Pashov Audit Group

Pare Security Review

September 7th 2026 - September 10th 2026

Contents

1. About Pashov Audit Group

Pashov Audit Group consists of 100+ freelance security researchers, who are well proven in the space - most have earned over $100k in public contest rewards, are multi-time champions or have truly excelled in audits with us. We only work with proven and motivated talent.

With over 500 security audits completed - uncovering and helping patch thousands of vulnerabilities - the group strives to create the most robust and pleasant audit experience possible. While 100% security is never possible to guarantee, we do guarantee you our team's best efforts for your project.

Check out our previous work here or reach out on Telegram @pashovkrum.

2. Disclaimer

A security review can never verify the complete absence of vulnerabilities. This is a time, resource and expertise bound effort where the researchers try to find as many issues as possible. We can not guarantee 100% security after the review. Subsequent reviews, bug bounty programs and on-chain monitoring are strongly recommended. The client's remediation of each issue has been reviewed; the outcome is recorded in the finding's Resolution section.

3. Risk Classification

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

Critical	Loss or compromise of critical assets, or protocol-wide impact.
High	Significant material loss of assets, or serious harm to a group of users.
Medium	Moderate material loss, or moderate harm, often under conditional assumptions.
Low	Minor impact, limited scope, or requires unlikely conditions to trigger.
Info	No direct security impact - best-practice, code-quality or hardening suggestion.

IMPACT

High	- leads to a significant material loss of assets in the protocol or significantly harms a group of users
Medium	- leads to a moderate material loss of assets in the protocol or moderately harms a group of users
Low	- leads to a minor material loss of assets in the protocol or harms a small group of users

LIKELIHOOD

High	- attack path is possible with reasonable assumptions that mimic on-chain conditions, and the cost of the attack is relatively low compared to the amount of funds that can be stolen or lost
Medium	- only a conditionally incentivized attack vector, but still relatively likely
Low	- has too many or too unlikely assumptions or requires a significant stake by the attacker with little or no incentive

4. Methodology

A large team of security researchers hunt vulnerabilities on the scope provided during a time-boxed review.

Every issue submission is deduplicated and validated by a judging team. Every accepted issue appears below with its impact and a recommended remediation. The client's response to each issue was then reviewed, and its resolution status - fixed, acknowledged, or invalidated - is recorded with the finding.

5. Executive Summary

A time-boxed security review of Pare was conducted by the Pashov Audit Group team. This report documents 41 issues across the reviewed codebase, reported by 0x37, 0xApple, 0xbrivan, Auditor-Nate, farismaulana, LoopGhost, Playboi, Valves Security.

SCOPE

https://github.com/pare-audit/contracts

Everything under "src/"

REVIEW COMMIT HASH

`2e780ae78beba72b8342ee9ef590ce420cb4356f`

FIXES REVIEW COMMIT HASH

`3dfa4a421d5d9d1fc2849f1847619e3b7dbf2ddb`

6. Findings

ID	TITLE	SEVERITY	STATUS
M-01	Two `unsynced` multiplier applies are recorded as one non-collapsed composite stamped at the LAST action's `effectiveAt`, same YT wipe, a distinct root cause	Medium	Fixed
M-02	USD-denominated feed is used as loan-token value without a loan-token/USD conversion	Medium	Fixed

ID	TITLE	SEVERITY	STATUS
M-03	Guardian resolution commitment can be reused after pending endpoints cycle	Medium	Fixed
M-04	Collapsed multiplier changes reuse the first update timestamp for all decomposed actions	Medium	Fixed
M-05	Manipulable Uniswap V3 TWAP can force premature PT liquidations	Medium	Fixed
M-06	Settled PT oracle lacks a redemption-value floor	Medium	Fixed
L-01	Holiday holds cross the immutable 4-day staleness gate because the feed stops printing hours before every close, freezing every Morpho market this oracle prices through the reopen	Low	Fixed
L-02	Magnitude-only dividend classification force-tags value-neutral in-band multiplier changes, permanently shifting PT principal to YT holders	Low	Fixed
L-03	Oracle accepts a feed during the L2 sequencer recovery window	Low	Fixed
L-04	Oracle resumes PT valuation before the post-dividend TWAP reflects the PT repricing	Low	Fixed
L-05	Oracle scaling is incorrect when stock and PT base-unit decimals differ	Low	Fixed
L-06	Pending guardian resolution becomes stale when sync collapses a later multiplier update	Low	Fixed
L-07	Pending multiplier endpoint equality hides intermediate updates	Low	Fixed
L-08	Per-call fee truncation permits fee-free micro-splits and YT redemptions	Low	Fixed
L-09	Per-call split-fee rounding permits fee-free fragmented deposits	Low	Fixed

ID	TITLE	SEVERITY	STATUS
L-10	Per-call YT fee rounding allows yield-fee avoidance through fragmented redemptions	Low	Fixed
L-11	YT redemption fee can be bypassed through fragmented redemptions	Low	Fixed
I-01	Multiplier processing can become unavailable because deployment accepts a zero administrator	Info	Fixed
I-02	Fees remain trapped and the deposit cap can be exceeded because the vault accepts itself as treasury	Info	Fixed
I-03	Guardian recovery can be delayed because malformed proposals overwrite valid resolutions	Info	Fixed
I-04	Timestamp freshness accepts economically stale feed answers	Info	Acknowledged
I-05	Immutable feed-decimal snapshot becomes stale after same-address feed migration	Info	Fixed
I-06	Chainlink feed can remain stale while the onchain stock market remains active	Info	Acknowledged
I-07	Accountant sync can clear while the Robinhood price feed remains deliberately paused	Info	Fixed
I-08	The oracle can bypass the PT vault's synchronization gate because its accountant is independently configured	Info	Fixed
I-09	Split tolerance can absorb a real dividend without crediting YT	Info	Acknowledged
I-10	Shape-only split classification routes value-additive clean-ratio events entirely into splitFactor, zeroing the YT drip at settlement	Info	Fixed
I-11	Immediate Multiplier Updates Can Be Missed at the Maturity Boundary	Info	Fixed

ID	TITLE	SEVERITY	STATUS
I-12	Four-Day Feed Freshness Window Accepts Stale Prices During Active Trading	Info	Acknowledged
I-13	accruedDripFactor exposes unsynchronized live index instead of a settlement-compatible value	Info	Fixed
I-14	Disabled feed-age validation accepts unfinished Chainlink rounds	Info	Fixed
I-15	Independent PT and YT redemption rounding strands matched-position dust	Info	Acknowledged
I-16	No recovery path for residual stock after all PT and YT liabilities are extinguished	Info	Fixed
I-17	Oracle Constructor Does Not Validate That the Pool Contains the Accountant's Stock Token	Info	Fixed
I-18	Oracle does not unconditionally reject incomplete Chainlink rounds	Info	Fixed
I-19	Post-maturity accountant changes can block settlement of an otherwise mature vault	Info	Fixed
I-20	Repeated dividend classification introduces cumulative downward index rounding drift	Info	Acknowledged
I-21	Split classification stores the rounded observed ratio instead of the validated clean ratio	Info	Fixed
I-22	Unsettled accruedDripFactor includes dividends occurring after maturity	Info	Fixed
I-23	Untracked token donations can permanently exhaust StripVault deposit capacity	Info	Fixed
I-24	Zero guardian delay removes the configured resolution reaction window	Info	Fixed

Medium Severity

[M-01] Two `unsynced` multiplier applies are recorded as one non-collapsed composite stamped at the LAST action's `effectiveAt`, same YT wipe, a distinct root cause

Medium Fixed

SUMMARY

When two ERC-8056 actions take effect with no sync() between them, a single sync() sees only their composite jump. The collapse flag is set only when a second poke observes a further change, so this composite pending is recorded as non-collapsed and never triggers guardian review:

```
// src/MultiplierAccountant.sol:161-167
```

_deriveEffectiveTimestamp then takes its first branch, because the token still exposes the most recent schedule (newUIMultiplier() == uiMultiplier()), so the whole composite is stamped with the last action's effectiveAt:

```
// src/MultiplierAccountant.sol:329-334
```

This is wrong for the composite: the first action's value accrued in-term, yet the composite inherits only the last action's timestamp. If that last effectiveAt falls after maturity, settle() freezes dT =dividendIndexAt(maturity) = d0.

Unlike the related block.timestamp fallback path, no newer schedule is involved here, and the stamp is a real (but wrong-for-the-composite) effectiveAt rather than block.timestamp, so this needs its own fix.

IMPACT

YT-only holders redeem 0 while PT holders absorb the entire term drip.

This path does not depend on any future schedule. It only requires two corporate actions in one observation window, for example two dividends, or a dividend plus a split, landing in the same keeper gap around maturity.

Because the pending is non-collapsed and two small dividends compose to a value within the dividend band (<= 3%), an honest classifier finalizes it directly. No guardian timelock ever gives holders a reaction window.

Fixing only the block.timestamp fallback does not close this path.

EXPLOIT PATH

1. Two ERC-8056 actions take effect with no sync() between them.
2. A single sync() observes only the composite jump and records it as non-collapsed.
3. The composite is stamped with the last action's effectiveAt, which lands after maturity.
4. The composite value stays within the dividend band, so the classifier finalizes directly with no timelock.
5. settle() freezes dT = d0, so YT holders redeem 0 and PT holders take the whole drip.

RECOMMENDATION

Do not attribute a multi-action composite to the last action's schedule. Require per-leg effective timestamps supplied at classification time (classifier for single-band composites, guardian otherwise)

and push one checkpoint per dividend leg. Route any classification whose checkpoint would land after a dependent series' maturity through the guardian timelock, so holders retain a window to exit via `merge()`.

RESOLUTION

Pare

Fixed in 3dfa4a4 on the audited repo: https://github.com/pare-audit/contracts/commit/3dfa4a4

[M-02] USD-denominated feed is used as loan-token value without a loan-token/USD conversion

Medium Fixed

SUMMARY

Morpho's `IOracle` expects the collateral price expressed in the market's loan token. This oracle derives that price from a stock/USD feed without any currency conversion in the path. The loan token's decimals enter only through the constructor's decimal normalization:

```
uint256 num = 36 + uint256(loanDecimals);
uint256 den = uint256(collateralDecimals) + uint256(feedDecimals);
scaleFactor = 10 ** (num - den);
```

`scaleFactor` changes the units, not the numeraire. The oracle never reads the loan token, its price, or any loan-token/USD rate, so `price()` is only correct while one loan token is worth exactly one USD.

The deployed market's loan token is USDG, and this peg assumption is nowhere enforced, asserted, or monitored. A depeg silently yields a wrong price, with no revert and no signal.

IMPACT

The USDG peg is a load-bearing, unchecked assumption behind every position in the market.

If USDG trades below one USD, the oracle understates collateral in the market's real unit of account and liquidates otherwise-healthy borrowers at the deployed 62.5% LLTV.

If USDG trades above one USD, collateral is overstated and borrowers can draw more USDG than the underlying stock can repay, leaving the excess as lender bad debt.

Because every oracle parameter is immutable, the numeraire cannot be corrected in place. The only remedy is deploying a new oracle and a new market.

RECOMMENDATION

Quote in the actual loan token, or add a validated loan-token/USD conversion leg to the price path. If neither is feasible, restrict deployment to a monitored peg and add a documented depeg circuit breaker.

RESOLUTION

Pare

Fixed in 89bb0da on the audited repo. Added an optional loanFeed (loan token / USD) leg to PareMorpho0racle with the same positivity and staleness checks as the stock feed. address(0) preserves the previous 1:1 behaviour and is now documented as a monitored-peg-only configuration.

https://github.com/pare-audit/contracts/commit/89bb0da

[M-03] Guardian resolution commitment can be reused after pending endpoints cycle

Medium Fixed

SUMMARY

A guardian proposal is meant to commit to one exact observed transition so the 2-day timelock lets holders review the decomposition that will actually execute. The commitment covers only the two endpoint multipliers and the decomposition itself:

```
resolutionHash = keccak256(abi.encode(pending.oldMultiplier, pending.newMultiplier, kinds,
ratiosWad));
```

It omits the `collapsed` flag, the effective timestamp, and any change identity. `sync()` mutates `pending` in place without clearing `resolutionHash`, even though the classifier path (`_finalize`) does clear it. `executeResolution` then re-hashes live storage and checks nothing further, and unlike `classifyDividend`/`classifySplit` it never calls `_loadClassifiablePending`, so `collapsed` is never consulted on the guardian path.

As a result a proposal authored against a single clean change can execute against a different, composite history whenever the multiplier returns to the proposed endpoint. For example the guardian proposes for

1. `0 -> 1.1`; during the timelock the token moves `1.1 -> 1.21 -> 1.1`, which marks the pending

change collapsed; the endpoints now match the original hash again, `executeResolution` accepts, and a single-action decomposition is applied to a three-action history. `isSynced()` then returns true and the incorrect split is locked in as settled state.

IMPACT

Multiplier growth is attributed to the wrong factor, `dividendIndex` versus `splitFactor`, for every consumer of the accountant, and the error is written as a checkpoint that later reads treat as historical truth.

Each StripVault snapshots `dividendIndex` at settlement, so a wrong decomposition permanently mis-splits principal from yield across the whole series: PT holders redeem against an index reflecting actions the guardian never decomposed, and YT holders receive the difference.

The timelock's disclosure guarantee is also defeated. What holders reviewed for two days is not what executed, and no fresh review window is opened.

EXPLOIT PATH

1. Guardian proposes a resolution for a clean `1.0 -> 1.1` change, starting the 2-day timelock.

2. During the timelock the token multiplier moves `1.1 -> 1.21 -> 1.1`, and `sync()` marks the pending change collapsed without clearing `resolutionHash`.
3. Because the endpoints again match the original hash, `executeResolution` accepts and applies the single-action decomposition to the now three-action history.
4. `isSynced()` returns true and the incorrect split/dividend attribution is locked in as settled state.

RECOMMENDATION

Bind each proposal to a monotonic pending-change nonce so a mutated pending change cannot satisfy an old commitment, and clear `resolutionHash` whenever `sync()` mutates the pending change.

RESOLUTION

Pare

Fixed in ca0bd02 on the audited repo: https://github.com/pare-audit/contracts/commit/ca0bd02

Added pendingNonce, incremented on every creation or mutation of the pending change, and bound into the resolution hash on both proposeResolution and executeResolution. sync() now clears resolutionHash/resolutionEta (emitting ResolutionCancelled) whenever it mutates the pending change. Test reproduces the 1.0 -> 1.1 -> 1.21 -> 1.1 path: the proposal is cancelled on the first mutation and executeResolution reverts.

[M-04] Collapsed multiplier changes reuse the first update timestamp for all decomposed actions

Medium Fixed

LOCATION

`MultiplierAccountant:sync:159-168`

SUMMARY

When multiple multiplier updates are collapsed into a single pending change, the timing of each underlying update is lost. `sync()` overwrites only `newMultiplier` and sets `collapsed=true`, but does not preserve the individual timestamps of the decomposed actions.

