation storage operation_, address
↪  positionManager, address fund, uint256 amount)
  internal
  view
{
  if (operation_.positionManager != address(0)) revert
  ↪  LibRetargetterErrors.OperationActive();
  if (operation_.orderLive) revert LibRetargetterErrors.OrderActive();
  _checkPair(positionManager);
  if (!LibStorage.whitelistsStorage().funds[fund]) revert
  ↪  LibRetargetterErrors.FundNotWhitelisted();
  if (amount > maxPrincipal(positionManager)) revert
  ↪  LibRetargetterErrors.PrincipalCapExceeded();
}
```

Both checks that touch `positionManager` read values the address reports about itself. `_checkPair` calls `assets()` on it, and `maxPrincipal` calls `collateralAmountQuoted()`, `debtAmount()`, and `config()` on the same address, then feeds those into the quoter to size the principal cap (`Retargetter.sol:688-712`):

```
// src/manager/rebalancer/Retargetter.sol:820-826
function _checkPair(address positionManager) internal view {
  RetargetterAssets storage assets_ = LibStorage.assetsStorage();
  (address collateralAsset, address debtAsset) =
  ↪  IPositionManager(positionManager).assets();
  if (collateralAsset != assets_.collateralAsset || debtAsset != assets_.debtAsset)
  ↪  {
    revert LibRetargetterErrors.AssetMismatch();
  }
}
```

A contract that returns the configured asset pair and any collateral/debt numbers passes both. `rebalance` then approves the bound address for the pulled principal and calls its `rebalance()`:

```
// src/manager/rebalancer/Retargetter.sol:487-492
if (resolved.collateral > 0) collateralAsset.safeApproveWithRetry(positionManager,
↪  resolved.collateral);
if (resolved.debt > 0) debtAsset.safeApproveWithRetry(positionManager,
↪  resolved.debt);
IPositionManager(positionManager).rebalance(resolved, address(this));
if (resolved.collateral > 0) collateralAsset.safeApprove(positionManager, 0);
if (resolved.debt > 0) debtAsset.safeApprove(positionManager, 0);
```

The post-call LTV checks at `Retargetter.sol:496-508` read `_positionManagerLtv(positionManager)` and `_moduleLtv()` on modules the same contract returns, so they are self-reported too.

Funds and flash-loan modules are owner-whitelisted (`setFund` at `Retargetter.sol:617-628`, `setFlashLoanModule` at `:636`). PositionManagers are not, even though they are the address that receives the approval.

Example: USDC debt asset and wUSCC collateral, both 6 decimals (`USCCFund.sol:50`). `EvilManager` is a contract returning the configured pair from `assets()`, fabricated healthy values from `collateralAmountQuoted()` / `debtAmount()` / `config()`, and an empty `borrowModules()`.

1. `REBALANCER_ROLE` calls `startRetargetting(evilManager, 2,000,000 USDC (2e12), ...)`. `_checkPair` passes on the reported pair. `EvilManager` reports 10,000,000 USDC of quoted collateral against 4,000,000 USDC of debt, so `maxPrincipal` returns a cap above the requested principal and `_checkStart` passes.

2. `CONSUMER_ROLE` calls `consume()` for 2,000,000 USDC of PT. The principal gate re-checks `maxPrincipal(evilManager)`, which the same contract answers.
3. `REBALANCER_ROLE` calls `pullRequestFunds(2,000,000 USDC)`, moving lender USDC from the Request into the Retargetter.
4. `REBALANCER_ROLE` calls `rebalance()` with `resolved.debt = 2,000,000 USDC`. The Retargetter approves `EvilManager` for 2e12 and calls its `rebalance()`, which runs `transferFrom(retargetter, attacker, 2e12)`.
5. `EvilManager` returns healthy LTVs, so the direction checks pass. The Request still records a 2,000,000 USDC repayment obligation with a 90-day deadline and no assets behind it.

Lenders lose the full 2,000,000 USDC of principal.

Recommendation: Gate the PositionManager binding the same way funds and flash-loan modules are gated: an owner-curated whitelist checked in `_checkStart`, so `startRetargetting` can only bind an address governance has approved.

A `PositionManagerFactory` deployment check is weaker on its own, since the factory is permissionless, but it does pin the implementation. Combining the two (factory-deployed and whitelisted) gives both an approved instance and a known implementation.

3F: Fixed in PR 213: each Retargetter is bound to a single owner-set PositionManager with the asset pair derived write-once from it, and operations run against the binding instead of a caller-supplied address.

Cantina Managed: Fix verified.

3.2.15 Shared Midas redemption capacity can be exhausted after the Fund pays its bond

Severity: Low Risk.

Context: MidasFund.sol#L300-L376, MidasFund.sol#L455-L488, MidasFund.sol#L528-L540, MidasFund.sol#L800-L856.

A Midas redeem is split across two separately submitted commits. The first commit pays the Repay-and-Redeem bond and records `bondPaid`; only after `PAYMENT_ROLE` calls `unlockInstantRedeem()` can the second commit call the external Midas redemption vault:

```
// MidasFund.sol:368-372
} else if (!$.instantRedeemUnlocked) {
  _legAmount = _commitBondLeg($, _currentOrderId, order.input);
} else {
  _legAmount = _commitRedeemLeg($, order);
}

// MidasFund.sol:812-817
uint256 _bondAmount = _input * $.bondConfig.amount / BPS;
IWrappedAsset($.wrappedShare).burn(msg.sender, address(this), _bondAmount);
$.mToken.safeTransfer(_recipient, _bondAmount);
$.bondPaid = _bondAmount;
```

The bond leg does not reserve the external redemption vault's daily capacity, payment-token allowance, or downstream liquidity. Those resources may be shared with ordinary users of the same Midas product. A third party can therefore consume the remaining shared capacity between the two commits. The Fund's second commit then reaches `redeemInstant()` and reverts:

```
uint256 _redeemAmount = order.input - $.bondPaid;
IWrappedAsset($.wrappedShare).burn(msg.sender, address(this), _redeemAmount);
_mToken.safeApproveWithRetry(_redemptionVault, _redeemAmount);
IMidasRedemptionVault(_redemptionVault).redeemInstant(
```

```
  _asset, _redeemAmount, _minOutput * $.assetScale
);
```

The transaction reverts atomically, but the earlier bond payment remains final. `cancel()` is unavailable once `bondPaid > 0`, and recovery can only end the unsettled order while relying on Midas to return the bond off-band. The Fund may retry after limits reset, so the primary impact is delayed settlement and conditional bond forfeiture rather than immediate theft.

Example: Assume the selected Midas route has a 2,000,000 mGLOBAL daily redemption limit. A Facility begins a 1,000,000 mGLOBAL redeem with a 5% bond. The first commit transfers 50,000 mGLOBAL to the bond recipient, leaving 950,000 mGLOBAL for the redemption leg.

Before that leg is submitted, another allowed Midas user consumes enough of the shared daily limit to leave less than 950,000 mGLOBAL of capacity. The Fund's `redeemInstant()` call reverts. The Facility must wait for capacity to reset or enter recovery, and the 50,000 mGLOBAL bond is not restored on-chain.

Recommendation: There is not much that can be done to prevent this other than documenting the risk.

3F: Acknowledged, no code change: the repay-and-redeem flow will be reserved to 3F operationally, and the finding itself concluded the residual shared-capacity risk can only be documented.

Cantina Managed: Acknowledged.

3.2.16 `FACILITATOR` can burn an intent's entire Midas share balance by overstating the redeem input

Severity: Low Risk.

Context: FacilityFunds.sol#L39-L73, FacilityFunds.sol#L94-L119, MidasFund.sol#L300-L376, MidasFund.sol#L800-L856.

`FacilityFunds.create()` lets `FACILITATOR_ROLE` choose the order `amount`, which becomes `order.input`, without bounding it against the Midas shares attributed to the selected intent:

```
function create(uint256 id, uint256 amount, uint256 minAmountOut, Mode mode)
  external onlyRoles(FACILITATOR_ROLE) returns (Order memory order)
{
  order = Order({
    mode: mode,
    owner: address(this),
    receiver: address(this),
    input: amount,
    output: minAmountOut,
    salt: keccak256(abi.encode(address(this), block.timestamp, id))
  });
  IFund(_fund).create(order);
}
```

For a Midas redeem, the first commit calculates the bond from that caller-selected input rather than from the intent's actual share balance:

