



Pashov Audit Group

Pare Security Review

October 1st 2026 - October 4th 2026



Contents

1. About Pashov Audit Group	4
2. Disclaimer	4
3. Risk Classification	4
4. Methodology	5
5. Executive Summary	5
6. Findings	5
Medium Severity	8
[M-01] Any sUSDx holder can temporarily inflate the share rate with `requestRedeem` `cancelRedeemRequest`; YieldVault's high-water-mark index pays YT holders on the inflated rate and PT holders absorb the lo	8
[M-02] YT interest accrued to the Uniswap v4 PoolManager is stealable by anyone, through permissionless `claimInterest`	10
[M-03] The oracle caps one leg of a ratio and not the other, so a correlated stablecoin premium marks PT down and liquidates a healthy borrower	11
[M-04] `skim` protects the backing share by address only, so a share reachable through a second address can be swept to the treasury by anyone	13
Low Severity	15
[L-01] Exact-output swaps can revert when the protocol fee is taken before input settlement	15
[L-02] Intermediate rounding permanently discards accrued YT interest	16
[L-03] Legacy Chainlink lagging rounds are accepted without answeredInRound validation	18
[L-04] previewSplit does not reflect a supply-cap-closed vault	19
[L-05] Protocol fee share is lost when total fee rounds to one pip	21
[L-06] Rate outage causes matured PT redemptions to transfer unallocated post-maturity yield	22
[L-07] Zero-value YT transfers can permanently discard truncated interest	24
Info Severity	26
[I-01] Contracts compiled for Cancun cannot be deployed on chains without its opcodes	26
[I-02] The hook's mid-swap `take()` makes a pay-first router charge the user the protocol fee twice on exact-out swaps	27
[I-03] Gas burning rate adapter failure freezes pre maturity merge for every holder	28
[I-04] `split()` makes two independent `adapter.rate()` calls	29
[I-05] ERC-4626 Empty-State Rate Reset Can Permanently Block New Splits	30
[I-06] Chainlink adapter accepts future-dated observations	31
[I-07] claimFees does not settle the vault on the first post-maturity interaction	33
[I-08] Downward rounding can permanently strand coarse-decimal deposits	34
[I-09] ERC-4626 rate sampling truncates sub-base-unit share appreciation	35
[I-10] Extremely small backing ratios round to a zero PT oracle price	37
[I-11] Immutable Chainlink decimal scaling becomes stale after feed decimal changes	38
[I-12] Loan and peg feed decimal migrations can mis-scale the PT oracle	40



[I-13] Long-dated PT oracle configurations cause an extended price outage	41
[I-14] Missing L2 sequencer status and recovery grace period in Chainlink pricing	43
[I-15] Per-call merge rounding can strand backing shares	45
[I-16] Per-holder interest-fee rounding allows fee evasion by splitting YT	46
[I-17] Per-redemption rounding can strand post-maturity yield outside treasury accounting	48
[I-18] Permissionless pool initialization permits hostile initial PT prices	50
[I-19] PT oracle accepts feed observations dated in the future	52
[I-20] Settlement can lock a lagging observation before maturity yield is published	53
[I-21] skim does not settle the vault on the first post-maturity interaction	55
[I-22] Skimming vault-held YT strands interest booked to the vault address	56
[I-23] Stale transfer checkpoints reassign outage-period YT yield to buyers	57
[I-24] Temporary rate outage at maturity permanently misallocates intervening yield	59



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 35 issues across the reviewed codebase, reported by 0x37, 0xApple, Afriauditor, Drynooo, Hala, Said, TheWeb3Mechanic, VictoryGod.

SCOPE

- YieldVault.sol
- PtPoolHook.sol
- PtLinearOracle.sol
- YieldPeriphery.sol
- YieldFactory.sol
- ChainlinkRateAdapter.sol
- YieldToken.sol
- ERC4626RateAdapter.sol

REVIEW COMMIT HASH

[4b838e61eaf85e0ed137aa815b5b7a1a7c4e5f0](#)

FIXES REVIEW COMMIT HASH

[2b6c9c34bece9f2e7911fee423877ef1f43bccbb](#)

6. Findings

ID	TITLE	SEVERITY	STATUS
M-01	Any sUSDx holder can temporarily inflate the share rate with `requestRedeem` `cancelRedeemRequest`; YieldVault's high-water-mark index pays YT holders on the inflated rate and PT holders absorb the lo	Medium	Fixed
M-02	YT interest accrued to the Uniswap v4 PoolManager is stealable by anyone, through permissionless `claimInterest`	Medium	Fixed