`executeResolution()` then applies all decomposed ratios while recording the final `dividendIndex` using the original `pending.timestamp`, regardless of when the later actions actually occurred.

Because vault maturity determines which dividend-index increases belong to a series, a dividend that became effective after maturity can be backdated to before maturity. For example, a change observed at timestamp 800 stays pending for a vault maturing at 1000; a 5% dividend at timestamp 1100 is collapsed into it and later checkpointed at 800, so StripVault freezes an inflated `dT`.

IMPACT

Post-maturity dividend growth is backdated into the vault term. PT holders are underpaid relative to their baseline redemption because of the inflated `dT` ratio, while YT holders receive value for dividends that only accrued after maturity.

The same defect can also leave the system unsynced while the collapsed resolution remains delayed, blocking settlement from completing.

EXPLOIT PATH

1. A first multiplier change is observed at timestamp 800 and remains pending for a vault maturing at timestamp 1000.
2. A 5% dividend occurs at timestamp 1100; `sync()` collapses it into the existing pending change without updating the timestamp.
3. Guardian resolution applies ratios for both the initial action and the 5% dividend, then writes the final checkpoint at timestamp 800.
4. Settlement at or after maturity treats the 5% dividend as pre-maturity growth, reducing PT payouts by the inflated `dT` ratio and shifting that value to YT holders.

PREREQUISITES

At least two multiplier updates must remain unclassified until the later one is observed, with one dividend component occurring after vault maturity, and the composite must be resolved (via guardian resolution) before settlement can complete.

RECOMMENDATION

Represent each action's timing explicitly in the resolution and write checkpoints separately, validating chronological order:

```
struct ResolutionAction {
    ActionKind kind;
    uint256 ratioWad;
    uint64 effectiveAt;
}
// Validate chronological timestamps, then push each dividend checkpoint.
```

If per-action timing cannot be recovered, conservatively prevent a collapsed resolution from being used for any vault whose maturity falls between the first and last observed updates.

RESOLUTION

Pare

Fixed in 0dc448e on the audited repo: https://github.com/pare-audit/contracts/commit/0dc448e

Each action in a resolution now carries its own effective timestamp (uint64[] alongside kinds and ratiosWad, bound into the commitment). executeResolution validates chronological order inside the pending change's observation window and pushes one checkpoint per Dividend action at that action's time, so a post-maturity dividend can no longer be backdated to the first update's timestamp. Test reproduces the 800 / 1000 / 1100 path: the maturity reads only the first action's index.

[M-05] Manipulable Uniswap V3 TWAP can force premature PT liquidations

Medium Fixed

LOCATION

`PareMorphoOracle:ptPerStockWad:105-119`

SUMMARY

The oracle values PT collateral by taking the stock value from Chainlink and estimating the PT discount from a PT/stock Uniswap V3 TWAP. It accepts any TWAP-derived PT price below parity and only caps it at parity:

```
uint256 token1PerToken0 = priceWadAtTick(meanTick);
uint256 p = ptIsToken0 ? token1PerToken0 : (WAD * WAD) / token1PerToken0;
return p > WAD ? WAD : p;
```

The parity cap only limits upward prices, so a downward manipulation passes through unchecked. The price is never compared against the PT's vault accounting or redemption economics.

If the pool is thin, an attacker can push the TWAP low enough to make healthy or marginal positions appear undercollateralized. Because merge requires both PT and YT, it does not establish a unilateral PT redemption floor that would neutralize this price source before maturity.

IMPACT

A depressed, manipulated TWAP can trigger liquidations at an artificially low PT valuation. Victims lose collateral even though the fair PT value remains above the liquidation boundary, while the attacker can profit if the seized PT later recovers or can be paired and merged.

EXPLOIT PATH

1. A victim supplies PT to Morpho and borrows near, but below, the fair-value liquidation limit.
2. The attacker trades or shifts concentrated liquidity to depress the pool tick and holds that price for enough of `twapWindow` to move the TWAP.
3. `PareMorphoOracle.price()` returns the depressed PT price, so the victim fails Morpho's collateralization check.
4. The attacker liquidates the victim, acquiring PT collateral at the manipulated valuation, and benefits if the TWAP and PT market later recover.

PREREQUISITES

The Morpho market must use this oracle and have positions near the liquidation threshold. The configured PT/stock pool must be thin enough that an attacker can depress its TWAP for a meaningful fraction of `twapWindow`.

RECOMMENDATION

Bound the AMM-derived PT price against an independent conservative valuation before exposing it to Morpho liquidation logic:

```
uint256 ammPrice = ptPerStockWad();
uint256 floor = conservativePtFloorWad();
if (ammPrice < floor) revert PriceOutsideBounds();
```

A stronger fix uses multiple independent sources and rejects liquidation pricing when the AMM TWAP deviates beyond a configured tolerance:

```
if (absDiff(ammPrice, referencePrice) > maxDeviation) {
    revert PriceDeviation();
}
```

RESOLUTION

Pare

Fixed in e9b32c8 on the audited repo: https://github.com/pare-audit/contracts/commit/e9b32c8

Added a configured floor, minPtPerStock, on the AMM-derived PT/stock leg. Before settlement, a TWAP below the floor makes price() revert with PriceOutsideBounds, so a depressed pool blocks liquidations rather than pricing them; supply and repay still work. Zero disables the check; a floor above parity is rejected at construction. Once settled the pool leg is not consulted. Tests cover a pool pushed to 0.50 (reverts), one tick either side of the floor, floor disabled, and floor not applying after settlement.

[M-06] Settled PT oracle lacks a redemption-value floor

Medium Fixed

LOCATION

`PareMorphoOracle:price/ptPerStockWad:99-125`

SUMMARY

After settlement, each PT can be redeemed for a fixed, known quantity of stock. The oracle should never value it below that redemption claim for solvency decisions, but it stays fully pool-priced and only caps the value from above at parity.

`ptPerStockWad()` derives a TWAP and applies only an upper cap:

```
uint256 p = ptIsToken0
    ? token1PerToken0
    : (WAD * WAD) / token1PerToken0;
return p > WAD ? WAD : p;
```

There is no comparison against `dividendIndex0` or `dividendIndexAtMaturity`. A downward TWAP can therefore report a value below the fixed redemption amount. For example, if settlement freezes a redemption of 0.90 stock per PT but the pool TWAP is pushed to 0.50, the oracle returns 0.50.

IMPACT

Borrowers can be liquidated based on a price below their PT collateral's fixed redemption value.

A liquidator seizes PT at the depressed oracle valuation and then redeems it for the higher fixed stock payout, capturing the spread. This transfers value directly from borrowers to the liquidator.

EXPLOIT PATH

1. `StripVault.settle()` freezes a redemption rate of 0.90 stock per PT while PT remains collateral in Morpho.
2. The attacker manipulates the PT/stock pool so the TWAP reads 0.50 stock per PT.
3. The oracle returns 0.50 because no redemption-value floor is applied.

4. The attacker liquidates borrowers at the depressed valuation, seizes their PT, and redeems it for 0.90 stock per PT.

PREREQUISITES

The StripVault must be settled, borrowers must hold PT collateral in the Morpho market, and the configured Uniswap v3 pool must be movable enough over the oracle's TWAP window.

RECOMMENDATION

Reference the StripVault and clamp the post-settlement pool price to the fixed redemption rate:

```
uint256 marketPrice = ptPerStockWad();
if (vault.settled()) {
    uint256 redemption = vault.dividendIndex0().mulDiv(
        WAD, vault.dividendIndexAtMaturity()
    );
    if (marketPrice < redemption) marketPrice = redemption;
}
return (uint256(answer) * scaleFactor).mulDiv(marketPrice, WAD);
```

A safer alternative is to use the fixed redemption rate directly after settlement, avoiding a manipulable pool leg for settled PT collateral entirely.

RESOLUTION

Pare

Fixed in e5b50c0 on the audited repo: https://github.com/pare-audit/contracts/commit/e5b50c0

Took the safer alternative. The oracle now references the PT's StripVault and, once settled() is true, prices the PT at dividendIndex0 / dividendIndexAtMaturity, the fixed redemption rate, without consulting the pool at all. Before settlement the TWAP leg is unchanged. The constructor checks vault.pt() matches the priced PT. Tests cover a settled vault with the pool pushed to 0.50 and to parity; both price at the redemption rate.

Low Severity

[L-01] Holiday holds cross the immutable 4-day staleness gate because the feed stops printing hours before every close, freezing every Morpho market this oracle prices through the reopen

Low Fixed

SUMMARY

`price()` reverts `StaleFeed(updatedAt)` as soon as the feed's last update is older than `maxFeedAge` (src/PareMorphoOracle.sol:119). `maxFeedAge` is immutable, set once at construction (:77, :104), and the oracle is deployed per series with the PT, pool, feed, and margin chosen at deploy time (:87-104). The live pSPY instance runs `345_600` seconds, exactly 4 days.

The Chainlink docs for these Robinhood tokenized-equity feeds state the feeds have no heartbeat during off-hours, may hold the last published price while markets are closed (weekends, holidays, thin overnight windows), and leave staleness bounds to the integrator.

The feed also stops printing hours before each close. Across the ten weekends on record (feed live since 2026-06-22, rounds 1 to 121) the last Friday print led the close by up to 8.6 hours, median 5.3, and the feed never printed on a Saturday. On 2026-08-21 it went silent at 07:25:52 ET, 8.57 hours before the 16:00 close.

The span the 4-day gate must survive is the early-stop lead plus the closure plus the reopen lag, and nothing in the docs commits to a print inside it:

- **Monday holidays** (Labor Day, MLK, Presidents' Day, Memorial Day): 89.5 hours from Friday close to Tuesday 09:30 reopen. The observed Labor-Day Friday stop (11:12 ET, 4.79 hours early) left only 1.71 hours of margin; repeating the observed 2026-08-21 stop makes the age at reopen 98.07 hours, crossing the gate 2.07 hours before the market opens.
- **Christmas on a Friday** (next 2026-12-25, with the 2026-12-24 half-day 13:00 close): 92.5 hours from Thursday close to Monday reopen. Eight of the ten observed stop patterns cross the gate before the Monday open; the median stop crosses by 1.8 hours.
- **Timing-independent crossings**: any closure of four or more full days (the 2001 shutdown closed the market for four trading days), and an `oraclePaused()` window held past four days, during which the feed holds the last known good value.

The oracle's own header (src/PareMorphoOracle.sol:51-53) warns a window shorter than a weekend "would brick every Monday; pick `maxFeedAge` with that in mind" and notes a wrong value is fixable only by deploying a new oracle and market (:55-56).

IMPACT

The freeze begins while the market is still closed and swallows the reopen, exactly when positions must be managed. Every price-gated operation reverts at and after the open: borrow, collateral withdrawal, and liquidation all fail, while supply and repay keep working because they never read `price()` (src/PareMorphoOracle.sol:47-50).

Because liquidation is blocked, a gap in the underlying stock at reopen cannot be liquidated and accrues as unenforceable bad debt to the market's lenders until the first post-closure print arrives.

The live pSPY/USDG market at 62.5% LTV is the instance running `345_600` today. Every future Pare series deploys a new oracle and inherits the same exposure unless the margin is re-sized, and every Robinhood tokenized-equity feed carries the documented no-heartbeat hold.

The oracle fails closed, so no mispricing occurs and the freeze self-heals on the next print. But because `maxFeedAge` is immutable, the exposure recurs at every long closure and correcting it requires deploying a new oracle and a new market.

EXPLOIT PATH

1. On the final session before a long closure the feed goes silent hours before the close (observed 2026-08-21: last print 07:25:52 ET).
2. It holds that last price through the closure, with no off-hours heartbeat.
3. On a Monday holiday the age at the Tuesday 09:30 reopen is 89.5 hours plus the stop lead, reaching

98.07 hours with the observed worst stop, so `price()` reverts `StaleFeed` before the market opens. On the Christmas shape, eight of ten observed stop patterns cross.

1. Borrow, collateral withdrawal, and liquidation on the affected Morpho market revert for the whole reopened session until the first fresh print lands.

RECOMMENDATION

Size the staleness bound to the span the gate actually faces, including the early stop and the reopen lag; the deployed 4 days sits inside that span on the strongest holiday shape combined with the feed's own worst observed stop. Deploy with at least 5 days of margin.

Replace hard immutability with a bounded setter so the margin can track the observed print cadence without redeploying the oracle and its market. Failing closed remains correct once a genuinely sized margin is exceeded.

```
-uint256 public immutable maxFeedAge;
+uint256 public maxFeedAge; // settable within [MIN_MAX_FEED_AGE, MAX_MAX_FEED_AGE]
```

RESOLUTION

Pare

Fixed in fc5b915 on the audited repo: https://github.com/pare-audit/contracts/commit/fc5b915

Sizing: agreed, and the live oracle will be redeployed with a 5-day window after the report. In code, the constructor now rejects any non-zero maxFeedAge under MIN_FEED_AGE = 5 days so an undersized window cannot be deployed again; the NatSpec carries the holiday and early-stop figures. We kept maxFeedAge immutable rather than adding a bounded setter: the oracle, like the rest of the system, has no admin role, and we'd rather keep that property and size the margin correctly at deploy than introduce a key that can change pricing parameters on a live market.

[L-02] Magnitude-only dividend classification force-tags value-neutral in-band multiplier changes, permanently shifting PT principal to YT holders

Low Fixed

SUMMARY

`classifyDividend` validates only the magnitude of the observed ratio, not whether the multiplier step actually carried new value (src/MultiplierAccountant.sol:174-181). The band check accepts any r in (1e18, 1e18 + maxDividendDeltaBps x WAD / BPS], which is (0, +3%] at the deployed 300 bps (script/Deploy.s.sol:37), then folds the full ratio into `dividendIndex` and pushes a drip checkpoint.

Nothing in that check establishes that value was created. A small split, an issuer multiplier correction, or a small stock dividend whose per-share price scales inversely moves `uiMultiplier` while leaving the per-token total-return value unchanged, and the accountant cannot observe the difference. Its own documentation concedes that no price-step check can separate the classes (src/MultiplierAccountant.sol:17-21), yet the magnitude band treats every in-band shape as reinvested dividend income.

The split path cannot rescue such an event. `_validateSplitRatio` gates on `|r - 1e18| >=minSplitDeltaBps` before the term check (src/MultiplierAccountant.sol:298-303), so any sub-3% ratio reverts `NotASplit`; the exact 2% witness 51:50 also exceeds maxSplitTerm 20. Declining to classify is not neutral either: while a change is pending, `settle()` reverts `AccountantNotSynced` (src/StripVault.sol:153),

blocking redemption of any maturing series, and the only attribution that avoids the 2-day guardian timelock (src/MultiplierAccountant.sol:200-208) is the value-shifting Dividend tag.