```
uint256 _bondAmount = _input * $.bondConfig.amount / BPS;
IWrappedAsset($.wrappedShare).burn(msg.sender, address(this), _bondAmount);
$.mToken.safeTransfer(_recipient, _bondAmount);
$.bondPaid = _bondAmount;
```

The Midas Fund returns `order.input` as the committed amount for both legs. Consequently, the Facility's `_committedAmount == _order.input` check passes even when the first leg only consumed

the computed bond. Its balance snapshot then records the real token decrease, but does not reject an order whose declared input exceeds the selected intent's custody.

A sufficiently overstated input makes the bond equal the entire real share balance. The first leg succeeds and transfers all shares to the bond recipient. After `unlockInstantRedeem()`, the second leg attempts to burn `order.input - bondPaid`, which does not exist and reverts. `cancel()` is unavailable because the bond has already been paid. Recovery can end the order, but the bond is only restored if Midas returns it off-band.

Example: The selected intent has 100 wrapped mGLOBAL shares and the configured bond is 5%. The FACILITATOR creates a redeem order with `order.input = 2,000`.

- Bond: `2,000 * 5% = 100` shares. The first commit succeeds and transfers all 100 real shares to the bond recipient.
- Remaining redeem amount: `2,000 - 100 = 1,900` shares. The second commit reverts because the Facility no longer holds them.
- With the honest `order.input = 100`, the bond would be only 5 shares and 95 shares would remain redeemable.

The FACILITATOR does not receive the bond directly and the impact is limited to a role-authorized operation, but the selected intent can lose its entire staged Midas share balance rather than the configured 5% bond.

Recommendation: In `FacilityFunds.create()`, require the redeem `amount` to be no greater than the selected intent's internally accounted Fund-share balance. As defense in depth, `MidasFund` can also reject a redeem whose full `order.input` exceeds the caller's wrapped-share balance, but the per-intent Facility check is required because the Facility may custody shares for several intents in one address.

3F: Fixed in PR 212: `setBondConfig` / `removeBondConfig` moved from VAULT_MANAGER_ROLE to owner/OPERATOR_ROLE, so an overstated bond can only land at an operator-vouched, Midas-controlled address recoverable off-band (an input bound is not implementable because the bond is paid with borrowed assets before full commitment).

Cantina Managed: Fix verified.

3.2.17 Permissioned mToken deposits can become stuck when vault greenlisting is disabled

Severity: Low Risk.

Context: *(No context files were provided by the reviewer).*

Description: `MidasFund._checkVaultAccess()` only validates the Fund and `WrappedAsset` against the Midas greenlist when `vault.greenlistEnabled()` is true (`MidasFund.sol:940-953`). This incorrectly assumes the vault flag also controls the mToken's transfer restrictions.

For permissioned mTokens such as mGLOBAL, `mTokenPermissioned._beforeTokenTransfer()` independently checks every nonzero sender and recipient (`mTokenPermissioned.sol:22-32`). Therefore, a configuration can have:

- `vault.greenlistEnabled() == false`;
- The Fund greenlisted, allowing mGLOBAL to be minted to it; and.
- The `WrappedAsset` not greenlisted.

In that state, `create()` and `commit()` both pass because the Fund skips the role checks. Midas can then approve the deposit request and mint mGLOBAL to the Fund. However, `unlock()` calls `_wrapTo()`, and `WrappedAsset.mint()` pulls mGLOBAL from the Fund into the wrapper. The permissioned token rejects the non-greenlisted wrapper, so `unlock()` reverts.

The order remains in the dynamic `UNLOCKING` state with the mGLOBAL held by the Fund. The deposit recovery path explicitly rejects an approved deposit in `UNLOCKING` (`MidasFund.sol:480-486`), so completion requires Midas to greenlist the wrapper or an administrative upgrade/configuration repair. No attacker profit is established, so this is a Low settlement-liveness issue.

Proof of Concept:

1. Configure a permissioned mGLOBAL token and set `depositVault.greenlistEnabled(false)` .
2. Grant mGLOBAL's greenlisted role to the Fund, but not to the `WrappedAsset` .
3. Create and commit a normal deposit order. Both calls succeed because `_checkVaultAccess()` skips the role checks.
4. Process the Midas request so mGLOBAL is minted to the Fund.
5. Call `unlock(order)` .

The call reaches:

```
function _wrapTo(MidasFundStorage storage $, address _receiver, uint256 _amount)
↪   internal {
    address _wrappedShare = $.wrappedShare;
    $.mToken.safeApproveWithRetry(_wrappedShare, _amount);
    IWrappedAsset(_wrappedShare).mint(_receiver, _amount);
}
```

`WrappedAsset.mint()` attempts `mGLOBAL.transferFrom(Fund, WrappedAsset, amount)` . The permissioned transfer hook checks both addresses and reverts because the wrapper is not greenlisted. The order cannot use `recovering()` because its dynamic state is already `UNLOCKING` .

Recommendation: Make sure Midas will always whitelist the wrapped asset.

3F: Acknowledged, no code change: Midas has confirmed it will greenlist the WrappedAsset, and the MidasFund header already requires both the Fund and the WrappedAsset be greenlisted when the mToken is permissioned.

Cantina Managed: Acknowledged.

3.3 Informational

3.3.1 Small hardening and documentation nits

Severity: Informational.

Context: MorphoBorrowPosition.sol#L790-L853, LibBorrowOffers.sol#L140-L204, Retargetter.sol#L68-L76, PositionManagerLP.sol#L190-L235, IPositionManagerAdmin.sol#L95-L108, MorphoFlashLoanRequest.sol#L238-L250, IRetargetter.sol#L51-L60, IRetargetter.sol#L276-L291, Retargetter.sol#L631-L653, BorrowOffersRegistry.sol#L129-L156, LibBorrowOffers.sol#L140-L188, MorphoBorrowPosition.sol#L1136-L1164, Request.sol#L419-L422, OfferReceiver.sol#L57-L66, BorrowOffersRegistry.sol#L98-L106, BorrowOffersRegistry.sol#L213-L220, BorrowOffersRegistry.sol#L262-L269, OfferReceiver.sol#L129-L156.

Description: Minor API and documentation items that can confuse integrations or future maintenance:

1. `offerCount()` (`MorphoBorrowPosition.sol:947`) returns `popCount(liveBits)` (`LibBorrowOffers.sol:499`), so it also counts offers that expired but were not yet pruned or revoked, while `IBorrowOffers.sol:105` documents it as the number of currently-live offers.
2. `offer()` and `offers()` (`MorphoBorrowPosition.sol:952` , `MorphoBorrowPosition.sol:961`) return the same allocated-but-expired slots, and `IBorrowOffers.sol:108-113` describes the results as live offers.

3. `_alloc()` (`LibBorrowOffers.sol:149`) and `consume()` (`LibBorrowOffers.sol:197`) delete expired offers without emitting a per-offer pruning event, so indexers see slots vanish with no receipt.
4. `preLiquidate()` (`MorphoBorrowPosition.sol:344`) returns only seized collateral and repaid assets, so one call that consumes several offers exposes the per-offer identities through the `OfferConsumed` events (`LibBorrowOffers.sol:214`) alone.
5. `_QUOTER` is immutable (`Retargetter.sol:86`), so replacing the quoting math requires deploying a new Retargetter.
6. `setLtv` documents its parameter as `ltv_` (`IPositionManagerAdmin.sol:97`) while `setFeeData` names a different quantity `LTV_prev` (`IPositionManagerAdmin.sol:107`) in the same interface, with no note that the two LTVs are unrelated.
7. `onMorphoFlashLoan()` zeroes raw debt at `MorphoFlashLoanRequest.sol:259-260` before Morpho pulls the approved repayment, and the comment there does not state that a failed pull reverts the whole transaction.
8. Retargetter inherits Solady Multicallable at Retargetter.sol:71 and exposes multicall(bytes[]), but IRetargetter.sol:51-60 and :276-291 do not declare it. Integrations that generate an ABI from IRetargetter therefore miss a live external entry point and can build an incomplete route or wrapper.
9. Request.consume at Request.sol:419-422 accepts a partial fill and prorates the YT mint, but _validateOffer records the consumed offer nonce in the maker-wide nonce at OfferReceiver.sol:143. The consumed offer and every lower-nonce offer become invalid, so the unfilled remainder needs a fresh signature. The public NatSpec documents proration but not this single-use behavior.
10. `BorrowOffersRegistry.sol:268-269` stores `minOfferBonusBpsPlusOne` with a biased `+1` sentinel and decodes it with `storedBonus - 1`. This is harder to reason about than an explicit `exists` flag; using a boolean would make the configured-versus-default state unambiguous and avoid the `+1` / `-1` logic.
11. `initialize()` stores `depositVault` without emitting `DepositVaultUpdated` (MidasFund.sol#L195-L241). The factory's `FundCreated` event already exposes the initial vault binding, so this is event-consistency and indexer hardening rather than missing onchain observability.
12. `MidasFundStorage` could pack some small fields alongside addresses to reduce storage slots (MidasFund.sol#L121-L166). This is a gas/layout optimization only; because the Fund is upgradeable, any repacking must preserve the established ERC-7201 storage layout and prove that narrowed field widths cover all valid values.

Recommendation: Fix the above issues.

3F: Fixed in PR 217 (live-slot view semantics documented, `LTV_ref` rename, flash-loan debt-clear note, single-use-offer note, explicit `minOfferBonusSet` flag, `MidasFundStorage` packing, `DepositVaultUpdated` event); the remaining NatSpec nits are acknowledged.

Cantina Managed: Fix verified.

3.3.2 A revoked mint authorization permanently starts the Retargetter loan clock

Severity: Informational.

Context: Retargetter.sol#L230-L248, Retargetter.sol#L304-L312, LibStorage.sol#L149-L214, LibStorage.sol#L260-L276.

The owner or a `CONSUMER_ROLE` account creates and revokes Request mint authorizations through `Retargetter.authorizeMinting` (`Retargetter.sol:264`). A nonzero authorization routes into `checkOffer`, which calls `checkConsumptionWindow`; with `startedAt == 0` that stores `block.timestamp` (`LibStorage.sol:199`) before any broker transfers capital or mints PT/YT. Repayment prices every later yield-token obligation from that origin.

The same caller can then revoke with `authorizeMinting(to, 0, 0)`. The zero-amount path skips every gate (`Retargetter.sol:276`), so the operation can sit at zero PT supply, zero YT supply, and no pending authorizations while `startedAt` stays set. After `tickThreshold` elapses, consume and nonzero authorizations revert `ConsumptionWindowClosed` (`LibStorage.sol:200-202`). Nothing resets `startedAt` short of `clearOperation` (`LibStorage.sol:271`), which only runs at `resolve` (`Retargetter.sol:351`).

The empty operation can still be settled through `repay` (owed is zero, so it marks the Request repaid) and `resolve` , then restarted. The owner can also widen the live `tickThreshold` within the configured bounds, which changes tick-based repayment economics. Neither a permanent lock nor direct fund loss follows: the cost is a Consumer-triggered operational restart.