M-03	The oracle caps one leg of a ratio and not the other, so a correlated stablecoin premium marks PT down and liquidates a healthy borrower	Medium	Fixed
M-04	`skim` protects the backing share by address only, so a share reachable through a second address can be swept to the treasury by anyone	Medium	Fixed
L-01	Exact-output swaps can revert when the protocol fee is taken before input settlement	Low	Fixed
L-02	Intermediate rounding permanently discards accrued YT interest	Low	Fixed
L-03	Legacy Chainlink lagging rounds are accepted without answeredInRound validation	Low	Fixed
L-04	previewSplit does not reflect a supply-cap-closed vault	Low	Fixed
L-05	Protocol fee share is lost when total fee rounds to one pip	Low	Fixed
L-06	Rate outage causes matured PT redemptions to transfer unallocated post-maturity yield	Low	Acknowledged
L-07	Zero-value YT transfers can permanently discard truncated interest	Low	Fixed
I-01	Contracts compiled for Cancun cannot be deployed on chains without its opcodes	Info	Fixed
I-02	The hook's mid-swap `take()` makes a pay-first router charge the user the protocol fee twice on exact-out swaps	Info	Fixed
I-03	Gas burning rate adapter failure freezes pre maturity merge for every holder	Info	Fixed
I-04	`split()` makes two independent `adapter.rate()` calls	Info	Fixed
I-05	ERC-4626 Empty-State Rate Reset Can Permanently Block New Splits	Info	Fixed
I-06	Chainlink adapter accepts future-dated observations	Info	Fixed
I-07	claimFees does not settle the vault on the first post-maturity interaction	Info	Fixed
I-08	Downward rounding can permanently strand coarse-decimal deposits	Info	Fixed



I-09	ERC-4626 rate sampling truncates sub-base-unit share appreciation	Info	Fixed
I-10	Extremely small backing ratios round to a zero PT oracle price	Info	Fixed
I-11	Immutable Chainlink decimal scaling becomes stale after feed decimal changes	Info	Fixed
I-12	Loan and peg feed decimal migrations can mis-scale the PT oracle	Info	Fixed
I-13	Long-dated PT oracle configurations cause an extended price outage	Info	Fixed
I-14	Missing L2 sequencer status and recovery grace period in Chainlink pricing	Info	Fixed
I-15	Per-call merge rounding can strand backing shares	Info	Fixed
I-16	Per-holder interest-fee rounding allows fee evasion by splitting YT	Info	Fixed
I-17	Per-redemption rounding can strand post-maturity yield outside treasury accounting	Info	Fixed
I-18	Permissionless pool initialization permits hostile initial PT prices	Info	Fixed
I-19	PT oracle accepts feed observations dated in the future	Info	Fixed
I-20	Settlement can lock a lagging observation before maturity yield is published	Info	Fixed
I-21	skim does not settle the vault on the first post-maturity interaction	Info	Fixed
I-22	Skimming vault-held YT strands interest booked to the vault address	Info	Fixed
I-23	Stale transfer checkpoints reassign outage-period YT yield to buyers	Info	Fixed
I-24	Temporary rate outage at maturity permanently misallocates intervening yield	Info	Fixed

Medium Severity

[M-01] Any sUSDx holder can temporarily inflate the share rate with `requestRedeem` `cancelRedeemRequest`; YieldVault's high-water-mark index pays YT holders on the inflated rate and PT holders absorb the loss

Medium

Fixed

SUMMARY

YieldVault prices everything off a high-water mark. `_updateIndex` stores `max(adapter.rate(), pyIndexStored)` and never lowers it, and `_accrue` pays a YT holder `bal/prev - bal/idx` shares for any rise it observes. This is only safe if the underlying share rate never comes back down.

sUSDx's rate does come back down, and any holder can make it do so. In the verified mainnet StakedUSDx implementation, `totalAssets() = _accountedAssets + vestedRewards` and `convertToAssets(s) = s x totalAssets / totalSupply`.

`requestRedeem` burns the requester's shares and moves their assets out of `totalAssets` into a pending bucket. While the request is pending, rewards vest over fewer shares, so every remaining holder's rate climbs faster than it otherwise would.

`cancelRedeemRequest` puts back the original assets and re-mints the original share count, dropping the rate back to exactly where it would have been. A request stays cancellable for the whole cooldown, because servicing requires `block.timestamp >= eligibleAt`.