The constructor's band separation is documented as guaranteeing that a keeper can never tag a real split as a dividend and steal PT principal (src/MultiplierAccountant.sol:29-31); a value-neutral in-band change defeats exactly that guarantee.

IMPACT

Every PT holder of an affected series permanently loses principal to the YT holders on each mis-tagged event. The PT cohort's loss is exactly the YT cohort's gain, moved by a corporate action that created no value.

A phantom Dividend tag multiplies `dividendIndex` by (1 + d) for an event of relative size d, so settlement freezes dT = d0 x (1 + d) (src/StripVault.sol:154-155). Each PT then redeems amount x d0/dT = amount / (1 + d), a loss of amount x d / (1 + d): 1.96% per PT at a 2% event, and 2.91% at the 3% deployed band cap (the maximum a single tag can move). Each YT redeems the identical figure gross (src/StripVault.sol:175), net of the 5% yield fee to the treasury.

Events compound: four 1% events take 3.90% of PT principal, and n band-cap events take 1 - 1/1.03^n, converging on all of it. Every live series maturing after an event suffers the transfer simultaneously.

The transfer is silent and one-way: payouts still sum exactly to vault holdings, so nothing reverts and no alarm fires. dT is frozen at settlement with no post-settlement correction path. A guardian [Split 1 + d] resolution of the identical on-chain state would return PT its full principal and pay YT nothing, but that correct route sits behind a 2-day public timelock that nothing obliges anyone to start.

EXPLOIT PATH

1. The issuer processes a value-neutral in-band action: `uiMultiplier` moves 1e18 to 1.02e18 with per-share price scaling by 1/1.02, so per-raw-token total-return value is continuous. Any caller syncs it.
2. The keeper tags Dividend, the protocol's designed route for every in-band increase: the magnitude check passes (0.02e18 <= 0.03e18), and no Split tag exists for this shape since the `|r - 1e18| >=2000 bps` gate rejects every sub-20% ratio.
3. `dividendIndex` advances to 1.02e18 and a phantom-drip checkpoint is pushed.
4. Every series maturing after the event settles with dT = 1.02e18; `redeemPT` pays a 196 bps principal haircut while `redeemYT` pays the same amount out as drip.
5. The mis-attribution is permanent: dT is frozen at settle with no correction path, and the guardian decomposition is only available while the change is pending.

PREREQUISITES

The value-transferring case requires a keeper holding the CLASSIFIER_ROLE. Even an honest keeper cannot avoid the loss for a genuinely value-neutral in-band event, because Dividend is the only tag that both passes classification and avoids the settlement-blocking timelock. A compromised or self-interested keeper positioned long YT can deliberately force-tag every value-neutral in-band event as Dividend to harvest amount x d / (1 + d) of PT principal per event.

RECOMMENDATION

An in-band magnitude alone does not establish that value was created, so the immediate keeper tag must not commit a value-shifting index fold on magnitude evidence only. Consider either of:

1. Make the Dividend classification provisional: record the intended fold at tag time and apply it to `dividendIndex` only after the same public reaction window the guardian path uses, letting holders

exit via the always-free `merge()` before the drip checkpoint lands and letting the guardian reattribute the leg during the window.

2. Gate the immediate tag on the issuer's declared corporate-action type: the ERC-8056 schedule that `sync()` already reads for `effectiveAt` (src/MultiplierAccountant.sol:328-337) carries the action class, and only a declared reinvested cash dividend should be immediately foldable.

```
 function classifyDividend() external onlyRole(CLASSIFIER_ROLE) {
     PendingChange memory p = _loadClassifiablePending();
     uint256 r = _ratioWad(p);
     if (r <= WAD || r - WAD > (maxDividendDeltaBps * WAD) / BPS) revert NotADividend(r);
-    dividendIndex = dividendIndex.mulDiv(p.newMultiplier, p.oldMultiplier);
-    _pushCheckpoint(p.timestamp, dividendIndex);
-    _finalize(ActionKind.Dividend, p);
+    _scheduleFold(p, p.newMultiplier.mulDiv(WAD, p.oldMultiplier)); // applies after
guardianDelay
 }
```

RESOLUTION

Pare

Fixed in 2db5aa9 on the audited repo: https://github.com/pare-audit/contracts/commit/2db5aa9

Took option 1. A Dividend classification (single or multi-leg) is now provisional: the fold is recorded and applied to dividendIndex only after guardianDelay, via applyFold() or the next sync(). During the window the guardian can call reattributeFoldAsSplit(), which credits the whole change to splitFactor instead. No further classification or resolution can start while a fold waits. StripVault.settle() now also requires foldPending() to be false, so a series cannot freeze dT on an index that excludes a tagged dividend; merge, split and the oracle are unaffected during the window. Option 2 was not available: the ERC-8056 schedule interface exposes only newUIMultiplier and effectiveAt, not the action class.

[L-03] Oracle accepts a feed during the L2 sequencer recovery window

Low Fixed

LOCATION

`PareMorphoOracle:price:113-122`

SUMMARY

The oracle is meant to reject Morpho price reads not only when a Chainlink answer is old, but also while an L2 sequencer is down or has only just recovered. This prevents an on-chain answer that looks recent from being trusted during the sequencer recovery window.

The implementation only checks that the answer is positive and that `updatedAt` is within `maxFeedAge`. It has no sequencer uptime dependency and no grace-period condition. A feed answer that was published before an outage and not yet refreshed after recovery therefore passes every local check.

Because the retained price can be materially stale, Morpho's borrowing, collateral withdrawal, and liquidation decisions use an incorrect collateral value.

IMPACT

During or immediately after sequencer downtime, Morpho can use a stale but age-valid collateral price.

If the retained price is too high, borrowers can over-borrow against collateral worth less. If it is too low, users face incorrect liquidations until a fresh feed update arrives.

EXPLOIT PATH

1. The oracle runs on a sequencer-based chain with a long `maxFeedAge` suitable for equity-market closures.
2. The sequencer goes offline while the Chainlink answer reflects a high stock price.
3. The stock price falls during downtime; after recovery the feed has not yet updated and its old timestamp still passes the age check.
4. A borrower uses the accepted high PT price to borrow more than the collateral is worth, or a liquidator/user receives an incorrect result until a fresh feed update lands.

PREREQUISITES

The oracle must run on a sequencer-mediated chain, the sequencer must experience downtime or a recent recovery, and the stale feed timestamp must remain within the configured `maxFeedAge`. These are realistic external conditions for L2 integrations.

RECOMMENDATION

Add a sequencer uptime feed and reject down or recently recovered states before reading the collateral price:

```
function _checkSequencer() internal view {
    (, int256 answer,, uint256 startedAt,) = sequencer.latestRoundData();
    if (answer != 0 || block.timestamp - startedAt <= gracePeriod)
        revert SequencerUnavailable();
}
```

Call `_checkSequencer()` at the start of `price()` and keep the existing `maxFeedAge` validation for the Chainlink price itself.

RESOLUTION

Pare

Fixed in c5c827d on the audited repo: https://github.com/pare-audit/contracts/commit/c5c827d

Added an optional sequencer uptime feed and grace period as recommended: price() reverts with SequencerUnavailable while answer != 0 or within sequencerGrace of startedAt, before any feed read. address(0) disables the check for chains without an uptime feed. One related change: the extra constructor arguments hit the stack limit, so the oracle now derives pt from vault.pt() rather than taking it separately; VaultMismatch is removed since there is nothing left to mismatch. Tests cover down, recovered-inside-grace, recovered-past-grace, and feed disabled

[L-04] Oracle resumes PT valuation before the post-dividend TWAP reflects the PT repricing

Low Fixed

LOCATION

`PareMorphoOracle:price:82-93`

SUMMARY

The oracle resumes pricing based only on `accountant.isSynced()`, but that sync flag flips immediately when a dividend is classified, while the PT/stock Uniswap v3 TWAP only changes gradually as old observations age out of the averaging window.

```
if (!accountant.isSynced()) revert NotSynced();
uint256 ptPerStock = ptPerStockWad();
return (uint256(answer) * scaleFactor).mulDiv(ptPerStock, WAD);
```

This opens a post-classification interval in which the oracle is available while its PT valuation still reflects the old, higher price. For example, if a dividend drops the PT/stock spot price from 1.00 to 0.95, the 30-minute TWAP can remain near 1.00 for the rest of the window, so the oracle keeps valuing PT at the pre-dividend level.

IMPACT

After an abrupt dividend, PT collateral can stay overvalued for up to the configured TWAP window.

A borrower can supply PT and borrow against the inflated oracle value. Once the TWAP converges to the post-dividend price the position is undercollateralized, leaving the Morpho market with bad debt.

EXPLOIT PATH

1. A dividend causes the PT/stock spot price to fall (e.g. 1.00 to 0.95) while the pool TWAP still reflects pre-dividend observations near 1.00.
2. The classifier calls `classifyDividend()`, so `accountant.isSynced()` returns true immediately.
3. The attacker supplies PT to Morpho and borrows against the still-inflated oracle value.
4. The TWAP converges to the post-dividend price, leaving the position undercollateralized and the market exposed to bad debt.

PREREQUISITES

A configured Morpho market must use this oracle, and a dividend must cause the PT/stock spot price to drop while the TWAP window still contains predominantly pre-dividend observations.

RECOMMENDATION

Delay oracle availability until the TWAP has incorporated sufficient post-classification observations. This requires exposing the classification/effective timestamp from the accountant and enforcing a conservative delay in the oracle:

```
if (!accountant.isSynced()) revert NotSynced();
if (block.timestamp < accountant.lastClassificationTime() + twapWindow) {
    revert TwapNotRefreshed();
}
```

Alternatively, apply the newly classified dividend adjustment immediately to the PT leg and use the TWAP only for the residual market discount, so pricing never relies on stale pre-event observations.

RESOLUTION

Pare

Fixed in 0de79e1 on the audited repo: https://github.com/pare-audit/contracts/commit/0de79e1

Took the first recommendation. MultiplierAccountant now exposes lastClassifiedAt, set on classifyDividend, classifyDividends, classifySplit and executeResolution. PareMorphoOracle refuses the pool leg with TwapNotRefreshed until one twapWindow has elapsed since it, so the TWAP has fully incorporated post-classification observations before collateral is priced. The hold does not apply after settlement, where the fixed redemption rate is used instead.

[L-05] Oracle scaling is incorrect when stock and PT base-unit decimals differ

Low Fixed

LOCATION

`PareMorphoOracle:constructor/ptPerStockWad:82-128`

SUMMARY

The vault keeps a one-to-one relationship between raw stock units and raw PT units, so PT raw supply is minted from the stock token's base-unit amounts. Morpho, however, needs an oracle value expressed in loan base units per PT base unit, which means the stock token's own decimal scale is part of the conversion and cannot be derived from PT metadata alone.

The oracle instead scales the Chainlink stock price using the configured PT collateral decimals (fixed at 18). When the stock token has non-18 decimals, this applies the wrong power of ten even though the PT/stock TWAP is measured in matching raw units, so the resulting price is deterministically mis-scaled by the decimal difference.

For example, a 6-decimal stock token minting one PT raw unit per stock raw unit produces roughly a `10^12` valuation error because the price is scaled as though stock used 18 decimals.

IMPACT

The Morpho oracle underprices or overprices PT by the decimal mismatch factor.

Underpricing (typical for lower-decimal stock tokens) can make the market unusable or trigger premature liquidations. Overpricing (possible for higher-decimal stock tokens) can permit undercollateralized borrowing beyond the actual PT-backed value and lead to bad debt.

EXPLOIT PATH

1. Deploy a vault for a 6-decimal stock token; PT stays 18 decimals and mints one PT raw unit per stock raw unit.
2. Deposit `1,000,000` stock base units and receive `1,000,000` PT base units, giving a PT/stock raw-unit pool ratio near parity.
3. Deploy `PareMorphoOracle` with `collateralDecimals = 18` and a feed reporting the price of one whole stock token.
4. Morpho consumes a price scaled as if stock raw units used 18 decimals, producing an approximately `10^12` valuation error and incorrect borrowing or liquidation decisions.

PREREQUISITES

A supported stock token has decimals different from PT's fixed 18 decimals, and a Morpho oracle is deployed using that PT as collateral. The configured Chainlink feed prices a whole stock token, as feeds conventionally do.

RECOMMENDATION

Use the stock token's decimals for the feed-to-raw-stock conversion, keeping PT decimals only for documenting and validating the Morpho collateral token:

```
uint8 stockDecimals = IERC20Metadata(stock_).decimals();
uint256 num = 36 + uint256(loanDecimals);
uint256 den = uint256(stockDecimals) + uint256(feedDecimals);
if (den > num) revert ScaleOverflow();
scaleFactor = 10 ** (num - den);
```

Alternatively, require the two to match in the constructor and reject unsupported mismatches:

```
if (IERC20Metadata(stock_).decimals() != collateralDecimals) {
    revert DecimalMismatch();
}
```

RESOLUTION

Pare

Fixed in 8867de4 on the audited repo: https://github.com/pare-audit/contracts/commit/8867de4

Took the first option. The constructor now reads decimals() from the vault's stock token and uses it in the scale factor; the collateralDecimals argument is removed so the value cannot be misconfigured. Test covers a 6-decimal stock producing a 1e28 scale factor against 1e16 for 18 decimals.

[L-06] Pending guardian resolution becomes stale when sync collapses a later multiplier update

Low Fixed

LOCATION

`MultiplierAccountant.sol:sync:147-167`

SUMMARY

A guardian proposal is meant to commit to one exact pending multiplier transition, executable after the public timelock only if its hash and decomposition still match. The permissionless `sync()` path breaks this: it can mutate the pending transition after the proposal was committed, while leaving the proposal hash and expiry untouched.

```
pending.newMultiplier = m;
pending.collapsed = true;
// resolutionHash and resolutionEta remain unchanged
```

Because `executeResolution()` hashes live `pending` storage rather than an immutable snapshot of the originally committed transition, the recomputed hash no longer matches:

```
bytes32 h = keccak256(abi.encode(
    p.oldMultiplier, p.newMultiplier, kinds, ratiosWad
));
```

For example, with a pending change from M0 to M1, a guardian proposes a decomposition for M0 to M1. If `token.uiMultiplier()` then becomes M2 and any caller runs `sync()`, `pending.newMultiplier` becomes M2 while `resolutionHash` still commits to M1. The later `executeResolution()` reverts with `ResolutionMismatch()`. This is triggered by a normal subsequent multiplier update, not by malicious guardian action.

IMPACT

A routine multiplier update during the timelock invalidates the existing guardian proposal without explicitly cancelling or restarting it, leaving the pending change unresolved and the accountant unsynced.

While in this state, StripVault settlement is blocked and `PareMorphoOracle.price()` reverts, so oracle price reads are unavailable until a fresh timelock completes.