```
// src/manager/rebalancer/Retargetter.sol:264-283
function authorizeMinting(address to, uint128 ptAmount, uint128 ytAmount)
  external
  onlyOwnerOrRoles(CONSUMER_ROLE)
  nonReentrant
{
  RetargetterOperation storage operation_ = LibStorage.operationStorage();
  address request = operation_.checkRequest();
  // Replace semantics: drop the account from the set so the principal gate sizes
  ↪  the new
  // amounts as fresh capital; a full revocation (both amounts zero) skips every
  ↪  gate, since
  // it only shrinks exposure and must stay available once the window has closed
  EnumerableSetLib.AddressSet storage accounts = operation_.authorizedAccounts;
  accounts.remove(to);
  if (ptAmount != 0 || ytAmount != 0) {
    operation_.checkOffer(request, ptAmount, ytAmount, ptAmount,
    ↪  maxPrincipal(operation_.positionManager));
    // The capacity bound keeps every loop over the set within gas reach; see the
    ↪  constant
    accounts.add(to, MAX_AUTHORIZED_ACCOUNTS);
  }
  IRequest(request).authorizeMinting(to, ptAmount, ytAmount);
  emit MintingAuthorized(request, to, ptAmount, ytAmount);
}
```

```
// src/libs/manager/rebalancer/LibStorage.sol:190-203
function checkConsumptionWindow(RetargetterOperation storage self) internal {
  if (self.consumptionClosed) revert LibRetargetterErrors.ConsumptionWindowClosed();
  uint256 startedAt = self.startedAt;
  if (startedAt == 0) {
    if (block.timestamp + MIN_DEADLINE_BUFFER > self.repaymentDeadline) {
      revert LibRetargetterErrors.DeadlineTooClose();
    }
    // Safe: block.timestamp fits in uint40 for ~35,000 years
    // forge-lint: disable-next-line(unsafe-typecast)
    self.startedAt = uint40(block.timestamp);
  } else if (block.timestamp > startedAt + configStorage().tickThreshold) {
    revert LibRetargetterErrors.ConsumptionWindowClosed();
  }
}
```

Recommendation: Reset `startedAt` when PT/YT supplies and pending mint authorizations are all zero after a revocation. If the one-way clock is intentional, document that any nonzero authorization starts it and revocation does not rewind it.

3F: Fixed in PR 213: a full revocation that leaves the operation empty (zero PT and YT supply, no pending authorization) now rewinds `startedAt` to zero, reopening the consumption window.

Cantina Managed: Fix verified.

3.3.3 One-wei repayment flips a module into the management-fee basis for the whole checkpoint interval

Severity: Informational.

Context: LibView.sol#L65-L83, MorphoBorrowPosition.sol#L478-L494, PositionManagerBase.sol#L65-L223.

`LibView.totalAssets()` includes a borrow module's full quoted collateral in `totalCollateral` only when `collateral >= debt` (`LibView.sol:90-96`). `MorphoBorrowPosition.totalBorrowed()` floors borrow shares to assets with `toAssetsDown` (`MorphoBorrowPosition.sol:605`), so the comparison is a one-wei cliff.

A module one debt unit above its quoted collateral contributes zero to `currentCollat`. Anyone can repay that unit on Morpho, which moves the module into the fee basis. `PositionManagerBase._pendingFees()` then charges the management fee on the new `currentCollat` over `elapsed = block.timestamp - lastFeeAccrualTimestamp` (`PositionManagerBase.sol:213-217`), the full checkpoint interval, because storage keeps no per-module inclusion time. This is a fee-eligibility cliff, separate from Retargetter's prorated-YT ceiling.

```
// src/libs/manager/LibView.sol:85-97
address[] memory modules = ps.borrowModules.values();
uint256 modulesLength = modules.length;
for (uint256 i = 0; i < modulesLength; ++i) {
  uint256 collateral = IBorrowPosition(modules[i]).totalCollateralQuoted();
  uint256 debt = IBorrowPosition(modules[i]).totalBorrowed();
  if (collateral >= debt) {
    amount += collateral - debt;
    totalDebt += debt;
    totalCollateral += collateral;
  } else {
    hasBadDebt = true;
  }
}
```

```
// src/manager/base/PositionManagerBase.sol:213-217
if (fd.managementFee > 0) {
  uint256 elapsed = block.timestamp - _storage.lastFeeAccrualTimestamp;
  managementFeeAssets = currentCollat.mulDiv(fd.managementFee * elapsed, BPS *
  ↪  SECONDS_PER_YEAR);
  managementFeeAssets = managementFeeAssets.min(totalAssets_);
}
```

Example: Debt asset USDC (6 decimals), collateral wUSCC (6 decimals), quoted 1:1 for readability.

1. Healthy modules provide 10,000,000 USDC of quoted collateral (1e13).
2. An excluded module holds 5,000,000 wUSCC (5e12), quoted at 5,000,000 USDC, against 5,000,000.000001 USDC of debt, one base unit above its boundary.
3. One day passes (86,400s) at a 100 BPS annual management fee.
4. A caller repays 0.000001 USDC (1 base unit) on Morpho, so `collateral >= debt` holds again.
5. The next accrual runs on `currentCollat = 15,000,000 USDC` for the full day instead of 10,000,000: `15e12 * 100 * 86400 / (10000 * 365 days)` = 410,958,904 base units (410.958904 USDC) instead of 273,972,602 (273.972602 USDC).

The extra 136.986302 USDC reduces the remaining LPs' value and benefits the configured fee recipient.

Recommendation: Charge a module only from the first checkpoint where it is observed as fee-eligible. Alternatively, document that management fees use checkpoint-end eligibility for the full elapsed interval.

3F: Acknowledged and documented in PR 216: the one-wei eligibility cliff and its full-interval fee consequence are now stated on the fee-facing surfaces, since per-module inclusion timestamps would add storage for an edge case liquidation is expected to prevent.

Cantina Managed: Acknowledged.

3.3.4 Fund adapters assume exact-transfer assets without documenting it

Severity: Informational.

Context: IFund.sol#L48-L103, CentrifugeFund.sol#L147-L219, CentrifugeFund.sol#L421-L459.

Fund adapters move token amounts between the local contract and the venue with exact transfer semantics. `CentrifugeFund.commit` pulls `order.input` at `CentrifugeFund.sol:221` and then approves and requests that same `order.input` at `CentrifugeFund.sol:222-223`, so any shortfall in the received amount breaks the venue request. Supported assets therefore need transfer behavior matching adapter accounting, and neither `IFund` nor the deployment material states that requirement.

Native-decimal asset amounts differ from base-18 share or quote units, and wrapper conversions can leave a bounded raw-atom remainder. Exact-transfer tokens bound that remainder. Fee-on-transfer or rebasing behavior invalidates the unit invariant and can cause loss beyond the bounded residue.

Scope: this is about the missing documented assumption, no unsupported token or transfer-tax token is currently admitted or deployed.

```
// src/funds/centrifuge/CentrifugeFund.sol:218-224
if (order.mode == Mode.DEPOSIT) {
  // Pull asset from depositor, approve vault, request deposit
  address _asset = $.asset;
  _asset.safeTransferFrom(msg.sender, address(this), order.input);
  _asset.safeApproveWithRetry(_vault, order.input);
  ICentrifugeVault(_vault).requestDeposit(order.input, address(this),
  ↪  address(this));
  _asset.safeApproveWithRetry(_vault, 0);
```

Recommendation: Document exact-transfer semantics and unit boundaries in `IFund`, adapter NatSpec, and deployment materials. Alternatively, publish a supported-asset list that excludes fee-on-transfer, rebasing, and callback tokens.

3F: Fixed in PR 213: the exact-transfer requirement is stated on `IFund` per transfer, binding every current and future adapter (fee-on-transfer, rebasing and callback tokens stay out of scope).

Cantina Managed: Fix verified.

3.3.5 Facility balance snapshots credit `PositionManager` fee-mint shares to the resolving intent

Severity: Informational.

Context: FacilityPositionManager.sol#L31-L134, FacilityPositionManager.sol#L191-L202, PositionManagerBase.sol#L65-L223.

`depositManager`, `withdrawManager`, and `burnManager` snapshot the Facility's collateral, debt, and PositionManager-share balances in `_initialPmParameters` (`FacilityPositionManager.sol:192-194`), then `_commitSnapshots` assigns each full balance delta to the selected intent (`FacilityPositionManager.sol:206-217`, `LibIntent.sol:342-361`).

```
// src/facility/base/FacilityPositionManager.sol:191-194
// take snapshots before the operation
collateralSnapshot = LibIntent.takeBalanceSnapshot(collateralAsset);
debtSnapshot = LibIntent.takeBalanceSnapshot(debtAsset);
sharesSnapshot = LibIntent.takeBalanceSnapshot(positionManager);
```

Every `IPositionManager` flow calls `_accrueFees()` before share balances move (`PositionManagerLP.sol:53`, `:99`, `:136`). With pending fees, `_accrueFees()` mints shares to `feeData.feeRecipient` (`PositionManagerBase.sol:292-296`). When that recipient is the Facility, the observed share delta is the operational delta plus the fee mint, and `commitBalanceSnapshot()` records the full signed net change against the selected intent. Deposits produce a positive operational delta; withdrawals and burns produce a negative one. In the deposit path the intent's LPs can claim the fee shares after resolution.

```
// src/manager/base/PositionManagerBase.sol:289-296
uint256 feeShares = managementFeeShares + performanceFeeShares;

// Mint fee shares
if (feeShares > 0) {
  address feeRecipient = LibStorage.positionManagerStorage().feeData.feeRecipient;
  _mint(feeRecipient, feeShares);
  emit IPositionManagerLP.FeesAccrued(feeRecipient, feeShares);
}
```

Preconditions: `feeRecipient == Facility`, pending fee shares, a resolving intent using that PositionManager, and a normal `FACILITATOR_ROLE` manager operation. A fixed-block check of 25 PositionManagers found external Safe recipients; that check did not establish current fleet exposure.

No extra shares are minted and no collateral moves. Pending fee shares shift from protocol fee custody to one intent's LPs, bounded by the fees pending at the checkpoint, repeatable while the configuration stands. Share conversion floors and fee rounding leave a bounded remainder on top of that.

Example: USDC PositionManager (`virtualShareOffset = 1e12`), `totalAssets = 1,000,000 USDC` (`1e12`), `totalSupply = 1e24` shares, `managementFee = 200` BPS (the `MAX_MANAGEMENT_FEE` cap) with one year elapsed and `performanceFee = 0`, assuming supply still sits at the `1e12` shares-per-USDC-unit parity a vault starts at.

1. `_pendingFees()` charges 20,000 USDC; `convertToShares(2e10, 1e24, 1e12 - 2e10, 1e12, false)` mints `20,408,163,265,305,705,955,851` shares (~20,408.16 shares), worth 20,000 USDC once the mint lands.
2. `depositManager` snapshots the share balance before the operation, so the recorded delta is the deposit's operational shares plus that fee mint.
3. The intent's LPs claim the extra ~20,408.16 shares, 20,000 USDC of fee value, at resolution.

Recommendation: Attribute operational share deltas separately from fee mints so `commitBalanceSnapshot()` credits only operation shares to the resolving intent. Alternatively, document that snapshots absorb pending fee shares when `feeRecipient == Facility`.

3F: Fixed in PR 215 on both sides: `_commitSnapshots` states the share delta includes fee shares minted to the Facility, and `setFeeData`'s `feeRecipient` states a Facility-operated PositionManager must never set its recipient to that Facility.

Cantina Managed: Fix verified.

3.3.6 Zero-share fee accrual is documented only in internal comments, not on the `feeData` interface

Severity: Informational.

Context: PositionManagerBase.sol#L65-L114, PositionManager.sol#L176-L205, IPositionManager.sol#L79-L98.

`IPositionManager.feeData()` documents each of the six returned fields, and `lastDebt()` documents the zero bootstrap sentinel (`IPositionManager.sol:91-98`). What the interface leaves out is the zero-share accrual case: `_pendingFees` returns zero management and performance shares whenever `totalFeeAssets >= totalAssets_` (`PositionManagerBase.sol:245-255`), and `convertToShares` rounds down, so an accrual can mint nothing while `heldManagementFees` still carries assets. That behavior is explained only in the internal comment block at `PositionManagerBase.sol:232-244`, which integrators reading the interface do not see.