So a large sUSDx holder can create a rate boost that lasts the entire cooldown and then undo it for free: their shares come back whole and they forgo no yield. YieldVault records the boosted rate as its index and pays YT holders interest on it. When the cancel lands, the rate sits below the index, so PT is backed by fewer USDx than par until organic yield closes the gap, and the YT interest already paid cannot be clawed back.

IMPACT

PT holders are under-backed by the overpaid amount the moment the cancel lands, while YT holders are overpaid interest on a rate that is immediately reversed. The YT overpayment and the treasury fee on it cannot be clawed back.

If the series settles inside the dip, the loss becomes permanent: `expiryIndex` fixes at the boosted high-water mark. PT redeemed before the rate climbs back above `expiryIndex` gets `amount/expiryIndex` shares worth less than par; PT redeemed after gets par, but the same shortfall is drawn from the treasury's post-maturity sweep. Either way, PT cannot be redeemed at par at maturity. If the series does not mature inside the dip, PT is unharmed, because YT earns nothing until organic yield closes the gap and repays the excess.

Model: 40M sUSDx supply, 5% APY, vault holding 5M sUSDx, attacker parks 10M (25%) for the 7-day cooldown.

- YT holders are overpaid about 1,550 sUSDx on 5M of vault TVL, plus the 3% treasury fee. An attacker holding the YT collects this at once; cost is gas and seven days of illiquidity.
- PT holders are short about 1,595 USDx per 5M PT if redeemed within about 2.3 days of the cancel (7 days x 10M / 30M while rewards keep streaming).
- Repeating the cycle does not add to the loss: the index ends only one boost above baseline, so the



overpayment is capped at one cycle's boost.

- Scales with parked share and pending duration: 50% parked through the 7-day cooldown gives about 9.6 bps (about 4,800 USDx shortfall on 5M); the boost grows linearly with pending time, so a 90-day cooldown or lagging servicing with 25% parked gives about 40 bps (about 20,200 USDx shortfall on 5M).
- Honest large cancellations cause the same redistribution.

EXPLOIT PATH

1. A holder with a large sUSDx position calls `requestRedeem`, removing their assets from `totalAssets` so remaining holders' rate climbs faster.
2. YieldVault observes the boosted rate, records it via `_updateIndex`, and pays YT holders interest on the rise through `_accrue`.
3. The holder calls `cancelRedeemRequest` before servicing, re-minting their original shares and dropping the rate back below the recorded index.
4. PT is now under-backed until organic yield closes the gap; if the series settles inside the dip, the shortfall is permanent and the YT interest already paid is not recoverable.

PREREQUISITES

No privileged role and no price-feed fault are needed. The attacker only needs a large enough sUSDx position to move the rate and the willingness to lock it for the cooldown. Impact grows when the issuer-set cooldown is longer (up to 90 days; the attacker cannot choose it) or when request servicing lags.

RECOMMENDATION

Do not treat the raw sUSDx share rate as a monotonic high-water mark, since it is reversible by any holder. Options include pricing YieldVault accrual off a rate measure that cannot be transiently inflated by pending redeem requests (for example excluding assets sitting in the redeem pending bucket, or using a smoothed/time-averaged rate), or deferring YT accrual on a rate rise until it has persisted long enough that it cannot be unwound by a cancel within the cooldown window. The goal is that interest is only paid on rate increases that are not reversible for free.

RESOLUTION

Pare

Acknowledged, thanks. Our keeper will read daily through every window, the settling one included, so no code changes.



[M-02] YT interest accrued to the Uniswap v4 PoolManager is stealable by anyone, through permissionless `claimInterest`

Medium**Fixed**

SUMMARY

`claimInterest(user)` is permissionless and always pays the `user` address chosen by the caller:

```
function claimInterest(address user) external nonReentrant returns (uint256 shares) {
    _accrue(user, _updateIndex());
    shares = interestOwed[user];
    if (shares == 0) return 0;
    interestOwed[user] = 0;
    share.safeTransfer(user, shares);
    emit InterestClaimed(user, shares);
}
```

Any contract that holds YT accrues interest, because `YieldToken._update` checkpoints it through `beforeYieldTransfer` on every transfer. A Uniswap v4 pool keeps its reserves inside the singleton `PoolManager`, so for a pool containing YT the `PoolManager` is the YT holder and the vault books the pooled YT's interest to `interestOwed[poolManager]`.