The guardian must propose a new M0-to-M2 resolution and wait another delay. Because the trigger is ordinary permissionless `sync()` activity, repeated naturally occurring multiplier updates can keep settlement and oracle reads unavailable for an arbitrarily long period. Recovery is possible via guardian action, so this is a temporary availability disruption rather than a permanent loss.

EXPLOIT PATH

1. A pending change from M0 to M1 exists and the guardian calls `proposeResolution()` for a decomposition of M0 to M1.
2. Before the two-day delay expires, `token.uiMultiplier()` becomes M2 and any caller invokes `sync()`; `pending.newMultiplier` becomes M2 while `resolutionHash` still commits to M1.
3. After `resolutionEta`, the guardian calls `executeResolution()` with the original decomposition; the recomputed hash for M0 to M2 differs and the call reverts with `ResolutionMismatch()`.
4. The guardian must cancel and propose a new M0-to-M2 resolution and wait another delay, while settlement and oracle price reads remain unavailable.

PREREQUISITES

A pending change already requires guardian resolution, a resolution proposal has been created, and `token.uiMultiplier()` changes again before `resolutionEta`. The external update must be observed through the permissionless `sync()` path.

RECOMMENDATION

Invalidate the proposal whenever `sync()` observes a new multiplier after a proposal was created:

```
if (m != pending.newMultiplier) {
    pending.newMultiplier = m;
    pending.collapsed = true;
    resolutionHash = bytes32(0);
    resolutionEta = 0;
    emit Synced(pending.oldMultiplier, m, pending.timestamp, true);
    return true;
}
```

Alternatively, reject `sync()` while `resolutionHash` is nonzero and require the guardian to execute or cancel the proposal before recording another update. The invalidation approach is generally safer because it preserves observation of the latest multiplier.

RESOLUTION

Pare

Addressed by the M-03 fix (ca0bd02): sync() now clears resolutionHash and resolutionEta and emits ResolutionCancelled whenever it mutates the pending change, which is the invalidation approach recommended here. Added a test for this exact sequence in 142a045: https://github.com/pare-audit/contracts/commit/142a045. M0 -> M1 proposed, M2 observed during the timelock, proposal cancelled explicitly, fresh M0 -> M2 proposal executes.

[L-07] Pending multiplier endpoint equality hides intermediate updates

Low Fixed

LOCATION

`MultiplierAccountant:sync:164-171`

SUMMARY

When a multiplier change is pending, `sync()` decides whether an intermediate transition occurred by comparing the current multiplier to the pending endpoint. It returns early as soon as they match:

```
if (m == pending.newMultiplier) return false;
pending.newMultiplier = m;
pending.collapsed = true;
emit Synced(pending.oldMultiplier, m, pending.timestamp, true);
```

A repeated endpoint does not prove that nothing happened in between. After an `A -> B` change is recorded, a `B -> C -> B` sequence lands back on `B`, so the equality check passes and `sync()` returns without setting `collapsed`.

The pending record stays non-collapsed, so `classifyDividend()` or `classifySplit()` finalizes only `A-> B` and never records the intermediate `B -> C -> B` actions. A net-zero sequence can also occur with no pending record at all, leaving the accountant marked synchronized while dividend or split actions are silently omitted.

IMPACT

Intermediate dividends or splits are dropped while the pending record still looks classifiable.

This corrupts `dividendIndex` and `splitFactor` state, and the omitted actions also disappear from checkpoint history. Later PT/YT settlement calculations that rely on these classified factors then produce incorrect results.

EXPLOIT PATH

1. `sync()` observes `A -> B` and creates a pending record with `oldMultiplier = A`, `newMultiplier= B`, `collapsed = false`.
2. The token changes `B -> C` before the next `sync()` call.

3. The token changes `C -> B`; the next `sync()` sees `B`, matches the pending endpoint, and returns false without setting `collapsed`.
4. The classifier finalizes `A -> B` only, leaving `B -> C -> B` absent from all accountant state and from subsequent settlement calculations.

PREREQUISITES

A non-collapsed pending change must exist, and the token must undergo at least two further multiplier updates that return to the pending endpoint before the next `sync()` call. Downstream settlement must later depend on the accountant's classified factors.

RECOMMENDATION

Do not rely on endpoint equality. Track a token-side update sequence number and compare it against the last consumed sequence, so any skipped update is detected:

```
(uint256 updateId, uint256 m) = token.multiplierState();
if (updateId == lastUpdateId) return false;
if (updateId > lastUpdateId + 1) {
    pending.collapsed = true;
}
lastUpdateId = updateId;
```

Alternatively, treat any observed change in the update identifier as invalidating ordinary classification and require guardian decomposition, even when the current multiplier equals `pending.newMultiplier`.

RESOLUTION

Pare

Fixed in 1ca353a on the audited repo: https://github.com/pare-audit/contracts/commit/1ca353a

The ERC-8056 interface exposes no update sequence number, so took the alternative using the closest identifier it does expose: effectiveAt moves on every update. sync() now stores it and, when it has changed and already taken effect since the last sync, treats that as a skipped update: a pending change re-observed at its endpoint is marked collapsed rather than returning false, and a round trip with nothing pending is recorded as a collapsed net-zero change so the guardian decomposes it. A schedule announced but not yet effective is not counted. Tests cover both exploit shapes and the no-false-positive cases.

[L-08] Per-call fee truncation permits fee-free micro-splits and YT redemptions

Low Fixed

LOCATION

`StripVault.sol:split:121-123`

SUMMARY

Both the split fee and the YT yield fee are calculated with integer division that floors the result, and there is no minimum-amount guard. For small enough amounts the fee truncates to zero:

```
uint256 fee = (amount * splitFeeBps) / BPS;
minted = amount - fee;
```

```
uint256 fee = (gross * ytYieldFeeBps) / BPS;
payout = gross - fee;
```

With `splitFeeBps = 100`, an amount of 99 atomic units gives `99 * 100 / 10000 = 0`, so no fee is taken. A caller can repeat these micro-operations through a batching contract to keep every fee rounded down to zero.

IMPACT

The treasury receives no fee on micro-splits and micro-YT redemptions even when a nonzero fee rate is configured.

The amount skipped per call is bounded by less than one atomic stock unit, so this is dust-level revenue leakage rather than theft of tracked user funds.

EXPLOIT PATH

1. A batching contract approves the vault and calls `split()` repeatedly with an amount of 99 atomic units.
2. Each call computes a fee of 0, minting 99 PT and 99 YT with no fee transferred.
3. After settlement, the caller redeems YT in chunks whose gross yield is small enough that the YT fee rounds to zero.
4. The caller receives full gross yield on those redemptions and the treasury collects nothing.

PREREQUISITES

A nonzero `splitFeeBps` or `ytYieldFeeBps` must be configured, and the caller must be able to submit repeated micro-amount operations (directly or via a batching contract) sized so the truncated fee is zero.

RECOMMENDATION

Round positive fees upward so any nonzero fee rate always charges at least one unit:

```
uint256 fee = splitFeeBps == 0
    ? 0
    : Math.ceilDiv(amount * splitFeeBps, BPS);
```

Use the analogous calculation in `redeemYT()`:

```
uint256 fee = ytYieldFeeBps == 0
    ? 0
    : Math.ceilDiv(gross * ytYieldFeeBps, BPS);
```

Alternatively, reject operations below a configured minimum amount, but the round-up approach is the smaller direct change.

RESOLUTION

Pare

Fixed in 40f79a8 on the audited repo: https://github.com/pare-audit/contracts/commit/40f79a8

redeemYT now rounds the yield fee up with Math.ceilDiv as recommended, so any non-zero rate charges at least one unit whenever there is drip, and nothing when there is none. The split-fee half of this finding is covered by the L-09 change in 14a1428, which enforces a one-unit minimum on split. Tests cover a micro YT redemption paying one unit and a zero-drip redemption paying zero.

[L-09] Per-call split-fee rounding permits fee-free fragmented deposits

Low Fixed

LOCATION

`StripVault:split:174-190`

SUMMARY

The `split` function is meant to charge `splitFeeBps` on every deposit and mint PT/YT from the remainder. The fee is rounded down independently on each call, so any deposit below the fee-unit threshold produces a zero fee while the full amount still gets minted as PT and YT.

With `splitFeeBps = 100`, a call of `split(1)` computes `floor(1 * 100 / 10000) = 0`, so `minted =1` and no fee is taken. Repeating these sub-threshold calls lets a user build up a large aggregate position while never paying the configured fee.

IMPACT

Users can avoid the configured split fee by fragmenting deposits into sufficiently small calls. The loss is limited to treasury fee revenue and does not affect user principal or vault solvency.

EXPLOIT PATH

1. User approves the vault and calls `split(1)` repeatedly.
2. Each call computes `floor(1 * 100 / 10000) = 0` and sets `minted = 1`.
3. The vault receives one stock unit and mints one PT and one YT per call.
4. The user completes a large aggregate deposit while the treasury receives zero split fees.

PREREQUISITES

The vault is before maturity, `splitFeeBps > 0`, and the stock token permits amounts smaller than the fee-unit threshold (`BPS / splitFeeBps`).

RECOMMENDATION

Apply a one-unit minimum fee whenever the configured fee is nonzero:

```
uint256 fee = amount * splitFeeBps / BPS;
if (splitFeeBps != 0 && fee == 0) fee = 1;
if (fee >= amount) revert ZeroAmount();
uint256 minted = amount - fee;
```

If a minimum fee is incompatible with token granularity, enforce a minimum deposit amount large enough to produce a nonzero fee instead. Accumulating fractional fee remainders across calls is an alternative mitigation.

RESOLUTION

Pare

Fixed in 14a1428 on the audited repo: https://github.com/pare-audit/contracts/commit/14a1428

Applied the recommended one-unit minimum: a non-zero splitFeeBps now yields at least one unit of fee, and a deposit that would be entirely fee reverts ZeroAmount. Tests cover a sub-threshold deposit paying one unit, a one-unit deposit rejected, and a normal deposit unchanged.

[L-10] Per-call YT fee rounding allows yield-fee avoidance through fragmented redemptions

Low Fixed

LOCATION

`StripVault:redeemYT:178-190`

SUMMARY

When YT holders redeem, the gross payout and the yield fee are computed independently on each call, with the fee rounding down to zero for small amounts:

```
uint256 gross = amount.mulDiv(dividendIndexAtMaturity - dividendIndex0,
dividendIndexAtMaturity);
uint256 fee = (gross * ytYieldFeeBps) / BPS;
payout = gross - fee;
if (fee > 0) stock.safeTransfer(treasury, fee);
```

A holder can pick chunk sizes whose gross payout is small enough that `(gross * ytYieldFeeBps) / BPS` truncates to zero. The YT is still burned and the full rounded gross payout is paid out, so the effective fee depends on how the redemption is fragmented rather than on the configured rate.

IMPACT

YT holders can underpay or fully avoid the configured treasury yield fee by redeeming in many small chunks instead of one aggregate call. The consequence is limited to lost fee revenue, consistent with a Low rating.

EXPLOIT PATH

1. After settlement, choose a YT chunk `x` whose calculated gross payout is positive but below the fee quantum.
2. Call `redeemYT(x)` repeatedly until the YT balance is exhausted.
3. Each call burns `x` YT and computes `fee == 0`, transferring the full gross payout to the holder and nothing to the treasury.

Compared to a single redemption of the aggregate amount, which would compute a positive fee, the treasury collects nothing.

PREREQUISITES

The series is settled, `ytYieldFeeBps > 0`, the holder has YT with a positive gross payout, and the holder can submit repeated redemption calls.

RECOMMENDATION

Reject zero-fee redemptions when a positive YT fee is configured:

```
uint256 fee = (gross * ytYieldFeeBps) / BPS;
if (ytYieldFeeBps != 0 && gross != 0 && fee == 0) revert ZeroAmount();
```

A more accounting-preserving alternative is to track a fractional fee remainder per series and carry it into subsequent fee calculations until a whole raw stock unit can be transferred.

RESOLUTION

Pare

Covered by the L-08 change (40f79a8): redeemYT now rounds the yield fee up with ceilDiv, so a chunk with positive gross always pays at least one unit and no fragmentation avoids the fee. Added a test for this sequence in d58ea16: https://github.com/pare-audit/contracts/commit/d58ea16. An aggregate redemption and the same amount redeemed in 200 small chunks; the fragmented path pays at least as much per YT.

[L-11] YT redemption fee can be bypassed through fragmented redemptions

Low Fixed

LOCATION

`StripVault:redeemYT:169-180`

SUMMARY

The YT yield fee is meant to apply to the total yield a holder redeems, regardless of how many transactions they use. But the fee is floored independently on every call, and no fractional remainder is carried across calls:

```
uint256 gross = amount.mulDiv(
    dividendIndexAtMaturity - dividendIndex0,
    dividendIndexAtMaturity
);
uint256 fee = (gross * ytYieldFeeBps) / BPS;
payout = gross - fee;
```

When a call produces a small enough `gross`, the integer division rounds the fee to zero. For example with `ytYieldFeeBps = 1000`, a chunk yielding `gross = 9` gives `floor(9 * 1000 / 10000) = 0`. A holder can repeat such sub-granularity calls to redeem their full position while every individual fee rounds to zero.

The same rounding-based bypass applies to any fee that shares this design. Split fees and YT yield fees are separate fee domains, so remainder accounting must be applied to each independently.

IMPACT

The treasury can receive less than the configured YT yield fee, potentially zero, when a holder fragments redemption into small enough calls.

The shortfall is bounded to fractional-token dust per call and is not theft of another user's funds, but it is fully user-controlled and repeatable, making it a real fee-policy bypass. A holder can also spread one position across multiple calls or addresses to keep each fee under the rounding threshold.

EXPLOIT PATH

1. A settled vault has `ytYieldFeeBps = 1000` and the holder controls YT worth at least 9,000 gross stock units.
2. The holder picks chunks each producing `gross = 9`, for which the fee floors to zero.
3. The holder issues 1,000 sequential `redeemYT(chunkAmount)` calls, burning all the corresponding YT.
4. The holder receives 9,000 stock units and the treasury receives nothing, versus 900 units in a single aggregate redemption.

PREREQUISITES

The vault is settled, `ytYieldFeeBps > 0`, and the holder has enough YT and redemption granularity to produce multiple small positive gross payouts.

RECOMMENDATION

Carry the fractional fee entitlement across calls with a running remainder so rounding cannot be reset per call:

```
uint256 feeNumerator = gross * ytYieldFeeBps + feeRemainder;
uint256 fee = feeNumerator / BPS;
feeRemainder = feeNumerator % BPS;
```

Apply the same remainder accounting separately to split fees and YT yield fees, since they are distinct fee domains.