```
// src/manager/base/PositionManagerBase.sol:245-255
if (totalFeeAssets >= totalAssets_) {
  return (
    totalAssets_,
    totalSupply_,
    currentDebt,
    0,
    0,
    advanceReference,
    advanceReference ? 0 : _storage.heldManagementFeeAssets
  );
}
```

Recommendation: Document the zero-share outcome and the held-fee lifecycle on `IPositionManager.feeData()` / `pendingFees()` : when the mint is skipped, that `lastFeeAccrualTimestamp` and the reference still move, and that a zero share result does not mean the held-fee accumulator is empty.

3F: Fixed in PR 215: `previewAccrueFees` now documents that zero returned shares do not prove empty fee state and that the timestamp, reference and held fees still follow their advance/hold rules on accrual.

Cantina Managed: Fix verified.

3.3.7 Held management fee accumulator tracks charged assets, not minted shares, and this is undocumented

Severity: Informational.

Context: PositionManagerAdmin.sol#L158-L196, PositionManagerBase.sol#L65-L114, IPositionManagerAdmin.sol#L100-L130.

Both fee transitions accrue under the previous terms before writing new state: `setFeeData` calls `_accrueFees()` at `PositionManagerAdmin.sol:164` before storing the new `FeeData` , and `resetPerformanceReference` calls it at `PositionManagerAdmin.sol:187` before moving the reference and zeroing `heldManagementFeeAssets` . That ordering is covered by the interface NatSpec at `IPositionManagerAdmin.sol:101-103` and `IPositionManagerAdmin.sol:112-129` .

What the NatSpec does not state is that the accumulator is denominated in charged assets rather than minted shares. `PositionManagerBase.sol:229` adds the whole interval's `managementFeeAssets` to `heldManagementFeeAssets` while the reference is held, but the share conversion at `PositionManagerBase.sol:260-267` rounds down, so an interval can mint zero shares to the fee recipient and still add its asset amount to the accumulator. That amount is deducted from the next positive performance basis at `PositionManagerBase.sol:223-224` , and it is rescaled with share supply at `LibStorage.sol:286-288` , so it needs to stay distinct from fee-recipient entitlement and from the performance deduction itself. The remainder is bounded by rounding.

```
// src/manager/base/PositionManagerBase.sol:219-229
// Performance fee on the levered-slice basis, net of the management fees charged
↪  since the
// reference last advanced: the held accumulator plus the current interval's charge.
heldManagementFees_ = _storage.heldManagementFeeAssets;
totalFeeAssets = managementFeeAssets;
if (fd.performanceFee > 0 && basis > managementFeeAssets + heldManagementFees_) {
  totalFeeAssets += (basis - managementFeeAssets -
  ↪  heldManagementFees_).mulDiv(fd.performanceFee, BPS);
}
// While the reference is held, the current interval's management fee joins the
↪  accumulator
// so the next crystallization deducts it; on advance the pending deduction is
↪  consumed (or,
// for any excess above the basis, forgiven) and the accumulator restarts.
heldManagementFees_ = advanceReference ? 0 : heldManagementFees_ +
↪  managementFeeAssets;
```

Recommendation: Document on `heldManagementFeeAssets` (`LibStorage.sol:97`) and in the `setFeeData` / `resetPerformanceReference` NatSpec that the accumulator records charged assets independently of whether the interval minted fee shares, and that a floored interval still reduces the next crystallization.

3F: Fixed in PR 215: the `feeData` NatSpec now states `heldManagementFees` is asset-denominated, can be non-zero with no fee shares pending, and enumerates the only events that consume or clear it.

Cantina Managed: Fix verified.

3.3.8 Retargetter's flash-loan callback binds neither the payload nor the delivered principal

Severity: Informational.

Context: Retargetter.sol#L468-L530.

`startSyncRetargetting` checks the module whitelist, stores the module in `WINDOW_TSLOT` / `MODULE_TSLOT` and the nominal loan amount in `AMOUNT_TSLOT`, then calls `IFlashLoanModule.flashLoan` (`Retargetter.sol:536-551`). Step authority for the whole window belongs to that module address.

`onFlashLoan` authenticates the caller and the nominal amount, then decodes the module-supplied `data` and delegatecalls it through `_multicall` (`Retargetter.sol:576-593`). The payload is not bound to the `data` passed to `startSyncRetargetting`, and the delivered principal is never checked against `AMOUNT_TSLOT` before execution. A module can deliver zero or partial funds, run a different `rebalance` payload, borrow the missing amount through PositionManager, and still receive the approval for the nominal repayment at `Retargetter.sol:592`.

```
// src/manager/rebalancer/Retargetter.sol:576-593
function onFlashLoan(uint256 amount, bytes calldata data) external {
  address module = MODULE_TSLOT.tLoadAddress();
  if (module == address(0) || msg.sender != module || amount !=
  ↪  AMOUNT_TSLOT.tLoadUint()) {
    revert LibRetargetterErrors.UnauthorizedFlashLoanCallback();
  }
  MODULE_TSLOT.tStoreAddress(address(0));
  // `data` is the abi.encode of the step calls built by startSyncRetargetting and
  ↪  forwarded
  // verbatim by the module; point a calldata array at it in place instead of
  ↪  copying it
  bytes[] calldata calls;
  assembly ("memory-safe") {
    let arrayOffset := add(data.offset, calldataload(data.offset))
```

```
    calls.offset := add(arrayOffset, 0x20)
    calls.length := calldataload(arrayOffset)
  }
  _multicall(calls);
  // The module pulls its repayment through this allowance after the callback
  ↪  returns
  LibStorage.assetsStorage().debtAsset.safeApproveWithRetry(module, amount);
}
```

An arbitrary external caller cannot forge the callback: `startSyncRetargetting` is `onlyOwnerOrRebalancer` and the module slot is single-shot. The in-repo `MorphoFlashLoanAdapter` transfers the full principal to the initiator and forwards the payload verbatim (`MorphoFlashLoanAdapter.sol:85-96`), so this needs a different whitelisted module.

Example: Assume USDC debt / wUSCC collateral (both 6 decimals), PositionManager NAV `totalAssets` of 1,000,000 USDC (1e12), `maxRebalanceLoss == 50` bps, and a collateral quote unchanged across the window.

1. Module opens a window for a nominal 1,000,000 USDC (1e12) loan and delivers zero, then supplies its own payload.
2. Payload borrows through PositionManager, so a 12,000 USDC (1.2e10) module profit lands as an equal debt increase and an equal NAV loss.
3. With `rebalanceCooldown == 0`, one aggregate 12,000 USDC call reverts at 120 bps, while three 4,000 USDC (4e9) calls pass at 40, 40.2, and 40.3 bps against the shrinking NAV.
4. A nonzero cooldown blocks the per-call slicing but leaves the payload, delivered principal, and execution window unbound.

Recommendation: Document that the flash-loan module is trusted and outside the compromised-module threat model.

3F: Fixed in PR 209: `onFlashLoan` now requires the incoming data to hash to the stored payload digest and the debt-asset balance to grow by the nominal amount before any step runs, with step authority scoped to the committed payload only.

Cantina Managed: Fix verified.

3.3.9 Standing `BorrowOffers` have no cap on liquidation bonus, seized equity, or close factor

Severity: Informational.

Context: MorphoBorrowPosition.sol#L255-L329, MorphoBorrowPosition.sol#L722-L756, LibBorrowOffers.sol#L287-L448, BorrowOffersRegistry.sol#L129-L156.

`proposeOffer` is gated by `OFFERS_REGISTRY.checkCanCreateOffer` (`MorphoBorrowPosition.sol:863`), so any `PROPOSER` (set by the registry owner via `BorrowOffersRegistry.setProposer`) can create a standing offer that repays debt and transfers nearly all position collateral to whoever fills it. The position owner has no bypass and no veto; revocation sits with guardians and the registry owner.

Both the proposal-time filter (`MorphoBorrowPosition.sol:898-908`) and the per-fill gate check profitability against a *minimum* bonus floor plus a strict LTV decrease. Neither bounds the bonus from above, the seized equity, or the close factor. The consume walk is limited only by the position's remaining collateral and shares (`LibBorrowOffers.sol:310-337`).