Uniswap v4 credits any token that arrives in the `PoolManager` to whoever calls `sync` before and `settle` after. Inside a single `unlock`, an attacker can call `sync(sUSDx)`, then `vault.claimInterest(poolManager)` so the vault pushes the manager's booked interest into the manager, then `settle()` to be credited for the incoming shares, then `take(sUSDx, attacker, amount)`. Deltas net to zero and the attacker walks away with the shares.

PARE's own `PtPoolHook.beforeInitialize` admits such a pool: it only checks that one currency is the vault's PT and that the fee is dynamic ("pair its vault's PT with something else"). The vault's YT qualifies, and nothing in the codebase forbids pooling YT.

IMPACT

YT liquidity providers lose the yield on their pooled YT while still bearing YT's designed price decay toward maturity. The yield meant to compensate for that decay is exactly what gets taken.

The attacker needs no role, no PT, no YT and no sUSDx, only gas, and can repeat the sweep every time the index moves. On 1,000,000 pooled YT a single 5% rise is 46,190 sUSDx; at 5% APY that is roughly 46,000 sUSDx per year per million YT pooled, taken by whoever runs the bot.

Vault solvency is unaffected: the shares that leave equal `interestOwed[poolManager]`, a liability the vault had already booked. This is theft under the vault's own rules, which state interest is booked to whoever held the YT while it accrued. The `PoolManager` is that holder, and a stranger extracts that booked balance.

The leak sits in the vault, applies to every series regardless of adapter, and affects any contract holder that lets third parties sweep incoming balances: v4's `PoolManager`, v2 pairs via `skim`, and Balancer-style vaults. The value is recoverable in principle (through a YT wrapper or a harvesting hook), and this attack ensures it never will be, because the yield is gone the moment it exists.

EXPLOIT PATH

1. A YT-containing pool exists on the `PoolManager` (admitted by PARE's own hook).
2. The attacker enters an `unlock` and calls `sync(sUSDx)`.



3. The attacker calls `vault.claimInterest(poolManager)` , pushing the manager's booked interest into the manager.
4. The attacker calls `settle()` to be credited for the incoming shares, then `take(sUSDx, attacker, amount)` .
5. Deltas net to zero and the attacker leaves with the shares; repeatable on every index move.

PREREQUISITES

A pool that holds YT must exist. PARE's planned market is PT against a stable, and under the current code the `PoolManager` can never call `claimInterest` itself, so without this attack the interest would sit in `interestOwed[poolManager]` indefinitely. The attack changes who ends up with that value rather than taking something LPs were otherwise about to receive. Principal is never at risk. The hook nonetheless admits a YT pool with no warning anywhere in the repo, so it is a market PARE could list and an ordinary LP could create on PARE's infrastructure.

RECOMMENDATION

Do not let an arbitrary caller direct another account's accrued interest. Restrict `claimInterest` so it can only be triggered by the holder itself or an account it has explicitly authorized (for example require `msg.sender == user` or an approved operator), so a contract holder like the `PoolManager` cannot have its booked interest pushed to it and swept by a third party.

Alternatively, prevent YT from being pooled in holders that credit incoming balances to external callers (block YT in `PtPoolHook.beforeInitialize`), and document that YT must not be deposited into `PoolManager` -style singletons, v2 pairs, or Balancer-style vaults.

RESOLUTION

Pare

Fixed in <https://github.com/pare-audit/contracts/commit/c0caaa1>

Took the first recommendation: `claimInterest(user)` now requires `msg.sender == user` (reverts `NotHolder`). Interest booked to a contract holder such as the `PoolManager` can no longer be pushed into it and swept with `sync/settle`; it stays booked to the holder.

Tests added for a third-party claim reverting and for the exact M-02 path (YT held by a pool singleton, attacker calls `claimInterest(manager)`): it reverts, nothing reaches the manager, and the interest remains owed. Both fail on the previous code.

[M-03] The oracle caps one leg of a ratio and not the other, so a correlated stablecoin premium marks PT down and liquidates a healthy borrower

Medium

Fixed

SUMMARY

Morpho consumes a single number from this oracle: how much loan token one PT is worth, the ratio `underlying/loan` . The oracle never bounds that ratio directly. Instead it bounds an intermediate value: `peg()` caps `underlying/USD` at 1 (`:121`), and `price()` then divides by the raw, uncapped `loan/USD` answer (`:132`).

```
ratio = peg / loan    // peg caps underlying/USD at 1, loan is uncapped
```

Capping the numerator of a ratio while leaving the denominator free makes the error one-sided. When both stablecoins trade at the same premium, which is the normal correlated state for two USD stables in a risk-off move, the true ratio is still exactly 1 and the PT's value has not moved. But the cap clips the numerator while the denominator stays at, say, 1.003, producing a 29.9 bps mark-down on a position whose real value did not change.