Alternatively, enforce a minimum redemption whose gross payout yields at least one fee unit. Note this changes the interface for small deposits, whereas remainder accounting preserves support for them:

```
if (ytYieldFeeBps != 0) {
    require(gross * ytYieldFeeBps >= BPS, "redemption below fee granularity");
}
```

RESOLUTION

Pare

Covered by the L-08 change (40f79a8) and tested for this exact fragmented path under L-10 (d58ea16). redeemYT now rounds the fee up with ceilDiv, so every chunk with positive gross pays at least one unit and no chunk size avoids the fee; the L-10 test redeems the same amount in one call and in 200 small chunks and shows the fragmented path pays no less per YT. We chose round-up over a carried remainder because it closes the bypass with no new state; the difference is at most one unit per call in the treasury's favour.

Info Severity

[I-01] Multiplier processing can become unavailable because deployment accepts a zero administrator

Info Fixed

SUMMARY

The `MultiplierAccountant` constructor grants `DEFAULT_ADMIN_ROLE` to the supplied `admin` without rejecting `address(0)` or assigning a fallback administrator. The AccessControl implementation in use permits this internal role grant.

Classifier and guardian roles start empty, and granting them requires administrator authority. If the deployer supplies a zero administrator, no usable caller can ever populate those roles.

The deployment script reads `ADMIN` without validating that it is nonzero. Its documented step to grant classifier and guardian roles after deployment cannot be completed in this configuration through the ordinary role-management API, and there is no in-place administrative recovery path.

IMPACT

Once a subsequent multiplier update produces pending state, no authorized classifier or guardian exists to process it.

Dependent unsettled vaults cannot settle while the accountant is unsynchronized, and an oracle using that accountant cannot return `price()`. Paired merge remains available.

Deploying a replacement accountant does not automatically migrate existing vaults' immutable references or outstanding claims. This requires a deployment mistake; an outsider cannot impose a zero administrator on a correctly initialized accountant, and no live affected instance exists.

RECOMMENDATION

Reject `admin == address(0)` in the constructor and validate the deployment script's `ADMIN` input before deployment.

Also confirm that the chosen nonzero administrator can actually exercise its authority and complete the documented role grants; a nonzero-address check alone does not prove those operational properties.

RESOLUTION

Pare

Fixed in 1ca0298 on the audited repo: https://github.com/pare-audit/contracts/commit/1ca0298

Constructor now reverts ZeroAdmin on address(0); the deploy script requires ADMIN before deploying. On the second point, agreed: the nonzero check is a guard, not proof the admin can act. The role grants are exercised as part of the deploy runbook against the chosen admin before any series is created.

[I-02] Fees remain trapped and the deposit cap can be exceeded because the vault accepts itself as treasury

Info Fixed

SUMMARY

`StripVault` rejects a zero treasury address but permits `treasury_ == address(this)`. This breaks the fee assumption in `split`: the cap check only accounts for the net minted amount because the fee is expected to leave the vault.

With an underlying token that supports ordinary balance-neutral ERC-20 self-transfers, forwarding the fee to the vault itself leaves the full gross deposit in custody. The function still mints only the net amount and never verifies that the fee actually left, so the final balance can end up above `depositCap` rather than briefly exceeding it mid-transaction.

For example, an empty vault configured with a cap of 999 whole raw stock-token units, a 10-bps split fee, and its own address as treasury. Before maturity a user deposits 1000 units. The fee is 1 and the minted amount is 999, so the cap check passes. The vault receives 1000 units, transfers the fee to itself without reducing its balance, and mints 999 PT and 999 YT, ending the transaction holding 1000 units against a cap of 999.

IMPACT

The configured cap can be violated at transaction completion, and the fee stays in vault custody with no corresponding PT/YT claims.

If the depositor merges all 999 claim pairs, the vault returns 999 units and retains the 1-unit fee. The contract has no independent treasury withdrawal or stock-recovery function, so that fee cannot be recovered through the ordinary API without claims to burn.

The same recipient alias can also retain yield fees during `redeemYT`. Deposited fees remain as surplus rather than becoming additional claim entitlements, so users do not receive unbacked claims or extract extra assets.

RECOMMENDATION

Reject `treasury_ == address(this)` in the constructor alongside the existing zero-address check, and validate the expected fee recipient at deployment before accepting deposits.

If keeping fees inside the vault is intended, add explicit fee accounting with an authorized withdrawal path and include retained fees in the cap calculation. Simply sending the fee before the net deposit does not fix this, since both transfers still reach the same vault.

RESOLUTION

Pare

Fixed in b4020bd on the audited repo: https://github.com/pare-audit/contracts/commit/b4020bd

The constructor now rejects treasury_ == address(this) alongside the zero-address check. Fees are meant to leave the vault, so no in-vault fee accounting was added. Test predicts the deployment address and confirms construction reverts.

[I-03] Guardian recovery can be delayed because malformed proposals overwrite valid resolutions

Info Fixed

SUMMARY

`MultiplierAccountant::proposeResolution` does not validate that the `kinds` and `ratiosWad` arrays are the same length before storing their hash and updating `resolutionEta`. The length check only exists in `executeResolution`, so the contract will accept a proposal that can never execute, and it will accept it as a replacement for a valid, already-executable proposal.

For example, a stable pending change has an observed ratio of `1.05e18`. The guardian correctly proposes `kinds = [Dividend]` and `ratiosWad = [1.05e18]`, which becomes executable after the configured two-day delay. Before execution, an operator mistake submits a replacement with `kinds = [Dividend,Split]` but only `ratiosWad = [1.05e18]`.

The malformed replacement is accepted, overwrites the valid proposal hash, and sets a new ETA. Executing the original arrays now fails commitment matching, and executing the malformed arrays fails the length check once the new ETA elapses. Submitting the corrected arrays starts a fresh two-day delay, and no API restores the original executable proposal or its elapsed ETA.

IMPACT

A guardian input mistake can discard an executable recovery proposal and extend the window during which the accountant stays unsynchronized.

For a pending 5% change under the supplied 3% dividend and 20% split thresholds, ordinary classification cannot clear the change, so recovery depends on a corrected guardian proposal completing its delay. Dependent unsettled vaults and oracle price reads remain blocked in the meantime.

When there is no existing valid proposal, early detection and immediate correction avoid the wait, since the contract does not require the guardian to wait before replacing or cancelling. The unavoidable restart in the example specifically comes from overwriting a proposal that was already executable.

RECOMMENDATION

Reject `kinds.length != ratiosWad.length` in `proposeResolution` before modifying `resolutionHash` or `resolutionEta`. Keep the execution-side length check as a defensive validation.

RESOLUTION

Pare

Fixed in 4bfdaae on the audited repo: https://github.com/pare-audit/contracts/commit/4bfdaae

proposeResolution now validates that kinds, ratiosWad and timestamps have equal length before storing the hash and eta, reverting with ResolutionMismatch otherwise. A malformed replacement proposal can no longer overwrite a valid one. The length check in executeResolution is unchanged. Test added: test_I03_proposeRejectsMismatchedArrays.

[I-04] Timestamp freshness accepts economically stale feed answers

Info Acknowledged

SUMMARY

`price()` trusts a Chainlink answer after only two checks:

```
if (answer <= 0) revert BadFeedAnswer(answer);
if (maxFeedAge != 0 && block.timestamp - updatedAt > maxFeedAge) revert StaleFeed(updatedAt);
```

The problem is that `updatedAt` only proves the feed published a round, not that the price in that round is current. Chainlink feeds refresh `updatedAt` on every heartbeat even when the answer has not moved, and the contract's own NatSpec notes that Robinhood equity feeds hold their last print while US markets are closed.

A held, days-old equity print is therefore republished with a fresh timestamp and passes the freshness gate unchanged. The same gate cannot distinguish a halted upstream source from a live but flat market.

The deployed oracle sets `maxFeedAge = 4 days` so a long weekend does not brick the market. This makes the only staleness bound four days wide, with nothing behind it: no deviation bound, no round-completeness check, no secondary feed, and no circuit breaker.

IMPACT

For up to four days the pSPY/USDG Morpho market can value collateral at a price the underlying equity no longer trades at.

If the stock gaps down while the market is closed or the source is halted, `price()` keeps returning the pre-gap value. Borrowers can hold and open positions against collateral worth materially less than reported, and liquidators have no economically correct price to act on.

At the deployed 62.5% LLTV, a gap exceeding the remaining collateral buffer leaves debt above the recoverable stock value, and the shortfall lands on the market's lenders as bad debt.

RECOMMENDATION

Add a deviation or source-liveness check, or a corroborating feed, and treat a held print as stale. A refreshed `updatedAt` is not evidence of a current price.

RESOLUTION

Pare

Acknowledged... We agree a refreshed updatedAt only proves the feed published a round, not that the price is current.

We are not changing the oracle in this fix window. A stateless "unchanged answer" check would treat every weekend and market holiday as stale and brick the market, which is the failure mode L-01 addressed, and there is currently no second SPY feed or market-hours feed on Robinhood Chain to run a deviation or corroboration check against. A stock cannot gap while its market is closed; it gaps at the open, when the feed prints the new price, so the residual exposure is a halted upstream source during trading hours. That is mitigated by the 62.5% LLTV buffer and will be documented as an operational risk in the market docs. If a corroborating feed or market-status feed becomes available on the chain we will add a liveness check in a follow-up audit.

[I-05] Immutable feed-decimal snapshot becomes stale after same-address feed migration

Info Fixed

SUMMARY

The constructor reads the feed's precision once and bakes it into an immutable scaling constant:

```
uint8 feedDecimals = IAggregatorV3(feed_).decimals();
...
scaleFactor = 10 ** (num - den);
```

`price()` fetches a fresh `answer` on every call but keeps scaling it by that deploy-time constant and never re-reads `decimals()`.

A Chainlink feed address is a proxy over a swappable aggregator, so the answer's precision is a runtime property while the scale applied to it is fixed forever, and nothing reconciles the two.

If the aggregator behind the configured address is migrated and starts publishing at a different precision, the mismatch is silent: no revert and no staleness signal, because the answer is still positive and still fresh. A same-address 8 to 18 decimal migration makes `price()` return exactly 1e10x the correct value, permanently.

IMPACT

Every subsequent price is wrong by a power of ten for the remaining life of the market.

Overvaluation lets borrowers extract far more USDG than the stock collateral is worth and removes any liquidation incentive. Undervaluation liquidates every healthy position at once.

Because the oracle and the Morpho market are both immutable, the error cannot be paused or patched. It persists until a replacement oracle and market are deployed and liquidity migrates.

RECOMMENDATION

Re-read `decimals()` inside `price()` and fail closed on any change from the deploy-time value.

RESOLUTION

Pare

Fixed in 90bb509 on the audited repo: https://github.com/pare-audit/contracts/commit/90bb509

The deploy-time feed decimals are now stored in the immutable feedDecimals, and price() re-reads feed.decimals() on every call, reverting with FeedDecimalsChanged if the value differs. The execution-time check fails closed so a same-address aggregator migration to a different precision can no longer produce a silently mis-scaled price. Test added: test_I05_feedDecimalsChangeFailsClosed.

[I-06] Chainlink feed can remain stale while the onchain stock market remains active

Info Acknowledged

SUMMARY

`PareMorphoOracle.price()` uses Chainlink's `latestRoundData()` and treats the feed as valid as long as `updatedAt` is within `maxFeedAge`. This freshness check only proves the feed hasn't gone silent for too long; it cannot tell whether the underlying equity market has moved while the feed stopped publishing new prices.

Robinhood's Stock Tokens stay tradable onchain 24/7, but their underlying assets follow asset-specific trading sessions, including a distinct all-day / 24-5 tradability flag. This means the onchain market can remain active during hours when the reference feed is not being updated.

IMPACT

When the Chainlink reference price stops updating but is still within `maxFeedAge`, the oracle keeps valuing PT collateral at the old price.

If the underlying market has materially repriced in the meantime, borrowers can draw more or less credit than the current market value justifies.

RECOMMENDATION

Do not rely on `maxFeedAge` alone for assets whose reference price can stop updating while the onchain market stays active. Consider incorporating a secondary onchain or reference-market price to value collateral during non-publishing periods.

RESOLUTION

Pare

Acknowledged. We agree maxFeedAge alone cannot tell a quiet feed from a moved market.

We are not changing the oracle in this fix window. There is currently no secondary SPY price source on Robinhood Chain to corroborate against: the only other onchain price is the PT/stock pool, which prices the PT relative to the stock token rather than the stock in dollars, so it cannot stand in for the reference feed. The underlying equity only reprices when its market trades, and the feed publishes when that happens, so the residual exposure is a halted feed during trading hours. That is mitigated by the 62.5% LLTV buffer and will be documented as an operational risk in the market docs. If a corroborating feed becomes available on the chain we will add it in a follow-up audit.

[I-07] Accountant sync can clear while the Robinhood price feed remains deliberately paused

Info Fixed

SUMMARY

`PareMorphoOracle.price()` only checks `accountant.isSynced()` before accepting the Chainlink price. The accountant's sync state tracks whether `uiMultiplier()` changes have been classified; it does not track the Stock Token issuer's `oraclePaused()` state.

Robinhood documents that during corporate actions the issuer can pause the oracle, holding the feed at its last known-good value until the corporate action and multiplier are confirmed aligned. Because these two states are independent, the accountant can report synced while the issuer feed remains deliberately paused.

IMPACT

The oracle can resume serving prices immediately after the accountant classifies a multiplier change, even though Robinhood's authoritative feed is still intentionally frozen.

During the window between multiplier classification and feed unpause, Morpho can value PT collateral using a stale pre-corporate-action price.

RECOMMENDATION

Have the oracle additionally verify the Stock Token's `oraclePaused()` state and revert while the issuer feed is paused.

RESOLUTION

Pare

Fixed in 5b86a98 on the audited repo: https://github.com/pare-audit/contracts/commit/5b86a98

The oracle now stores the PT's stock token and price() reads its oraclePaused() flag on every call, reverting with IssuerFeedPaused while the issuer holds its feed. This check is separate from the accountant's isSynced() gate, so classifying a multiplier change no longer resumes pricing while Robinhood's feed is still frozen. Test added: test_I07_issuerOraclePauseBlocksPrice.

[I-08] The oracle can bypass the PT vault's synchronization gate because its accountant is independently configured

Info Fixed

SUMMARY

`PareMorphoOracle` saves the accountant passed to its constructor without verifying that it is the same accountant the configured PT's vault actually uses. The PT exposes its vault, and the vault exposes its accountant, so the correct relationship is knowable at deploy time but is never enforced.

Consider two correctly implemented accountants A and B that track the same stock token. The PT vault uses A, but the oracle is deployed with B plus the correct PT/stock pool. After a genuine multiplier update, B's classifier processes the event while A stays pending. With otherwise valid feed and pool inputs, the oracle's synchronization check passes against B even though the configured PT's vault still has unresolved accounting in A.

The existing check that the pool contains the PT and the accountant's stock does not catch this, because both accountant instances track the same stock. No malicious accountant or mismatched stock token is required.

IMPACT

The oracle's synchronization circuit breaker can reflect the state of an unrelated accountant instance, allowing price reads while the PT vault's own accountant is unsynchronized.

The accountant reference is immutable, so fixing the wiring requires deploying a new oracle.