```
// src/libs/borrow/LibBorrowOffers.sol:436-444
uint256 seizedValue = fillCollateral.mulDiv(inp.price, ORACLE_PRICE_SCALE);
uint256 repaidDebtValue = fillShares.toAssetsUp(inp.totalBorrowAssets,
↪  inp.totalBorrowShares);
```

```
if (!isProfitableAboveBonusFloor(seizedValue, repaidDebtValue,
↪  inp.minOfferBonusBps)) return FillAction.Skip;

uint256 remainingDebtValue =
↪  remainingPositionShares.toAssetsUp(inp.totalBorrowAssets,
↪  inp.totalBorrowShares);
uint256 remainingCollateralValue = remainingPositionCollateral.mulDiv(inp.price,
↪  ORACLE_PRICE_SCALE);
if (!strictlyLowersLtv(seizedValue, repaidDebtValue, remainingDebtValue,
↪  remainingCollateralValue)) {
  return FillAction.Stop;
}
```

A proposer can therefore price an offer close to full collateral value. Once the veto window (`activeAt`) passes, any public liquidator can repay and fill it: `test/borrow/MorphoDormantOfferActivation.t.sol` fills an offer sized at 90% of position shares and asserts the seizure exceeds 89% of position collateral. A former registry owner who retains `PROPOSER` after handoff has the same authority and can capture all but one raw unit of another manager's equity; that stale-handoff case changes the actor, not the loss ceiling, and is outside this rating.

Recommendation: Cap the proposer-authorized liquidation bonus, seized equity, and close factor in the standing-offer fill path. If proposers are meant to be fully trusted administrators, document full liquidation as approved emergency authority instead.

3F: Acknowledged, no code change: an uncapped bonus is intended so a lender can price an offer high enough to attract a filler when 3F is no longer live, with semi-trusted proposers and guardian revocation as the controls.

Cantina Managed: Acknowledged.

3.3.10 Offer-walk `Stop` / `Skip` semantics and raw-unit rounding are not documented on the external liquidation surface

Severity: Informational.

Context: LibBorrowOffers.sol#L140-L262, LibBorrowOffers.sol#L287-L408, MorphoBorrowPosition.sol#L255-L329.

The offer walk converts between collateral units, debt shares, and oracle-scaled values with mixed rounding: `fillShares` rounds up (`LibBorrowOffers.sol:391`), the repaidShares-mode collateral cap and the position-clamp rescale round down (`LibBorrowOffers.sol:381` , `LibBorrowOffers.sol:404`), and `_priceAction` rounds collateral value down and debt value up (`LibBorrowOffers.sol:436-441`). One-atom boundaries therefore decide whether an offer yields `Consume` , `Skip` , or `Stop` . `Skip` moves to the next offer, `Stop` ends the walk (`LibBorrowOffers.sol:318-319`), so the same caller-selected partial target can land on a different tail of the offer book and produce different final seized/repaid totals.

These rules live only in internal `LibBorrowOffers` natspec. The external surface callers actually use, `preLiquidate` and `previewConsume` , does not state them: the `preLiquidate` natspec covers the proportional-path bonus math, the Morpho health adjustment, the callback, and the LTV dispatch (`MorphoBorrowPosition.sol:260-334`), but says nothing about per-offer rounding directions or `Stop` versus `Skip` . Integrations and monitoring cannot interpret a partial offer liquidation from the public docs alone.

```
// src/libs/borrow/LibBorrowOffers.sol:315-319
(FillAction action, uint256 fillCollateral, uint256 fillShares) =
  _computeFill(inp, walkOffer.remainingCollateral, walkOffer.remainingDebtShares,
  ↪  totalSeized, totalDebtShares);
```

```
if (action == FillAction.Stop) break;
if (action == FillAction.Skip) continue;
```

Recommendation: Document the raw-unit rounding directions and the exact `Stop` and `Skip` conditions on the external liquidation surface (`preLiquidate` / `previewConsume` natspec plus integrator docs), so callers interpret partial liquidation results consistently.

3F: Fixed in PR 215: `previewConsume` now documents the skip conditions, every walk-halting condition and the raw-unit rounding directions, and `preLiquidate` points at that note and states the strict LTV-reduction revert.

Cantina Managed: Fix verified.

3.3.11 Minimum tick is payable out of `PositionManager` while committed Request principal stays unused

Severity: Informational.

Context: Retargetter.sol#L253-L285, RetargetterQuoter.sol#L117-L129.

`Retargetter.repay()` sizes the transfer as `owed - requestBalance` (`Retargetter.sol:314-315`), so an operation that consumed an offer but never called `pullRequestFunds` still leaves the full principal sitting in the Request and only tops up the yield leg from PositionManager.

`RetargetterQuoter.paidDuration` floors at one tick (`RetargetterQuoter.sol:188-189`), so even same-block settlement owes a full tick of YT. The maker later redeems PT and YT for principal plus that tick, and PositionManager LPs eat the tick plus the borrow cost for capital that never left the Request.

```
// src/manager/rebalancer/Retargetter.sol:312-318
owedAmount = _owed(operation_);
address debtAsset = LibStorage.assetsStorage().debtAsset;
uint256 requestBalance = debtAsset.balanceOf(request);
uint256 shortfall = owedAmount > requestBalance ? owedAmount - requestBalance : 0;
if (shortfall > 0) {
  debtAsset.safeApproveWithRetry(request, shortfall);
  IRequestInteractions(request).repay(shortfall);
```

```
// src/manager/rebalancer/RetargetterQuoter.sol:187-190
if (tickDuration == 0 || tickThreshold >= tickDuration) revert
↪  LibRetargetterErrors.InvalidParameters();
uint256 paidTicks = (elapsed + tickDuration - tickThreshold) / tickDuration;
if (paidTicks == 0) paidTicks = 1;
duration = paidTicks * tickDuration;
```

Example: Config: `horizon = 90 days`, `tickDuration = 1 days`.

1. Consumer consumes a maker offer for `amount = 1,000,000.000000` USDC principal and `expectedReturn = 12,000.000000` USDC yield. The Request holds 1,000,000 USDC; the maker holds 1,000,000 PT and 12,000 YT.
2. Rebalancer skips `pullRequestFunds`, so no wUSCC collateral is ever bought and the USDC stays put.
3. Rebalancer calls `repay()` immediately. `elapsed = 0` gives `paidTicks = 1`, `duration = 1 days`, so `owed = 1,000,000e6 + ceil(12,000e6 * 1 / 90) = 1,000,000e6 + 133,333,334`.
4. `requestBalance = 1,000,000e6`, so `shortfall = 133.333334` USDC, borrowed from PositionManager.

5. Maker burns PT + YT for 1,000,000.133334 USDC. PositionManager LPs are down 133.333334 USDC plus borrow interest.

Recommendation: Document that `PositionManager` LPs can fund the minimum tick while Request principal stays unused, or skip the tick floor when no principal was pulled.

3F: Fixed in PR 213, no behavior change: an operation that consumes an offer without pulling the funds still owes a full tick of YT out of position-derived funds.

Cantina Managed: Fix verified.

3.3.12 `remediationDelta` counts expected collateral yield twice

Severity: Informational.

Context: RetargetterQuoter.sol#L78-L104, RetargetterQuoter.sol#L132-L155.

`ltvDownPrincipal` sizes `collateralToFreeQuoted = principal * (1 + Yr) / (1 + Yc)`, so the freed wUSCC already has the expected collateral yield `Yc` discounted out of it (`RetargetterQuoter.sol:172-174`). `remediationDelta` then applies the raw price ratio `rho = p1 / p0` to the full repayment (`RetargetterQuoter.sol:215-219`). When the only price movement is the expected collateral yield, `rho = 1 + Yc`, the redemption proceeds equal the repayment and the true mismatch is zero, but the helper reports `repayment * Yc`.

```
// src/manager/rebalancer/RetargetterQuoter.sol:172-174
collateralToFreeQuoted = principal.fullMulDivUp(
  WAD + _scaledRate(requestYieldRate, duration), WAD +
   ↪  _scaledRate(collateralYieldRate, duration)
);
```

```
// src/manager/rebalancer/RetargetterQuoter.sol:214-219
// The fixed bridge repayment the redemption proceeds are measured against
uint256 repayment = principal.fullMulDiv(WAD + _scaledRate(requestYieldRate,
 ↪  duration), WAD);
uint256 magnitude;
if (priceDriftWad >= WAD) {
  // Surplus: proceeds exceed the repayment; rounds down so the fold never
   ↪  overshoots
  magnitude = repayment.fullMulDiv(priceDriftWad - WAD, WAD);
```

Selector `0x114d0641` has no consumer in this repo, and no state-changing use that could misallocate funds was identified. An off-chain caller sizing a remediation or showing a quote gets the wrong surplus.

Example:

1. `requestYieldRate = collateralYieldRate = 5e16` (5% per 365 days), `duration = 30 days`, so `Yr = Yc = 4109589041095890` (0.41096%).
2. `principal = 1_000_000e6` USDC. `ltvDownPrincipal` frees `collateralToFreeQuoted = 1_000_000e6` (1,000,000 wUSCC quoted in USDC).
3. Repayment is `1_000_000e6 * (1 + Yr) = 1_004_109.589041e6` USDC.
4. Price moves exactly by the collateral yield: `priceDriftWad = 1.004109589041095890e18`. Proceeds are `1_000_000e6 * (1 + Yc) = 1_004_109.589041e6` USDC, so the true delta is 0.
5. `remediationDelta` returns `+4_126.477763e6`, i.e. a phantom 4,126.47 USDC surplus per 1,000,000 USDC of principal.

Recommendation: Document `remediationDelta` as an advisory raw-price quote, not an exact settlement value or a state-changing cap, and require callers to treat it that way.

3F: Fixed in PR 208: `remediationDelta` now takes the `collateralYieldRate` the operation was sized with and pivots the surplus/shortfall split at the expected growth `1 + Yc` instead of unity.

Cantina Managed: Fix verified.

3.3.13 Rebalance cooldown is enforced per call, so a `SYNC` payload with two `PositionManager` rebalances always reverts

Severity: Informational.

Context: PositionManagerRebalancing.sol#L45-L119, Retargetter.sol#L468-L530.

Description: `Retargetter.startSyncRetargetting` runs its whole payload inside one flash-loan callback, so every inner call shares a block timestamp. `PositionManagerRebalancing.rebalance` checks the cooldown once per call (`PositionManagerRebalancing.sol:62-74`) and writes `lastRebalanceTimestamp = uint40(block.timestamp)` on success (`PositionManagerRebalancing.sol:106`). Multiple operations inside one `rebalance()` call share one check; multiple calls each get their own.

With `rebalanceCooldown > 0`, the second `PositionManager.rebalance()` in a SYNC payload sees zero elapsed time, reverts with `RebalanceCooldownNotElapsed`, and rolls back the flash-loan transaction. A payload with one rebalance succeeds. No supported production payload has been shown to need two.

A prior successful rebalance also blocking the first call of a later SYNC is specified cooldown behavior, separate from this.