When both legs trade at the same discount the cap does not bind and they cancel exactly (the PoC control asserts price == fair). So the error only ever points one way: against the borrower.

IMPACT

A borrower sitting 0.225% inside the LLTV limit, confirmed healthy with `liquidate` reverting `position is healthy`, becomes liquidatable one correlated +0.3% print later with nothing about the position changed.

In the PoC the liquidator seizes 10 PT truly worth 8.500000 USDC and repays only 8.258476, so 0.241524 USDC moves from the borrower to the liquidator, the liquidation incentive applied to a phantom deficit.

Across a market, every borrower within roughly 30 bps of the limit becomes liquidatable the moment both stables move together. The magnitude is bounded by the premium (measured -4/-9/-24/-49/-99 bps at 0.05-1% premiums, 0 on the discount control). The error is always downward, so it is lender-safe and cannot create bad debt. It is worth fixing because the liquidation is wrongful (the identical position reverted moments earlier) and the trigger is the most common stablecoin condition, not an exotic one.

RECOMMENDATION

Cap the ratio, not one leg:

```
ratio = min(peg, max(loan, 1)) / loan
```

The `max(loan, 1)` keeps a genuine loan-token discount credited, since a PT really is worth more loan tokens when the loan token is worth less. A naive `min(peg/loan, 1)` would merely move the one-sidedness rather than remove it, under-pricing by 13% in a USDC depeg.

All four cases are verified by the PoC: correlated premium prices at par, correlated discount prices at par, loan depeg credits the discount, and underlying depeg marks down.

Also refuse (or explicitly document) the half-configured `loanFeed` -without- `pegFeed` case, which hard-codes the underlying at \$1 while marking the loan token to market.

RESOLUTION

Pare

Fixed in <https://github.com/pare-audit/contracts/commit/33fe480>

`price()` now applies the cap to the ratio when a loan feed is set, as recommended:

```
ratio = min(underlying, max(loan, 1)) / loan
```

- Correlated premium: prices at par (previously marked down ~30 bps)
- Correlated discount: prices at par
- Loan-token depeg: discount still credited
- Underlying depeg: still marks down
- Underlying premium alone: still capped at par
- `loanFeed` without `pegFeed`: now refused in the constructor (`LoanFeedNeedsPegFeed`)
- No-feed path: unchanged

Tests added in `test/PtLinearOracle.t.sol` for all four PoC cases, the underlying-premium-alone case, the refused configuration, and a fuzz test that any common move of both feeds leaves the price unchanged. The premium, fuzz and refusal tests fail on the previous code.

[M-04] `skim` protects the backing share by address only, so a share reachable through a second address can be swept to the treasury by anyone

Medium

Fixed

SUMMARY

`skim` is permissionless and transfers the vault's entire balance of `token` to the treasury. The only protection for the shares that back every position is a single address comparison:

```
function skim(IERC20 token) external nonReentrant returns (uint256 amount) {
    if (address(token) == address(share)) revert CannotSkimShare();
    amount = token.balanceOf(address(this));
    if (amount == 0) return 0;
    token.safeTransfer(treasury, amount);
}
```

The property the function needs is "the share balance does not leave". What it actually checks is "the argument is not the share's address". These only agree for tokens with a single entry point.

A dual-entry token, where a second contract reads and moves the same ledger, passes the check under that second address: `balanceOf(vault)` reports the vault's share balance, and `transfer` moves it. This is not standard proxy behavior; the shared ledger has to be built in deliberately, as some tokens have done.

SCOPE.md states that any ERC-4626 share or any token with a rate feed may be listed through `YieldFactory` without further review, and flags "a share that reports a different address" as a `skim` concern. Neither `YieldFactory.createSeries`, the adapters, nor the vault constructor checks for a second entry point. sUSDx has no such alias, so the exposure is to later listings.

IMPACT

A single call moves the vault's entire share balance to the treasury. Every PT, YT, and booked interest claim is left unbacked, and `merge`, `redeemPT`, and `claimInterest` revert because the vault has no shares to pay out.

Recovery depends on the treasury returning the funds by hand; holders have no on-chain path to get them back.