RECOMMENDATION

Derive the accountant from the authentic PT vault, or reject deployment unless the supplied accountant equals `PT.vault().accountant()`.

Keep the existing deployment checks for the authentic PT, vault, and intended pool: trusting a counterfeit token's self-reported vault would not establish authenticity.

RESOLUTION

Pare

Fixed in 7402c70 on the audited repo: https://github.com/pare-audit/contracts/commit/7402c70

The oracle constructor now reads accountant() from the configured vault and reverts with AccountantMismatch unless the supplied accountant is that same address. The synchronization gate can therefore only ever reflect the accountant the PT's vault settles against. The existing PT, vault and pool checks are unchanged. Test added: test_I08_accountantMustMatchTheVaults.

[I-09] Split tolerance can absorb a real dividend without crediting YT

Info Acknowledged

SUMMARY

`classifySplit()` accepts an observed multiplier ratio when it is close enough to a declared clean split ratio, then assigns the entire multiplier change to `splitFactor` and leaves `dividendIndex` unchanged.

The validator only checks the distance from the declared split ratio:

```
uint256 target = (num * WAD) / den;
uint256 diff = r > target ? r - target : target - r;
if (diff > splitRatioToleranceWad) revert NotASplit(r);
```

Once that check passes, the full observed ratio is recorded as split growth and no residual is credited to `dividendIndex`:

```
splitFactor = splitFactor.mulDiv(p.newMultiplier, p.oldMultiplier);
_finalize(ActionKind.Split, p);
```

So if a single multiplier update combines a genuine split with a small real dividend, and that dividend falls inside `splitRatioToleranceWad`, the dividend is silently treated as principal rather than yield.

IMPACT

YT holders receive none of the yield contained in such a combined update; the vault assigns the full balance to PT instead.

For an observed ratio of `2.000001` against a declared 2:1 split, the difference is exactly the configured `1e12` tolerance, so the update is accepted. The clean split is `2.0`, leaving a residual dividend factor of

1. `0000005`. For a position of 1,000 raw stock tokens the correct YT entitlement is `499,999,750,000,124`

wei, but the implementation keeps `dividendIndex` at `1e18` and pays YT zero.

The loss is bounded by the tolerance and is only meaningful when the residual represents real yield rather than rounding noise.

RECOMMENDATION

After validating that the observed ratio is within tolerance of the declared split, do not attribute the full observed multiplier change to `splitFactor`. Split the change: apply the exact clean split ratio to `splitFactor`, and credit any residual (the observed ratio divided by the clean split ratio) to `dividendIndex` so real yield in a combined update reaches YT holders.

RESOLUTION

Pare

Acknowledged. We agree that a residual inside the split tolerance is attributed entirely to splitFactor.

We are keeping this behaviour. splitRatioToleranceWad is 1e12, one millionth of the ratio, while the smallest real dividend on a covered stock is on the order of 1e15, a thousand times larger. A single update combining a genuine split with a real dividend therefore cannot pass the clean-ratio check; it is rejected by classifySplit and can only be resolved through the guardian decomposition, which applies the exact split ratio as a Split leg and the residual as a checkpointed Dividend leg, as recommended. Any residual small enough to sit inside the tolerance is rounding in the issuer's multiplier rather than yield, and crediting it would add a dust checkpoint on every split. We will document the tolerance and this reasoning in the accountant's parameter notes.

[I-10] Shape-only split classification routes value-additive clean-ratio events entirely into splitFactor, zeroing the YT drip at settlement

Info Fixed

SUMMARY

`classifySplit` validates the shape of a multiplier step, never that the underlying event was value-neutral. `_validateSplitRatio` (src/MultiplierAccountant.sol:297-307) only checks the ratio is at least `minSplitDeltaBps` away from 1e18, that its terms stay under `maxSplitTerm`, and that it matches the declared p/q within tolerance.

A stock dividend declared as a shares-per-share rate produces a ratio that is structurally a clean p/q: rate 0.25 gives exactly 5:4, rate 1.0 gives 2:1. These land inside the deployed 2000 bps split band (script/Deploy.s.sol:38) yet outside the dividend band, whose 300 bps cap rejects r = 1.25e18 because 25% exceeds it (src/MultiplierAccountant.sol:177). So the tag is forced: `classifyDividend` reverts, and the only in-band classification left writes `splitFactor` while leaving `dividendIndex` and the checkpoint list untouched. The Split path never pushes a checkpoint, unlike the Dividend path (:179) and the guardian path (:243).

The same pending state can also be resolved correctly as `[Dividend 1.25e18]` through the guardian timelock, which only checks that the decomposition's product reproduces the observed ratio (src/MultiplierAccountant.sol:216-250). But a classifier call supersedes a live guardian proposal outright (src/MultiplierAccountant.sol:312-314), so even a guardian who proposes the correct dividend treatment is overridden by the Split tag.

This defeats the documented band-separation guarantee (src/MultiplierAccountant.sol:29-31) that a keeper can never tag a real dividend as a split and steal the drip from YT: a clean-ratio dividend above 20% does exactly that, on shape evidence alone.

IMPACT

Each mis-tagged event transfers `minted * (r - 1)/r` of series notional from the YT cohort to the PT cohort, where `r` is the event ratio and `minted` is the series split notional.

Because the Split tag writes only `splitFactor` (src/MultiplierAccountant.sol:190-191), the dividend index stays flat and every series maturing after the event settles at dT = d0 (src/StripVault.sol:154-155). `redeemPT` then pays `amount * d0/dT = amount` in full (:165) while `redeemYT` pays `amount * (dT - d0)/dT = 0` (:175). The correct dividend treatment of the same ratio would instead pay YT `amount * (r - 1)/r` and PT `amount / r`.

The transfer scales with r: 20% of notional at a 5:4 event, 50% at 2:1, 66.7% at 3:1, approaching the whole event as r grows. On a 100-stock split at the deployed 10 bps fee (99.9 PT and 99.9 YT minted, src/StripVault.sol:120-121), a single 5:4 event moves 19_980_000_000_000_000_000 wei (19.98 raw stock) between cohorts.

Every live series maturing after the event suffers the transfer simultaneously, and settlement makes it permanent: dT is frozen at settle (src/StripVault.sol:155) with no post-settlement correction path. The vault stays solvent throughout, payouts always sum exactly to holdings, so nothing reverts and no alarm fires. The difference between the two outcomes is pure mis-attribution, not accounting necessity.

EXPLOIT PATH

1. The issuer processes a value-additive clean-ratio event: `uiMultiplier` moves 1e18 to 1.25e18 with per-share price unchanged, so per-raw-token total-return value rises 25%. Any caller syncs it.
2. `classifyDividend` reverts `NotADividend(1.25e18)`; `classifySplit(5, 4)` succeeds (25% >= 20%, terms 5 and 4 <= 20, ratio within the 1e12 tolerance).
3. `splitFactor` absorbs the step; `dividendIndex` and the checkpoints stay flat. A guardian's correct `[Dividend 1.25e18]` proposal can be overridden because a classifier call supersedes a live guardian resolution.
4. Every affected series settles with dT = d0, so `redeemYT` pays 0 and `redeemPT` pays full principal. A keeper positioned long PT, or paid by PT holders, repeats this per event.

PREREQUISITES

Requires the keeper (CLASSIFIER_ROLE) acting maliciously or mistakenly. The README explicitly scopes a compromised keeper as an in-scope actor (attention item 4). No band violation or revert is needed; the routing is a routine, in-permissions tag choice.

RECOMMENDATION

The class of a multiplier step cannot be inferred from its ratio shape; only the issuer's declaration carries it. Gate the immediate Split tag on the declared corporate-action type. The ERC-8056 schedule that `sync()` already reads for `effectiveAt` (src/MultiplierAccountant.sol:328-337) carries the action class, so route declared value-additive types through the timelocked guardian decomposition even when the ratio is band-legal for Split:

```
 function classifySplit(uint256 num, uint256 den) external onlyRole(CLASSIFIER_ROLE) {
     PendingChange memory p = _loadClassifiablePending();
     uint256 r = _ratioWad(p);
     _validateSplitRatio(r, num, den);
+    if (_declaredActionType() != ActionType.ForwardSplit) revert NotASplit(r);
     splitFactor = splitFactor.mulDiv(p.newMultiplier, p.oldMultiplier);
     _finalize(ActionKind.Split, p);
 }
```

Alternatively, make a Split classification provisional for the same `guardianDelay` reaction window the guardian path already uses, during which the guardian can reattribute the leg into `dividendIndex` and holders can exit via the always-free `merge()`. Either way, shape alone must not settle a decision that moves value between holder cohorts.

RESOLUTION

Pare

Fixed in e4be0c6 on the audited repo: https://github.com/pare-audit/contracts/commit/e4be0c6

We took the second recommended approach. classifySplit is now provisional: it schedules a fold into splitFactor behind the same guardianDelay reaction window the Dividend tag already uses, rather than writing splitFactor immediately. During that window holders can exit via merge() and the guardian can call the new reattributeFoldAsDividend(), which folds the step into dividendIndex with its checkpoint instead. Settlement already refuses while any fold is pending, so no series can settle on an unattributed clean-ratio step. Both reattribute functions reject the wrong fold kind. Tests added: test_I10_splitTagIsProvisional, test_I10_guardianReattributesSplitAsDividend, test_I10_reattributeRespectsKindAndWindow.

[I-11] Immediate Multiplier Updates Can Be Missed at the Maturity Boundary

Info Fixed

SUMMARY

`StripVault.settle()` permanently freezes the maturity dividend index using whatever state is visible when the settlement transaction executes:

```
accountant.sync();
if (!accountant.isSynced()) revert AccountantNotSynced();
uint256 dT = accountant.dividendIndexAt(maturity);
dividendIndexAtMaturity = dT;
settled = true;
```

Per Chainlink's Robinhood Tokenized Equities documentation, Robinhood updates the multiplier through two paths: an immediate `updateMultiplier(uint256)` and a scheduled `updateMultiplier(uint256, uint256 effectiveAt)`. The docs note that small updates, including routine dividend reinvestments, are applied immediately through automated processes (https://docs.chain.link/data-feeds/tokenized-equity-feeds/robinhood).

If an immediate multiplier update is ordered after `settle()` within a block whose `block.timestamp` equals `maturity`, the vault freezes the old dividend index before observing the new dividend. Because the vault is already settled, that later same-timestamp dividend can never be included in YT settlement.

IMPACT

A dividend that becomes effective exactly at the vault's maturity timestamp can be excluded from the YT claim.

YT holders then receive less stock than their maturity-period dividend entitlement, while PT holders receive excess stock through `redeemPT()`. Because settlement is immutable, the incorrect allocation is permanent.

RECOMMENDATION

Disallow settlement at the exact maturity timestamp, so any dividend effective at `maturity` is observed before the dividend index is frozen.

RESOLUTION

Pare

Fixed in 9b39d35 on the audited repo: https://github.com/pare-audit/contracts/commit/9b39d35

settle() now requires block.timestamp to be strictly greater than maturity instead of greater or equal, so a multiplier update applied later in the maturity block can no longer be excluded from the frozen maturity index. Test added: test_I11_settleRevertsAtExactMaturity.

[I-12] Four-Day Feed Freshness Window Accepts Stale Prices During Active Trading

Info Acknowledged

SUMMARY

The oracle treats a Chainlink price as fresh whenever it falls inside the configured `maxFeedAge` window:

```
block.timestamp - updatedAt <= maxFeedAge
```

The deployed `maxFeedAge` is set to roughly four days so the oracle can tolerate weekends and market closures. The AAPL Chainlink feed, however, has a 24 hour heartbeat.

Because the acceptance window is much wider than the feed heartbeat, during active trading hours the oracle can keep accepting a price that has already exceeded the feed's expected update interval by up to about three additional days.

IMPACT

If the AAPL/SPY Chainlink feed stops updating during active trading and the underlying equity price drops, Morpho may value PT collateral using an outdated, inflated price.

This can lead to:

- excessive borrowing against PT collateral;
- delayed liquidation of unhealthy positions;
- potential bad debt for Morpho lenders.

RECOMMENDATION

Adopt a freshness policy that distinguishes expected market closures (weekends, holidays) from unexpected feed inactivity, rather than applying a single wide `maxFeedAge` window that also covers normal trading hours.

RESOLUTION

Pare

Acknowledged. We agree a single maxFeedAge window sized for weekends is wider than the feed's heartbeat during trading hours.

We are not changing the oracle in this fix window. A session-aware freshness policy needs an onchain market calendar or a market-status feed, and neither is available on Robinhood Chain today; a hardcoded calendar would misfire on holidays and early closes and brick the market each time. The residual exposure is a feed that halts during trading hours, which is mitigated by the 62.5% LLTV buffer and will be documented as an operational risk in the market docs. If Chainlink publishes a market-status feed for these assets on the chain we will gate freshness on it in a follow-up audit.

[I-13] accruedDripFactor exposes unsynchronized live index instead of a settlement-compatible value

Info Fixed

LOCATION

`StripVault:accruedDripFactor:192-195`

SUMMARY

The drip-factor view is meant to return a value consistent with the accountant state the vault actually uses for settlement and redemption. While the accountant is unsynced, that state is not yet valid for settlement.

The implementation returns `accountant.dividendIndex()` for every unsettled series without checking `isSynced()`. When a multiplier change is pending, the view keeps reporting the old classified index, while `settle()` rejects that same state and later relies on a maturity checkpoint (`accountant.dividendIndexAt(maturity)`), which can differ.

Redemption remains governed by the settled checkpoint, so this cannot be used to move funds; the effect is a stale or misleading view result.

IMPACT

Integrators and users can read a dividend factor that does not correspond to the eventual maturity checkpoint or redemption value. This can drive incorrect UI projections and integration decisions.

The view itself cannot mint, burn, or transfer funds, so the impact is limited to display and integration correctness.

EXPLOIT PATH

1. A multiplier update becomes effective and anyone calls `accountant.sync()`, creating a pending change.
2. An integrator calls `vault.accruedDripFactor()`; because the vault is unsettled, it reads the unchanged `accountant.dividendIndex`.
3. The integrator calls `settle()`, which reverts because `accountant.isSynced()` is false.
4. After classification, settlement uses `accountant.dividendIndexAt(maturity)`, which may differ from the previously displayed factor.

PREREQUISITES

The accountant has an unclassified pending multiplier change, or the series is unsettled and its current index differs from the historical maturity checkpoint, and the caller relies on `accruedDripFactor()` for a redemption or yield estimate.

RECOMMENDATION

Reject the view while the accountant is unsynced so callers cannot mistake a provisional value for a valid one:

```
function accruedDripFactor() external view returns (uint256) {
    if (!accountant.isSynced()) revert AccountantNotSynced();
    uint256 d = settled ? dividendIndexAtMaturity : accountant.dividendIndex();
    return d.mulDiv(1e18, dividendIndex0);
}
```

Alternatively, return `(factor, valid)` and set `valid = accountant.isSynced()` so off-chain consumers must explicitly handle the provisional state.