```
// src/manager/base/PositionManagerRebalancing.sol:59-74
// Enforce cooldown between consecutive rebalance calls
// (block-scoped so the temporaries do not deepen the stack for the rest of the
↪  function)
{
  uint40 cooldown = _storage.rebalanceConfig.rebalanceCooldown;
  uint40 lastRebalance = _storage.rebalanceConfig.lastRebalanceTimestamp;
  if (cooldown > 0 && lastRebalance > 0) {
    // Safe: block.timestamp fits in uint40 for ~35,000 years
    // Subtraction is safe because block.timestamp >= lastRebalance (time is
    ↪  monotonic).
    // Using `elapsed < cooldown` instead of `timestamp < lastRebalance + cooldown`
    // to avoid uint40 overflow when cooldown is large.
    // forge-lint: disable-next-line(unsafe-typecast)
    if (uint40(block.timestamp) - lastRebalance < cooldown) {
      revert LibManagerErrors.RebalanceCooldownNotElapsed();
    }
  }
}
```

Recommendation: Scope the cooldown to the flash-loan callback rather than to each nested `PositionManager.rebalance()` call, so a multi-call payload takes one check. If the per-call semantics stay, document that compatible legs must be packed into a single `rebalance()` call and that routes needing an intervening Fund action have to run as separate transactions.

3F: Acknowledged and documented on `startSyncRetargetting` in PR 213: the cooldown is per rebalance call, so compatible legs must be packed into a single step (scoping the cooldown to the flash-loan window was rejected as it inverts the trust direction).

Cantina Managed: Acknowledged.

3.3.14 Rebalance post-check reverts on modules the payload never touched

Severity: Informational.

Context: Retargetter.sol#L392-L455.

Description: `Retargetter.rebalance()` snapshots every module LTV before calling the PositionManager and re-checks every module after, including modules absent from the payload (`Retargetter.sol:500-507`). A module ending above `target` must have strictly decreased. An untouched module sitting above `target` has `moduleLtvAfter == moduleLtvsBefore[i]`, so it reverts with `PositionAboveTarget`, taking the valid legs on the other modules with it. Direct owner calls skip the block (`Retargetter.sol:495`); rebalancer calls and anything inside a flash-loan window do not.

```
// src/manager/rebalancer/Retargetter.sol:495-508
if (msg.sender != owner()) {
  uint256 ltvAfter = _positionManagerLtv(positionManager);
  if (ltvAfter > target && ltvAfter >= ltvBefore) {
    revert LibRetargetterErrors.AboveTargetLtv(ltvAfter, ltvBefore, target);
  }
  for (uint256 i = 0; i < modulesLength; ++i) {
    address module = modules[i];
    uint256 moduleLtvAfter = _moduleLtv(module);
    if (moduleLtvAfter == type(uint256).max) revert
    ↪  LibRetargetterErrors.BadDebtPosition(module);
    if (moduleLtvAfter > target && moduleLtvAfter >= moduleLtvsBefore[i]) {
      revert LibRetargetterErrors.PositionAboveTarget(module);
    }
  }
}
```

Near the target, floors can flatten a one-atom improvement into an equal value and trip the same guard. That is the same liveness boundary, not a loss path.

Example:

1. Target is 70%. Module A holds wUSCC worth 100,000 USDC against 71,000 USDC debt (71% LTV). Module B holds wUSCC worth 100,000 USDC against 75,000 USDC debt (75% LTV).
2. The rebalancer submits a payload repaying 10,000 USDC on module B only, taking it to 65,000/100,000 = 65%.
3. The post-check reads module A at 71% after and 71% before: above target, not strictly decreasing, so `PositionAboveTarget(A)` fires.
4. The 10,000 USDC repayment on module B reverts with it. Aggregate LTV drops from 73% to 68%, and the call still fails.

Recommendation: Skip the per-module direction check for modules the payload does not touch, or require only that an untouched module does not increase.

3F: Fixed in PR 209: the per-module post-check was relaxed from `>=` to `>`, so an untouched module sitting above target no longer reverts the payload and only a worsened module does.

Cantina Managed: Fix verified.

3.3.15 `pullRequestFunds(0)` is an undocumented close of the Retargetter funding round

Severity: Informational.

Context: Retargetter.sol#L253-L262, LibStorage.sol#L201-L214.

Description: `Retargetter.pullRequestFunds` calls `closeConsumption` at `Retargetter.sol:292`, before it resolves or transfers `amount`. `closeConsumption` sets `consumptionClosed` and revokes every pending mint authorization on the Request (`LibStorage.sol:211-219`). Calling it with `amount == 0` moves no capital and still ends the funding round.

After that call `consume` and nonzero `authorizeMinting` revert with `ConsumptionWindowClosed`; zero-amount revocations still pass. Signed offers stay stored but are unusable for the operation, and funding resumes only after the operation settles and restarts or is abandoned and cleared. The owner and `REBALANCER_ROLE` are authorized for this transition, but no separate entry point or documentation names it.

```
// src/manager/rebalancer/Retargetter.sol:287-296
function pullRequestFunds(uint256 amount) external onlyOwnerOrRebalancer
↪  nonReentrant {
  RetargetterOperation storage operation_ = LibStorage.operationStorage();
  address request = operation_.checkRequest();
  // Pulling is the point of no return for the funding round: capital entry shuts
  ↪  and every
  // pending authorization is revoked, so funds being deployed can no longer be
  ↪  diluted
  operation_.closeConsumption(request);
  if (amount == FULL_BALANCE_SENTINEL) amount =
  ↪  LibStorage.assetsStorage().debtAsset.balanceOf(request);
  IRequestInteractions(request).pullFunds(amount, "");
  emit RequestFundsPulled(request, amount);
}
```

```
// src/libs/manager/rebalancer/LibStorage.sol:211-219
function closeConsumption(RetargetterOperation storage self, address request)
↪  internal {
  self.consumptionClosed = true;
  EnumerableSetLib.AddressSet storage accounts = self.authorizedAccounts;
  for (uint256 remaining = accounts.length(); remaining > 0; --remaining) {
    address account = accounts.at(remaining - 1);
    IRequest(request).authorizeMinting(account, 0, 0);
    accounts.remove(account);
  }
}
```

Recommendation: Add an explicit `closeFundingRound()` entry point and reserve `pullRequestFunds` for transfers. If the overload stays, document `pullRequestFunds(0)` as the close transition, its authorized roles, that no value moves, and that existing Request funds remain pullable.

3F: Acknowledged and documented in PR 213: the zero-amount close is stated as the named funding-round transition in the `pullRequestFunds` NatSpec (a dedicated `closeFundingRound()` was rejected as duplicative).

Cantina Managed: Acknowledged.

3.3.16 Retargetter's one-operation-at-a-time guard is per instance, so two Retargetters can each admit the full principal cap

Severity: Informational.

Context: Retargetter.sol#L803-L829, PositionManagerRebalancing.sol#L55-L64.

Description: `_checkStart` enforces one live operation per Retargetter using that instance's own storage, and `maxPrincipal(positionManager)` (`Retargetter.sol:688-712`) derives the cap from

the PositionManager's current `collateralAmountQuoted()` and `debtAmount()`. Neither reads principal already admitted by a different Retargetter.

If the owner grants `REBALANCER_ROLE` on the same PositionManager to two Retargetters, both can pass `_checkStart` at the full cap before either operation moves PositionManager state, so combined admitted principal exceeds the intended single-operation capacity.

```
// src/manager/rebalancer/Retargetter.sol:791-800
function _checkStart(RetargetterOperation storage operation_, address
↪  positionManager, address fund, uint256 amount)
  internal
  view
{
  if (operation_.positionManager != address(0)) revert
  ↪  LibRetargetterErrors.OperationActive();
  if (operation_.orderLive) revert LibRetargetterErrors.OrderActive();
  _checkPair(positionManager);
  if (!LibStorage.whitelistsStorage().funds[fund]) revert
  ↪  LibRetargetterErrors.FundNotWhitelisted();
  if (amount > maxPrincipal(positionManager)) revert
  ↪  LibRetargetterErrors.PrincipalCapExceeded();
}
```

Recommendation: Document that at most one Retargetter may hold `REBALANCER_ROLE` on a given `PositionManager`, and that the exclusivity guarantee is per Retargetter instance rather than per `PositionManager`.

3F: Acknowledged and documented on the contract header and factory checklist in PR 213: the guard is per instance, so a `PositionManager`'s rebalancer role must be granted to at most one Retargetter.

Cantina Managed: Fix verified.

3.3.17 Splitting an exit across many transactions rounds `heldManagementFeeAssets` down repeatedly

Severity: Informational.

Context: LibStorage.sol#L174-L196.

Description: `rebaseSnapshot` rescales the pending management-fee deduction by `newSupply / prevSupply` on every supply change, flooring each time (`LibStorage.sol:288`). Splitting one exit into N exits applies N floors instead of one, so the surviving deduction is up to N-1 base units (1e-6 USDC) below the aggregate result.

`_pendingFees` subtracts that deduction from the performance basis (`PositionManagerBase.sol:223-224`), so a smaller deduction means a larger performance-fee mint on the next crystallization.