RESOLUTION

Pare

Fixed in 0b92ec3 on the audited repo: https://github.com/pare-audit/contracts/commit/0b92ec3

accruedDripFactor() now reverts with AccountantNotSynced while the accountant has an unclassified pending change, matching the gate settle() applies, so the view can no longer return a provisional index that settlement would reject. Test added: test_I13_accruedDripFactorRevertsWhileUnsynced.

[I-14] Disabled feed-age validation accepts unfinished Chainlink rounds

Info Fixed

LOCATION

`PareMorphoOracle:price:78-84`

SUMMARY

The oracle validates only the sign of the Chainlink `answer` and checks `updatedAt` conditionally, based on `maxFeedAge`. When `maxFeedAge` is zero, a response with `updatedAt == 0` is accepted without any further validation, and a positive `answer` is incorporated into the returned price.

Disabling the maximum-age policy is meant to disable only the freshness threshold, not the check that the returned round has actually completed. As a result, an unfinished or invalid round (positive answer, `startedAt == 0`, `updatedAt == 0`) passes through as an authoritative price.

IMPACT

An invalid positive feed answer from an unfinished round is accepted and propagated into Morpho collateral valuation.

This can distort borrowing power and liquidation thresholds until the external feed recovers, potentially permitting excess borrowing or triggering incorrect liquidations.

EXPLOIT PATH

1. Oracle is deployed with `maxFeedAge == 0`.
2. The configured feed returns `answer > 0` with `startedAt = 0` and `updatedAt = 0`.
3. A Morpho market operation calls `price()`, which accepts the response and returns a scaled valuation.
4. Morpho runs risk calculations on the invalid value, permitting excess borrowing or an incorrect liquidation.

PREREQUISITES

Requires the oracle to be deployed with `maxFeedAge == 0` and the configured feed to return a positive answer with `updatedAt == 0`.

RECOMMENDATION

Reject zero and future timestamps regardless of `maxFeedAge`:

```
if (updatedAt == 0 || updatedAt > block.timestamp) {
    revert StaleFeed(updatedAt);
}
if (maxFeedAge != 0 && block.timestamp - updatedAt > maxFeedAge) {
    revert StaleFeed(updatedAt);
}
```

For explicit unfinished-round validation, also retain and validate `startedAt`:

```
(, int256 answer, uint256 startedAt, uint256 updatedAt,) = feed.latestRoundData();
if (startedAt == 0 || updatedAt == 0) revert StaleFeed(updatedAt);
```

RESOLUTION

Pare

Fixed in f57b464 on the audited repo: https://github.com/pare-audit/contracts/commit/f57b464

price() now rejects a round whose updatedAt is zero or in the future with StaleFeed before the maxFeedAge check runs, so disabling the age policy no longer disables round-completeness validation. The same check is applied to the optional loan feed with StaleLoanFeed. Tests added: test_I14_unfinishedRoundRefusedWhenAgeCheckDisabled, test_I14_unfinishedLoanRoundRefused.

[I-15] Independent PT and YT redemption rounding strands matched-position dust

Info Acknowledged

LOCATION

`StripVault:redeemPT/redeemYT:183-207`

SUMMARY

Equal PT and YT claims are meant to jointly redeem exactly the raw stock amount deposited into the vault. The PT claim represents the baseline fraction and the YT claim the complementary dividend fraction, so their gross payouts should conserve the deposited amount.

The implementation floors both components independently. For `d0 = 1e18`, `dT = 3e18`, and `amount =1`, both `amount.mulDiv(d0, dT)` and `amount.mulDiv(dT - d0, dT)` evaluate to zero. Both tokens are still burned, leaving the corresponding stock stranded in the vault.

There is no remainder accounting or sweep mechanism that preserves the holders' residual value, so the loss is permanent.

IMPACT

Matched PT/YT holders can lose up to one raw stock unit per equal redemption when the complementary floor operations both round down. The loss is dust-sized per operation, but it is repeatable across many redemptions and is never returned to the claim holders.

EXPLOIT PATH

1. A user calls `split(1)` and receives one PT and one YT against one raw stock unit held by the vault.
2. The series settles with `dividendIndex0 = 1e18` and `dividendIndexAtMaturity = 3e18`.
3. The user calls `redeemPT(1)` and `redeemYT(1)`; each token is burned but both computed gross payouts floor to zero.
4. The user receives no stock, and the raw stock unit remains stranded in the vault.

PREREQUISITES

A settled series whose complementary redemption fractions do not divide evenly, plus a holder with matching PT and YT claims.

RECOMMENDATION

Make one complementary component round up so the gross PT/YT pair conserves the original amount:

```
uint256 gross = Math.ceilDiv(
    amount * (dividendIndexAtMaturity - dividendIndex0),
    dividendIndexAtMaturity
);
```

Alternatively, maintain an explicit dust/remainder balance and distribute it under a documented policy rather than silently retaining it:

```
uint256 exact = amount * numerator;
uint256 payout = exact / denominator;
remainder += exact % denominator;
```

RESOLUTION

Pare

Acknowledged. We agree that flooring both redemption legs independently can strand up to one raw unit per matched PT/YT redemption.

We are keeping the floor on both sides. Rounding one leg up makes the gross pair conserve the deposit for a single holder, but the two legs are redeemed independently and in different chunkings by different holders; a YT cohort redeeming in many small amounts would then draw up to one raw unit per redemption above its exact share, and the vault, which holds only the exact total, could fall short by a few units and revert the final PT redemption. Flooring both legs keeps the vault solvent under every redemption pattern. The stranded amount is bounded by one raw unit (1e-18 of a stock token) per matched redemption and is not recoverable by anyone, so we consider a remainder ledger and sweep policy more surface than the value warrants. We will document the rounding behaviour in the vault's redemption notes.

[I-16] No recovery path for residual stock after all PT and YT liabilities are extinguished

Info Fixed

LOCATION

`StripVault:redeemPT:163-173, StripVault:redeemYT:175-187`

SUMMARY

The vault is meant to be fully wound down once all PT and YT have been merged or redeemed, with no stock left behind. In practice PT and YT payouts are rounded independently, so the sum of the two floored payouts can be smaller than the raw amount backing the burned PT/YT pair.

That difference leaves stock in the vault after the underlying liabilities are gone. The vault also accepts direct ERC-20 transfers, which can add balance that no PT or YT supply accounts for.

Once the last PT and YT are destroyed, `merge()`, `redeemPT()`, and `redeemYT()` are all unusable to recover the leftover balance because each of them requires tokens to burn. With no terminal sweep or close path, the residual stock is stranded.

IMPACT

Protocol-accounted stock dust can remain permanently locked after all PT and YT have been destroyed. The stuck amount is generally limited to rounding dust plus any unsolicited direct transfers.

EXPLOIT PATH

1. Users split stock into equal PT/YT and the series later settles with a nonzero dividend index increase.
2. Holders redeem PT and YT in fragmented amounts; each payout floors independently, leaving a positive raw-stock remainder.
3. The final PT and YT balances are burned via `redeemPT()`, `redeemYT()`, or `merge()`.
4. The vault still holds the remainder, but every stock-transfer function requires PT/YT to burn, so the balance is locked permanently.

PREREQUISITES

A settled vault whose redemption rounding leaves a positive residual, or an external stock transfer that increased the vault balance, followed by all PT and YT liabilities being burned.

RECOMMENDATION

Add a finalization function callable only when both PT and YT supply are zero, transferring the remaining stock according to the protocol's documented policy:

```
function sweepResidual(address to) external {
    if (pt.totalSupply() != 0 || yt.totalSupply() != 0) revert OutstandingLiabilities();
    stock.safeTransfer(to, stock.balanceOf(address(this)));
}
```

If unsolicited donations must not be treated as protocol funds, track accounted stock separately and sweep only the surplus above it:

```
uint256 accounted = /* remaining liability-backed stock */;
uint256 surplus = stock.balanceOf(address(this)) - accounted;
stock.safeTransfer(to, surplus);
```

RESOLUTION

Pare

Fixed in 12b2e53 on the audited repo: https://github.com/pare-audit/contracts/commit/12b2e53

Added sweepResidual(), which reverts with OutstandingLiabilities unless both PT and YT total supply are zero and otherwise transfers the vault's full stock balance to the immutable treasury and emits ResidualSwept. The recipient is fixed to the treasury rather than caller-supplied so the call can remain permissionless without letting the first caller choose where residual stock goes. Tests added: test_I16_sweepResidualAfterAllClaimsBurned, test_I16_sweepRefusedWhileClaimsOutstanding.

[I-17] Oracle Constructor Does Not Validate That the Pool Contains the Accountant's Stock Token

Info Fixed

LOCATION

`PareMorphoOracle:constructor:91-93`

SUMMARY

The oracle is meant to accept only a Uniswap v3 pool pairing the configured PT with the exact stock token tracked by the accountant, since the TWAP is later interpreted specifically as a PT-versus-stock price.

The constructor records whether PT is token0 but never checks token1, and it never compares either pool token to `accountant.token()`. As a result, a PT/unrelated-token pool passes construction and its unrelated TWAP gets used with the stock feed.

The oracle price then becomes unrelated to the protocol's actual PT/stock market, and the resulting market is unsafe if the configuration is incorrect or the unrelated pool is manipulable.

IMPACT

A misconfigured Morpho market can use an unrelated PT/token TWAP as though it were PT/stock, producing arbitrary collateral valuations and potentially causing bad debt or unjustified liquidations.

EXPLOIT PATH

1. A pool contains PT and an unrelated ERC-20 rather than the accountant's stock token.
2. The oracle is deployed with that pool, the legitimate stock feed, and a synced accountant; construction succeeds.
3. `price()` interprets the unrelated pair's TWAP as PT per stock and multiplies it by the stock feed.
4. Morpho values PT using the fabricated ratio, potentially enabling bad debt or incorrect liquidations.

PREREQUISITES

A PareMorphoOracle and Morpho market are deployed with a pool that does not contain the accountant's stock token (a misconfiguration at deployment).

RECOMMENDATION

Validate both pool assets against PT and the accountant's stock token during construction.

```
address stock = address(accountant_.token());
address t0 = IUniswapV3PoolMinimal(pool_).token0();
address t1 = IUniswapV3PoolMinimal(pool_).token1();
if (!((t0 == pt_ && t1 == stock) || (t1 == pt_ && t0 == stock))) {
    revert InvalidPool();
}
ptIsToken0 = t0 == pt_;
```

Alternatively, add an explicit stock-token constructor parameter, require it to match `accountant.token()`, then validate the pool against that address.

RESOLUTION

Pare

Fixed in c0a3c1a on the audited repo: https://github.com/pare-audit/contracts/commit/c0a3c1a

The constructor now reads token0 and token1 from the pool and reverts with InvalidPool unless one is the PT and the other is the stock token under the PT's vault; ptIsToken0 is derived from that validated pair. Test added: test_I17_poolMustPairPtWithStock.

[I-18] Oracle does not unconditionally reject incomplete Chainlink rounds

Info Fixed

LOCATION

`PareMorphoOracle:price:106-116`

SUMMARY

The oracle reads only `answer` and `updatedAt` from the Chainlink feed and ignores the round identifiers. A positive answer alone is treated as valid, but that is not sufficient when the feed response carries incomplete-round metadata.

When freshness checking is disabled, an `updatedAt == 0` response does not revert and its positive answer flows into the PT price calculation. A legacy superseded response can likewise pass if its timestamp satisfies `maxFeedAge`.

This requires malformed or legacy feed behavior; it is not a permissionless manipulation of the oracle.

IMPACT

Morpho may consume a price derived from an incomplete or superseded feed response when the feed's round metadata is invalid and freshness checking is disabled or insufficient, leading to incorrect collateral valuation.

EXPLOIT PATH

1. The configured feed returns `(roundId, positiveAnswer, ..., updatedAt = 0, ...)` while `maxFeedAge` is configured as zero.
2. A Morpho action calls `PareMorphoOracle.price()`.
3. `price()` accepts the response because `answer > 0` and no age check is active.
4. The invalid answer is multiplied by the PT/stock TWAP and used for Morpho's collateral valuation.

PREREQUISITES

The configured feed must return a positive response with invalid round metadata, and `maxFeedAge` must be disabled or fail to reject it. No permissionless attacker can manufacture this response through `PareMorphoOracle` itself; it depends on malformed or legacy feed behavior.

RECOMMENDATION

Reject incomplete responses independently of `maxFeedAge`:

```
(uint80 roundId, int256 answer,, uint256 updatedAt,) = feed.latestRoundData();
if (roundId == 0 || updatedAt == 0) revert InvalidFeedRound();
if (answer <= 0) revert BadFeedAnswer(answer);
```

For deployments that explicitly support legacy aggregators, add a separately documented `answeredInRound >= roundId` check. Do not add it solely for modern OCR feeds, where the field is deprecated.

RESOLUTION

Pare

Fixed in 0fbfea8 on the audited repo: https://github.com/pare-audit/contracts/commit/0fbfea8

price() now reads the round id from latestRoundData on both the stock feed and the optional loan feed and reverts with InvalidFeedRound when it is zero. A zero or future updatedAt is already rejected on the current head independently of maxFeedAge, so an incomplete round is now refused on either signal. We did not add an answeredInRound check, per the note that it is deprecated on modern OCR feeds. Test added: test_I18_zeroRoundIdRefused.

[I-19] Post-maturity accountant changes can block settlement of an otherwise mature vault

Info Fixed

LOCATION

`StripVault:settle:149-158`

SUMMARY

A mature vault should be able to freeze its `dividendIndexAtMaturity` and unlock redemption using the accountant checkpoint at the vault's maturity timestamp.

Instead, `settle()` first synchronizes the entire shared accountant and requires `isSynced()` to be true. A multiplier change whose effective timestamp is after the vault's maturity therefore blocks settlement, even though it cannot affect the historical index returned by `dividendIndexAt(maturity)`.

For example, a vault matures at time T while the accountant is synced. At T plus seven days a new dividend becomes effective and a separate `accountant.sync()` records it as pending. When a holder later calls `settle()`, the pending change makes `isSynced()` false and the call reverts, despite the maturity index already being historically determined.

IMPACT

A mature but unsettled vault can be blocked from settling and redeeming PT/YT because an unrelated, later stock-token multiplier change is pending in the shared accountant.

The state is not permanently corrupted, but users must wait for a keeper or guardian to classify or resolve the pending change before they can settle and redeem.

This particularly hurts holders without equal PT and YT balances, since they cannot fall back on the equal-amount merge path as an alternative exit.

EXPLOIT PATH

1. A vault matures at time T while the accountant is synchronized, but no user calls `settle()`.
2. At T plus seven days a new dividend becomes effective and a separate call to `accountant.sync()` records it as pending.
3. A PT holder calls `vault.settle()`; the call's sync sees the pending change, `isSynced()` is false, and the transaction reverts.
4. Settlement and both redemption functions stay unavailable until the post-maturity change is classified or resolved.

PREREQUISITES

The vault has matured but is not yet settled, and a subsequent multiplier change has become effective and remains pending in the shared accountant.

RECOMMENDATION

Allow settlement to use the maturity checkpoint without requiring later accountant changes to be finalized. Expose a historical settlement read that excludes changes after the queried timestamp:

```
function dividendIndexAtOrBefore(uint256 timestamp)
    external view returns (uint256);
function settle() external nonReentrant {
    if (block.timestamp < maturity) revert NotMatured();
    if (settled) revert AlreadySettled();
    dividendIndexAtMaturity = accountant.dividendIndexAtOrBefore(maturity);
```

```
    settled = true;
}
```

The accountant implementation must ensure the returned checkpoint is valid for the maturity timestamp and that pending changes after maturity are excluded from this vault's snapshot.

RESOLUTION

Pare

Fixed in 796d475 on the audited repo: https://github.com/pare-audit/contracts/commit/796d475

Added MultiplierAccountant.indexFinalAt(timestamp), which returns true only when no pending change's window and no waiting fold's earliest leg can reach the queried timestamp. settle() now requires indexFinalAt(maturity) rather than a fully synced accountant, then freezes dividendIndexAt(maturity) as before. A change that becomes effective after maturity therefore no longer blocks settlement, while a change whose window can include maturity still does. Tests added: test_I19_postMaturityChangeDoesNotBlockSettle, test_I19_postMaturityFoldDoesNotBlockSettle.

[I-20] Repeated dividend classification introduces cumulative downward index rounding drift

Info Acknowledged

LOCATION

`MultiplierAccountant:classifyDividend/classifySplit:174-193`

SUMMARY

The `dividendIndex` is meant to represent the exact cumulative product of all classified dividend ratios. Each classification instead updates the already-rounded index with a floored multiply-divide, discarding the fractional remainder:

```
dividendIndex = dividendIndex.mulDiv(
    p.newMultiplier,
    p.oldMultiplier
);
_pushCheckpoint(p.timestamp, dividendIndex);
```

Because the discarded remainder is never carried forward, repeated valid classifications can drive the index below the exact telescoping product. Each transition loses at most a fraction of a WAD-scaled unit, but the loss is persistent and compounds across many updates.

The same floor-rounded update pattern is used for `splitFactor`, which is subject to analogous drift.

IMPACT

YT maturity payouts can be lower than the mathematically exact cumulative dividend factor. The understated denominator slightly increases PT allocation and decreases YT allocation, leaving the corresponding stock residual stranded in the vault.

The per-transition error is generally below one WAD-scaled unit, but it accumulates across many valid updates.

EXPLOIT PATH

1. A multiplier change is observed and `sync()` records exact `oldMultiplier` and `newMultiplier` values.
2. A classifier calls `classifyDividend()`, which floors `dividendIndex * newMultiplier / oldMultiplier` and stores the result.
3. This repeats across many independent dividend changes, each starting from the previously floored index.
4. At maturity, `redeemYT()` uses the lower checkpoint-derived index and pays less than the exact cumulative dividend claim.

PREREQUISITES

Multiple valid dividend updates must be classified sequentially, with at least one producing a non-integral fixed-point multiplication.

RECOMMENDATION

Store the cumulative dividend factor as a higher-precision rational value and derive the WAD index from it:

```
uint256 public dividendNumerator;
uint256 public dividendDenominator;
```

Update the numerator/denominator with full-precision arithmetic per transition. If a rational accumulator is impractical, retain the division remainder and feed it into the next update rather than dropping it:

```
(uint256 q, uint256 r) = Math.mulDiv(dividendIndex, newMultiplier, oldMultiplier,
Math.Rounding.Floor);
dividendIndex = q;
dividendRemainder = r;
```

Apply the same treatment to `splitFactor`.

RESOLUTION

Pare

Acknowledged. We agree each classification floors once and the discarded remainder is not carried forward.

We are keeping the floored update. The loss is bounded by one unit at 1e-18 scale per classification, and a series sees on the order of ten dividends over its life, so the cumulative understatement of dividendIndex is a few units in the eighteenth decimal, well below the smallest transferable amount of stock for any realistic position. A rational accumulator would overflow within a few updates without a reduction step, and a carried remainder has to be rescaled across each update's different denominator; both add nontrivial arithmetic to the core index for an attowei-level improvement, which we judge a worse trade than the drift. Flooring also keeps the index conservative in the vault's favour, so the vault can never owe more than it holds. We will document the rounding direction and bound in the accountant notes.

[I-21] Split classification stores the rounded observed ratio instead of the validated clean ratio

Info Fixed

LOCATION

`MultiplierAccountant:classifySplit:186-191`

SUMMARY

The `splitFactor` is meant to be the cumulative product of the clean stock split ratios supplied during classification. Tolerance exists only to absorb small discrepancies between the observed ERC-8056 multiplier and the declared corporate-action ratio.

After validation, `classifySplit` ignores `num` and `den` and multiplies `splitFactor` by the observed multiplier ratio instead:

```
uint256 r = _ratioWad(p);
_validateSplitRatio(r, num, den);
splitFactor = splitFactor.mulDiv(p.newMultiplier, p.oldMultiplier);
```

Any accepted nonexact ratio causes `splitFactor` to diverge from the product of the classified ratios.

IMPACT

`splitFactor` can accumulate rounding or tolerance deviations from the clean split ratios accepted by classification. This is a real accounting/API inconsistency, but it does not presently alter StripVault redemption because `splitFactor` is unused there.

EXPLOIT PATH

1. The token multiplier changes so that `newMultiplier / oldMultiplier` is slightly below the exact 3:2 ratio.
2. Anyone calls `sync()`, creating a pending change.
3. The classifier calls `classifySplit(3, 2)`, and validation accepts the ratio within tolerance.
4. `splitFactor` is updated with the rounded observed ratio, leaving it different from `splitFactorOld* 3 / 2`.

PREREQUISITES

A pending multiplier change must pass split-ratio validation with an observed ratio within tolerance of, but not exactly equal to, the supplied `num:den` ratio.

RECOMMENDATION

Use the declared ratio after validation:

```
_validateSplitRatio(r, num, den);
splitFactor = splitFactor.mulDiv(num, den);
```

If the protocol instead intends `splitFactor` to track observed multiplier changes, revise its documentation and downstream consumers to use that meaning explicitly.

RESOLUTION

Pare

Fixed in bf2f92a on the audited repo: https://github.com/pare-audit/contracts/commit/bf2f92a

classifySplit now records the validated num:den on the scheduled fold, and applying the fold multiplies splitFactor by that declared ratio rather than by the observed multiplier ratio. splitFactor is therefore the exact cumulative product of the classified clean ratios; the tolerance only decides whether the change is accepted. Test added: test_I21_splitFactorUsesDeclaredRatio.

[I-22] Unsettled accruedDripFactor includes dividends occurring after maturity

Info Fixed

LOCATION

`StripVault:accruedDripFactor:182-185`

SUMMARY

A series-specific view is meant to report the dividend factor for that series, using `dividendIndex0` as its baseline and stopping accrual at the series maturity.

Before settlement, the implementation instead reads the accountant's global `dividendIndex`. That accumulator keeps advancing for dividends that affect the stock after the vault's maturity, while `settle()` uses the historical `dividendIndexAt(maturity)` checkpoint. As a result the two functions report different terminal factors for the same series.

For example, a vault matures with `dividendIndex0 = 1.00e18` and a maturity checkpoint of `1.05e18`. A dividend classified after maturity advances the global `dividendIndex` to `1.10e18`, so the view returns

1. `10e18` even though the maturity checkpoint stays at `1.05e18`. Once `settle()` runs it freezes the factor

at `1.05e18`. The view is therefore time-dependent after maturity, temporarily rising with later-classified dividends and then dropping at settlement, and it can exceed the eventual settlement factor.

IMPACT

Integrators using `accruedDripFactor()` before settlement can overestimate the series' terminal dividend entitlement and misprice PT or YT.

External pricing, market-making, lending, and risk systems can act on this inflated pre-settlement value even though redemption later uses the lower maturity checkpoint. The view does not itself transfer funds, so this is primarily an integration and reporting risk, but it exposes a materially inconsistent accounting value.

EXPLOIT PATH

1. A vault reaches maturity with `dividendIndex0 = 1.00e18` and a maturity checkpoint of `1.05e18`, but `settle()` has not been called.
2. A post-maturity dividend is classified, advancing the global accountant `dividendIndex` to `1.10e18`.
3. An integrator calls `accruedDripFactor()` and receives `1.10e18`, above the true maturity checkpoint of `1.05e18`.
4. `settle()` is called and freezes the factor at `1.05e18`, so redemptions use a lower yield factor than the view previously exposed.

PREREQUISITES

The vault has reached maturity but has not yet been settled, the accountant classifies a dividend whose checkpoint timestamp is after maturity, and an external integrator relies on `accruedDripFactor()` for pricing or risk calculations.

RECOMMENDATION

Use the maturity-bounded checkpoint whenever the series is not settled:

```
function accruedDripFactor() external view returns (uint256) {
    uint256 d = settled
        ? dividendIndexAtMaturity
        : accountant.dividendIndexAt(maturity);
    return d.mulDiv(1e18, dividendIndex0);
}
```

If the view is intentionally meant to expose live global accrual, rename it and add a separate maturity-bounded function so integrators cannot confuse the two meanings.

RESOLUTION

Pare

Fixed in 5861650 on the audited repo: https://github.com/pare-audit/contracts/commit/5861650

accruedDripFactor() now reads accountant.dividendIndexAt(maturity) while the series is unsettled, so the reported factor stops accruing at maturity and matches the checkpoint settle() will freeze. Before maturity the value is unchanged. Test added: test_I22_accruedDripFactorBoundedAtMaturity.

[I-23] Untracked token donations can permanently exhaust StripVault deposit capacity

Info Fixed

LOCATION

`StripVault.sol:split:117-133`

SUMMARY

The deposit cap is meant to limit stock backing protocol-issued PT/YT claims, so direct token transfers to the vault should count as untracked excess rather than consuming capacity. Instead, the cap check uses the vault's entire ERC-20 balance as its baseline.

Because any token holder can transfer stock directly to the vault address without calling `split()`, the raw balance can be inflated without creating any corresponding claims. Once that balance reaches the cap, the strict greater-than check rejects every positive deposit.

For example, with `depositCap = 10,000e18`, a donation of `10,000e18` sent straight to the vault makes a later `split(100e18)` observe `10,000e18 + 99e18` and revert, even though no PT/YT was ever minted against the donated tokens.

IMPACT

An attacker can disable all future deposits and PT/YT issuance for a capped vault by donating enough stock to reach `depositCap`.

The donated tokens carry no PT/YT ownership record, so merge and redemption cannot identify or withdraw them. With no rescue path, the cap stays exhausted for the vault's lifetime. Even an empty vault with zero outstanding liabilities can be bricked this way. This is a permissionless griefing vector that disables the vault's primary issuance function even while protocol-created exposure remains below the cap.

EXPLOIT PATH

1. A vault is deployed with `depositCap = 10,000e18` and a maturity weeks in the future.
2. The attacker transfers `10,000e18` stock tokens directly to the vault address instead of calling `split()`.
3. A legitimate user calls `split(100e18)`; the cap check sees the inflated balance and reverts.
4. The donation created no PT/YT and no rescue path exists, so no further split can succeed before maturity.

PREREQUISITES

A nonzero `depositCap` is configured, the vault is before maturity, and the attacker can transfer at least `depositCap` minus the vault's current stock balance to the vault's stock token address. No special role is required.

RECOMMENDATION

Track live principal with an internal accounting variable instead of the externally mutable ERC-20 balance:

```
uint256 livePrincipal;
if (depositCap != 0 && livePrincipal + minted > depositCap) {
    revert DepositCapExceeded();
}
livePrincipal += minted;
```

Decrement `livePrincipal` whenever `merge()`, `redeemPT()`, or `redeemYT()` transfers accounted principal out of the vault, so capacity is restored only when the corresponding stock liability is retired. Any balance above `livePrincipal` is then ignored by the cap and can optionally be exposed through a controlled excess-recovery function.

RESOLUTION

Pare

Fixed in f3009a6 on the audited repo: https://github.com/pare-audit/contracts/commit/f3009a6

The cap check in split() now compares pt.totalSupply() + minted against depositCap instead of the vault's ERC-20 balance. PT is minted one per raw stock unit and burned in lockstep with YT by merge, and split is closed after maturity, so PT supply is exactly the live principal the recommendation's livePrincipal counter would track, without new state. Stock transferred directly to the vault backs no claims and no longer consumes capacity; residual balance remains recoverable through sweepResidual() once all claims are burned. Test added: test_I23_donationDoesNotConsumeCapacity.

[I-24] Zero guardian delay removes the configured resolution reaction window

Info Fixed

LOCATION

`MultiplierAccountant:constructor:118-143`

SUMMARY

Guardian resolutions are meant to have a public delay so holders can inspect a proposed decomposition and exit through merge before it takes effect. The delay is stored immutably so it cannot be shortened after deployment.

The constructor accepts zero without a guard. With `guardianDelay == 0`, `proposeResolution` sets `resolutionEta` to `block.timestamp`, while `executeResolution` allows execution whenever the current timestamp is not earlier than `resolutionEta`. The guardian can therefore propose and execute in immediate succession, removing the reaction window for every resolution in that deployment.

IMPACT

A deployment configured with `guardianDelay == 0` gives holders no time to review a guardian decomposition or exit through merge before it changes dividend accounting.

This is a configuration-dependent loss of the intended safety window, driven by a trusted guardian, not a permissionless theft path.

EXPLOIT PATH

1. Deploy `MultiplierAccountant` with `guardianDelay_ = 0`.
2. With a pending gap-band or collapsed multiplier change, the guardian calls `proposeResolution(kinds, ratiosWad)`.
3. `resolutionEta` is set to `block.timestamp`, so the guardian calls `executeResolution` with the matching arguments immediately afterward.
4. The resolution updates `dividendIndex` or `splitFactor` with no public interval for holders to review and merge out.

PREREQUISITES

The accountant must be deployed with `guardianDelay_ == 0`, and the action requires the trusted guardian role.

RECOMMENDATION

Reject a zero delay in the constructor:

```
error InvalidGuardianDelay();
if (guardianDelay_ == 0) revert InvalidGuardianDelay();
guardianDelay = guardianDelay_;
```

RESOLUTION

Pare

Fixed in 11f413d on the audited repo: https://github.com/pare-audit/contracts/commit/11f413d

The constructor now reverts with InvalidGuardianDelay when guardianDelay_ is zero, so the timelock on guardian resolutions and the reaction window on provisional classifications can never be configured away at deployment. Test added: test_I24_constructorRejectsZeroGuardianDelay.