```
// src/libs/manager/LibStorage.sol:285-290
if (newSupply != prevSupply && prevCollat > 0) {
  uint256 heldManagementFeeAssets = self.heldManagementFeeAssets;
  if (heldManagementFeeAssets > 0) {
    self.heldManagementFeeAssets = heldManagementFeeAssets.mulDiv(newSupply,
    ↪  prevSupply);
  }
}
```

Example:

1. `heldManagementFeeAssets = 5_000_123456` (5,000.123456 USDC), share supply `10_000e18`.

2. One exit burning `100e18` shares: `floor(5_000_123456 * 9_900e18 / 10_000e18) = 4_950_122_221` (4,950.122221 USDC).
3. Instead, 100 exits of `1e18` shares each, flooring after each: `4_950_122_156` (4,950.122156 USDC). The next positive basis charges the performance fee on that extra 0.000065 USDC.

Recommendation: Document the loss, as it's dust level.

3F: Fixed in PR 214: the supply-ratio rescaling was removed entirely, so no per-flow flooring remains to accumulate, and PR 215 pins the nominal-across-flows property with a regression test.

Cantina Managed: Fix verified.

3.3.18 Partial fills can leave all offer slots live but unusable

Severity: Informational.

Context: LibBorrowOffers.sol#L91-L142, LibBorrowOffers.sol#L160-L188, LibBorrowOffers.sol#L273-L368.

Description: `LibBorrowOffers.consume()` clears an offer only when its remaining collateral or remaining debt shares reach zero (`LibBorrowOffers.sol:205`). A partial fill can leave both values nonzero even when the remaining ratio no longer passes the profitability, bonus-floor or de-risking checks in `_priceAction` (`LibBorrowOffers.sol:436-446`).

```
// src/libs/borrow/LibBorrowOffers.sol:202-216
for (uint256 i; i < walkLength; ++i) {
  WalkOffer memory walkOffer = offers[i];
  if (walkOffer.filledCollateral == 0) continue; // untouched by the walk
  bool exhausted = (walkOffer.remainingCollateral == 0 ||
  ↪  walkOffer.remainingDebtShares == 0);
  if (exhausted) {
    clearBits |= uint256(1) << walkOffer.id;
    delete s.slab[walkOffer.id];
  } else {
    Offer storage offer = s.slab[walkOffer.id];
    offer.remainingCollateral = walkOffer.remainingCollateral;
    offer.remainingDebtShares = walkOffer.remainingDebtShares;
  }
  emit IBorrowOffers.OfferConsumed(walkOffer.id, walkOffer.filledCollateral,
  ↪  walkOffer.filledShares, exhausted);
}
```

The unusable offer stays in `liveBits`. `_alloc()` frees only expired slots, so a book holding `MAX_OFFERS` (32) such offers rejects new proposals with `TooManyOffers`, and `preLiquidate` reverts with `NoConsumableOffer` (`MorphoBorrowPosition.sol:429`) when the walk finds no valid fill.

```
// src/libs/borrow/LibBorrowOffers.sol:149-164
function _alloc(BorrowOffersStorage storage s) private returns (uint8 id) {
  uint256 liveBits = s.liveBits;
  uint256 scanBits = liveBits;
  while (scanBits != 0) {
    uint256 scanId = LibBit.ffs(scanBits);
    scanBits &= scanBits - 1; // clear the lowest set bit
    if (block.timestamp >= s.slab[scanId].expiresAt) {
      liveBits &= ~(uint256(1) << scanId);
      delete s.slab[scanId];
    }
  }
  uint256 freeBits = ~liveBits & ((uint256(1) << MAX_OFFERS) - 1);
  if (freeBits == 0) revert LibBorrowErrors.TooManyOffers();
```

```
    id = uint8(LibBit.ffs(freeBits));
    s.liveBits = uint32(liveBits | (uint256(1) << id));
  }
```

Example: Debt is USDC and collateral wUSCC, both 6 decimals; debt shares are quoted at Morpho's `VIRTUAL_SHARES` ratio of `1e6` shares per asset unit.

1. Each of the 32 offers holds 800,000 wUSCC (`8e11`) of collateral against 600,000 USDC (`6e11`) of debt, i.e. `6e17` debt shares.
2. A liquidator fills `8e11 - 3` collateral base units (800,000 wUSCC less 3 units of dust).
3. `fillShares = ceil((8e11 - 3) * 6e17 / 8e11) = 6e17 - 2,250,000`, leaving remaining collateral `3` (0.000003 wUSCC) and remaining debt shares `2,250,000` (about 0.00000225 USDC).
4. Both remaining values are nonzero, so the offer stays in `liveBits`, while `seizedValue` on 3 collateral units rounds down and `repaidDebtValue` rounds up, so the profitability and bonus-floor checks fail.
5. Repeat across all 32 slots. The book stays full holding 96 base units (0.000096 wUSCC) of collateral in total, and new offers covering the remaining 19,200,000 USDC of pre-liquidation capacity are blocked until a guardian revokes them or they expire.

Recommendation: Clear an offer when its remaining amounts cannot produce a nonzero fill that passes the offer checks.

3F: Fixed in PR 210: the consume write-back now releases a touched slot whose leftover fails strict profitability, so unusable dust remainders free their slots instead of blocking new proposals and consumes.

Cantina Managed: Fix verified.

3.3.19 `setConfig` reprices the repayment of an already-started Retargetter operation

Severity: Informational.

Context: Retargetter.sol#L148-L190, Retargetter.sol#L493-L498, Retargetter.sol#L702-L718.

Description: `startRetargetting()` snapshots only the per-operation yield cap into operation storage (`Retargetter.sol:227-231`). `tickDuration`, `tickThreshold`, and `horizon` stay in config storage, and `_owed()` reads them live at `Retargetter.sol:834-843`. The owner can call `setConfig()` after lenders already hold PT and YT, and the running operation settles on the new terms.

```
// src/manager/rebalancer/Retargetter.sol:832-844
function _owed(RetargetterOperation storage operation_) internal view returns
↪  (uint256) {
  (uint128 ptSupply, uint128 ytSupply) =
  ↪  ITokenController(operation_.request).totalSupplies();
  RetargetterConfig storage config_ = LibStorage.configStorage();
  return IRetargetterQuoter(_QUOTER)
    .repaymentOwed(
      ptSupply,
      ytSupply,
      block.timestamp - operation_.startedAt,
      config_.tickDuration,
      config_.tickThreshold,
      config_.horizon
    );
}
```

`RetargetterQuoter.repaymentOwed` charges at least one full tick (`RetargetterQuoter.sol:188-205`), so raising `tickDuration` raises the yield leg of the repayment for an operation that is already funded.

Example: PT and YT are denominated in the debt asset, USDC (6 decimals). Assume `horizon` 90 days (`MIN_HORIZON`), `tickDuration` 1 day, `tickThreshold` 0.

1. Operation has 5,000,000 USDC of PT (5e12) and 150,000 USDC of YT (1.5e11).
2. Owed at start: `paidDuration` charges one tick = 1 day, owed = 5e12 + ceil(1.5e11 * 86,400 / 7,776,000) = 5e12 + 1,666,666,667 = 5,001,666.666667 USDC.
3. Owner sets `tickDuration` to 30 days (`MAX_TICK_DURATION`) before repayment.
4. Same supplies now owe 5e12 + 1.5e11 * 2,592,000 / 7,776,000 = 5e12 + 5e10 = 5,050,000 USDC.

The repricing costs the borrower 48,333.33 USDC on a repayment that was funded on the 1-day tick.

Recommendation: The implementation `@dev` at `Retargetter.sol:600-601` states config changes apply to an in-flight operation, but `IRetargetter.setConfig` (`IRetargetter.sol:298-300`) does not. Carry that note onto the interface so lenders and integrations reading the ABI docs see that `tickDuration`, `tickThreshold`, and `horizon` are live inputs to repayment.

3F: Fixed in PR 213: `horizon`, `tickDuration` and `tickThreshold` are snapshotted into the operation at start alongside the yield cap and read from the operation in `_owed()`, so `setConfig` can no longer reprice a running operation.

Cantina Managed: Fix verified.

3.3.20 `CONSUMER_ROLE` can authorize itself to mint YT up to the yield cap

Severity: Informational.

Context: Retargetter.sol#L191-L230, LibStorage.sol#L149-L175, Request.sol#L328-L339.

`Retargetter.authorizeMinting()` at `Retargetter.sol:264` is gated by `onlyOwnerOrRoles(CONSUMER_ROLE)` and lets the caller pick the `to` recipient freely, including its own address. The only limits on the amounts are the yield ratio gate and the principal cap in `LibStorage.checkOffer()` at `LibStorage.sol:170-176`.

The recipient then calls `Request.mint()` (`Request.sol:378`), transfers `ptMintAuth` debt-asset units in, and receives the authorized PT and YT. The Retargetter trust description does not state that `CONSUMER_ROLE` holds this authority.

```
// src/manager/rebalancer/Retargetter.sol:264-283
function authorizeMinting(address to, uint128 ptAmount, uint128 ytAmount)
  external
  onlyOwnerOrRoles(CONSUMER_ROLE)
  nonReentrant
{
  RetargetterOperation storage operation_ = LibStorage.operationStorage();
  address request = operation_.checkRequest();
  // Replace semantics: drop the account from the set so the principal gate sizes
  ↪  the new
  // amounts as fresh capital; a full revocation (both amounts zero) skips every
  ↪  gate, since
  // it only shrinks exposure and must stay available once the window has closed
  EnumerableSetLib.AddressSet storage accounts = operation_.authorizedAccounts;
  accounts.remove(to);
  if (ptAmount != 0 || ytAmount != 0) {
    operation_.checkOffer(request, ptAmount, ytAmount, ptAmount,
    ↪  maxPrincipal(operation_.positionManager));
```

```
    // The capacity bound keeps every loop over the set within gas reach; see the
    ↪  constant
    accounts.add(to, MAX_AUTHORIZED_ACCOUNTS);
  }
  IRequest(request).authorizeMinting(to, ptAmount, ytAmount);
  emit MintingAuthorized(request, to, ptAmount, ytAmount);
}
```

```
// src/libs/manager/rebalancer/LibStorage.sol:170-177
if (expectedReturn * BPS > amount * self.operationMaxYieldBps) {
  revert LibRetargetterErrors.YieldTooHigh();
}
(uint128 ptSupply,) = ITokenController(request).totalSupplies();
if (ptSupply + pruneNullAuthorizations(self, request) + ptAmount > principalCap) {
  revert LibRetargetterErrors.PrincipalCapExceeded();
}
checkConsumptionWindow(self);
```

Example: PT and YT are denominated in the debt asset, USDC at 6 decimals.

1. `operationMaxYieldBps = 1000` (10%), live principal cap has 1,000,000 USDC (1e12) of headroom.
2. Consumer calls `authorizeMinting(consumer, 1e12, 1e11)`, i.e. 1,000,000 USDC of PT and 100,000 USDC of YT.
3. Yield gate: `1e11 * 10000 = 1e15` is not `> 1e12 * 1000 = 1e15`, so it passes at exactly the cap.
4. Principal gate: `ptSupply + authorizations + 1e12 <= cap`, passes.
5. Consumer calls `Request.mint()`, transfers 1,000,000 USDC (1e12), receives 1,000,000 USDC of PT and a 100,000 USDC yield entitlement in YT, directed to itself.

Recommendation: Document that `CONSUMER_ROLE` can authorize itself or another recipient to mint YT up to the operation's yield cap.

3F: Acknowledged as intended and documented on `authorizeMinting` in PR 213: `CONSUMER_ROLE` may authorize itself or any recipient to mint YT up to the operation's yield cap, bounded by the yield and principal gates.

Cantina Managed: Fix verified.

3.3.21 Offer proposers cannot cancel their own timelocked offers

Severity: Informational.

Context: MorphoBorrowPosition.sol#L715-L772, IBorrowOffers.sol#L79-L94.

Description: `proposeOffer` records `msg.sender` as the proposer and sets `activeAt = block.timestamp + timelock` (`MorphoBorrowPosition.sol:862-878`). `revokeOffers` gates on `checkCanRevokeOffer`, which requires `GUARDIAN_ROLE` or the registry owner (`MorphoBorrowPosition.sol:916-917`, `BorrowOffersRegistry.sol:174-176`). The docstring at `MorphoBorrowPosition.sol:913-915` states the omission is deliberate.

A proposer that submits a mistaken offer has to ask a guardian or the registry owner to pull it. If neither responds before `activeAt`, a permissionless liquidator can consume the offer on its submitted terms.

```
// src/borrow/MorphoBorrowPosition.sol:862-864
function proposeOffer(uint128 collateral, uint128 debtShares, uint40 expiresAt)
↪  external override returns (uint8 id) {
```

```
  OFFERS_REGISTRY.checkCanCreateOffer(msg.sender);
  if (collateral == 0 || debtShares == 0) revert LibBorrowErrors.OfferAmountZero();
```

```
// src/borrow/MorphoBorrowPosition.sol:916-917
function revokeOffers(uint8[] calldata ids) external override {
  OFFERS_REGISTRY.checkCanRevokeOffer(msg.sender);
```

Recommendation: Let the recorded proposer revoke its own offer before `activeAt`. Keep guardian and registry-owner revocation available at all times.

3F: Fixed in PR 210: the recorded proposer may now revoke its own offers at any time, before and after `activeAt`, while other callers still pass the guardian/owner check evaluated per offer.

Cantina Managed: Fix verified.

3.3.22 `PositionManager` exposes no getter for the bad-debt exclusion flag

Severity: Informational.

Context: LibView.sol#L55-L83, PositionManager.sol#L139-L147, IPositionManager.sol#L52-L60.

Description: `LibView.totalAssets()` skips any borrow module whose debt exceeds its quoted collateral, so that module contributes nothing to `amount`, `totalDebt`, or `totalCollateral`, and sets `hasBadDebt` (`LibView.sol:90-96`).

`PositionManager.totalAssets()` drops `hasBadDebt` (`PositionManager.sol:145-147`), and no public view returns it. Monitoring has to query every borrow module and re-implement the `collateral >= debt` comparison to tell whether the reported NAV covers all modules or only the solvent ones.

```
// src/libs/manager/LibView.sol:85-98
address[] memory modules = ps.borrowModules.values();
uint256 modulesLength = modules.length;
for (uint256 i = 0; i < modulesLength; ++i) {
  uint256 collateral = IBorrowPosition(modules[i]).totalCollateralQuoted();
  uint256 debt = IBorrowPosition(modules[i]).totalBorrowed();
  if (collateral >= debt) {
    amount += collateral - debt;
    totalDebt += debt;
    totalCollateral += collateral;
  } else {
    hasBadDebt = true;
  }
}
```

```
// src/manager/PositionManager.sol:144-147
/// @inheritdoc IPositionManager
function totalAssets() public view nonReadReentrant returns (uint256 amount) {
  (amount,,,) = LibStorage.positionManagerStorage().totalAssets();
}
```

Recommendation: Add a public `hasBadDebt()` view returning the flag from the same `LibView.totalAssets()` call, and document on `IPositionManager.totalAssets()` that the value excludes underwater modules when it is true.

3F: Fixed in PR 216: an additive `hasBadDebt()` getter, sourced from the same single-pass `LibView.totalAssets()` iteration, exposes the flag, and `totalAssets()` documents that it covers only solvent modules while the flag is set.

Cantina Managed: Fix verified.

3.3.23 Retargetter ignores the Fund state returned by `cancelOrder`

Severity: Informational.

Context: *(No context files were provided by the reviewer).*

Description: `cancelOrder` calls `IFund.cancel(order)` at `Retargetter.sol:409` but ignores the returned Fund state. The Retargetter then sets its local `orderLive` flag to false and clears its stored order.

Current Fund implementations return `EMPTY` or revert, so they remain consistent with the Retargetter. However, a future admitted Fund could return another state without reverting while its order remains active. The Retargetter would then forget an order that the Fund still processes.

Proof of Concept: Source: `src/manager/rebalancer/Retargetter.sol:409` :

```
IFund(fund).cancel(order_);
```

A non-conforming future Fund could behave as follows:

```
Future Fund.cancel(order):
  returned state = PROCESSING
  Fund order     = still active

Retargetter:
  orderLive      = false
  stored order   = cleared
```

Recommendation: Require `IFund.cancel(order)` to return `State.EMPTY` before clearing the Retargetter's local order state.

3F: Fixed in PR 213: `cancelOrder` now requires `IFund.cancel` to return `State.EMPTY` before clearing the local order and reverts `OrderPending` otherwise.

Cantina Managed: Fix verified.

3.3.24 Timelock changes do not retime existing offers

Severity: Informational.

Context: BorrowOffersRegistry.sol#L189-L209, BorrowOffersRegistry.sol#L251-L270, MorphoBorrowPosition.sol#L862-L878, LibBorrowOffersConstants.sol#L36-L42.

Description: `setOfferTimelock` schedules a new timelock using the collateral's current effective timelock. `offerConfig()` returns the pending value once its deadline is reached, even if the storage promotion has not yet been written. `proposeOffer` snapshots that effective value into `activeAt` when the offer is created.

For example, if the current timelock is 15 minutes and governance schedules an increase to 7 days, an offer proposed before the scheduled change becomes effective receives the 15-minute activation delay. After the change takes effect, new offers use 7 days, but the earlier offer keeps its original `activeAt` and can remain valid for up to the configured 365-day lifespan.

```
// src/borrow/BorrowOffersRegistry.sol:199-209
uint40 currentTimelock = _floorTimelock(_promoteTimelock(config));
uint40 effectiveAt = uint40(block.timestamp + currentTimelock);
config.pendingTimelock = timelock;
config.pendingTimelockAt = effectiveAt;
```

```
// src/borrow/MorphoBorrowPosition.sol:862-878
(uint40 timelock, uint16 minOfferBonusBps) = _offerConfig();
uint40 activeAt = uint40(block.timestamp + timelock);
//...
id = LibBorrowOffers.borrowOffersStorage().insert(msg.sender, activeAt, expiresAt,
↪  collateral, debtShares);
```

Recommendation: Document that timelock changes are grandfathered for existing offers.

3F: Acknowledged and documented in PR 215 on both ends: `proposeOffer` states later timelock changes never alter an offer's `activeAt`, and `setOfferTimelock` states a change affects future proposals only.

Cantina Managed: Fix verified.

3.3.25 `MidasFund` accepts arbitrarily old oracle rounds

Severity: Informational.

Context: MidasFund.sol#L668-L680, MidasFund.sol#L886-L928.

Description: `_getOraclePrice()` validates that the answer is positive, that `updatedAt` is nonzero, and that `answeredInRound >= roundId`. It does not check how much time has elapsed since `updatedAt`:

```
function _getOraclePrice() internal view returns (uint256) {
  AggregatorV3Interface _oracle = AggregatorV3Interface(_midasFundStorage().oracle);
  (uint80 _roundId, int256 _answer,, uint256 _updatedAt, uint80 _answeredInRound) =
  ↪  _oracle.latestRoundData();

  if (_answer <= 0) revert LibFundsErrors.ChainlinkInvalidAnswer();
  if (_updatedAt == 0) revert LibFundsErrors.ChainlinkIncompleteRound();
  if (_answeredInRound < _roundId) revert LibFundsErrors.ChainlinkStaleRound();

  return _answer.toUint256();
}
```

A completed round remains valid indefinitely even if the feed has stopped publishing. The price is used by `totalAssets()` and by the create-time sanity check for deposit `order.output`. A stale-high mToken price lowers the expected token output, allowing a lower minimum output to pass the deviation check; it also overstates the wrapper-wide AUM returned by `totalAssets()`.

The actual deposit mint amount is still determined when the Midas administrator approves the request, and governance can replace the feed through `setOracle()`. No direct attacker-controlled loss from a currently abandoned production feed has been demonstrated, so this is configuration and monitoring hardening.

Example: Assume the oracle's last completed round reports an mToken price of $1.00 and has not updated for one year. The current economic value has fallen to $0.80. All existing checks pass because the old answer is positive, `updatedAt` is nonzero, and the round is complete.

For a 100,000 USDC deposit, the stale price produces an expected output of 100,000 mTokens, so the 10% deviation check accepts an `order.output` of 90,000. At the current $0.80 value, the expected

output would be 125,000 mTokens and the equivalent lower bound would be 112,500. `totalAssets()` likewise continues valuing every wrapper share at $1.00.

Recommendation: Document the behavior and make sure this is monitored correctly.

3F: Acknowledged and documented on `setOracle` in PR 215: completed rounds carry no maximum-age check, so operators must monitor feed freshness and rotate a halted feed.

Cantina Managed: Fix verified.