In the PoC, Alice splits 100 shares, then an account with no position calls `skim` with the share's alias address:

	Before <code>skim(alias)</code>	After <code>skim(alias)</code>
Vault share balance	100	0
Treasury share balance	0	100
Alice <code>merge(100 PT + 100 YT)</code>	succeeds	reverts

EXPLOIT PATH

1. A share token with a second entry point (an alias contract over the same ledger) is listed through `YieldFactory`.
2. Users split shares into PT and YT as normal.
3. Any account, with no position of its own, calls `skim` passing the alias address, which passes the `address(token) == address(share)` check.
4. The vault's whole share balance is transferred to the treasury, leaving all positions unbacked and claim functions reverting.

RECOMMENDATION

Check the effect rather than the address. Record the share balance before the transfer and revert if it dropped:

```
function skim(IERC20 token) external nonReentrant returns (uint256 amount) {
    if (address(token) == address(share)) revert CannotSkimShare();
    +   uint256 shareBefore = share.balanceOf(address(this));
    amount = token.balanceOf(address(this));
    if (amount == 0) return 0;
    token.safeTransfer(treasury, amount);
    +   if (share.balanceOf(address(this)) < shareBefore) revert CannotSkimShare();
    emit Skimmed(address(token), amount);
}
```

Adding more addresses to the guard (PT, YT, a known proxy) does not close this, since any address check misses entry points it does not list, including ones a token adds later via upgrade. Comparing `token.balanceOf(vault)` with `share.balanceOf(vault)` also fails: an unrelated token can hold the same balance by coincidence, and an alias that reports balances differently would still pass. The before/after check reads the share's own `balanceOf`, so it holds for any second entry point as long as the share reports its own balances correctly.



RESOLUTION

Pare

Fixed in <https://github.com/pare-audit/contracts/commit/852036b>

Applied the recommendation as written: skim records `share.balanceOf(vault)` before the transfer and reverts `CannotSkimShare` if it dropped, so the guard checks the effect rather than the address.

Test added reproducing the PoC with an alias contract over the share's ledger: an account with no position calls `skim(alias)`, it reverts, the vault keeps all 100 shares, the treasury receives nothing, and Alice's merge still succeeds. The test fails on the previous code.

Low Severity

[L-01] Exact-output swaps can revert when the protocol fee is taken before input settlement

Low

Fixed

LOCATION

PtPoolHook:afterSwap:160-175

SUMMARY

The hook charges a protocol share on every swap: output-side for exact-in swaps and input-side for exact-output swaps.

For exact-output swaps, `afterSwap()` selects the input currency and immediately calls `poolManager.take`. That currency is not necessarily held by the PoolManager yet, since it is ordinarily transferred by the trader during the subsequent unlock settlement.

```
poolManager.take(inputCurrency, treasury, fee)
```

If the manager has no pre-existing input balance, `take` fails and the entire swap reverts. This makes exact-output trading depend on unrelated prior balances of the input token, so a newly bootstrapped or one-sided pool can support a valid exact-output swap economically yet reject it because the fee is collected at the wrong stage.

IMPACT

Exact-output swaps can be unavailable in pools that lack a pre-existing balance of the input token. This is a temporary but real availability failure for valid trades, particularly in newly initialized or one-sided pools.

EXPLOIT PATH

1. Initialize a dynamic-fee PT pool with liquidity only in the token that exact-output traders want to receive, and configure a nonzero protocol share.
2. A trader requests an exact-output swap and can provide the required input amount.



- Before the trader's settlement callback transfers the input, `afterSwap()` calls `poolManager.take(inputCurrency, treasury, fee)`.
- The manager lacks the input balance needed for `take`, so the hook reverts and the valid swap is blocked before settlement can fund it.

PREREQUISITES

The pool has `protocolPips > 0`, the trade is exact-output, the PoolManager's current balance of the input currency is less than the calculated protocol fee, and the pool otherwise has enough output liquidity for the requested swap.

RECOMMENDATION

Charge the fee through the hook delta rather than taking the not-yet-settled input immediately, allowing the normal unlock settlement to fund it:

```
if (isExactOutput) {  
    return (IHooks.afterSwap.selector, int128(uint128(fee)));  
}
```

Alternatively, defer `poolManager.take` until after the input currency has been settled, and verify that the deferred path cannot be bypassed or double-charge the trader.

RESOLUTION

Pare

Fixed in <https://github.com/pare-audit/contracts/commit/f4f08af>

On exact-output swaps the hook now books the protocol fee as a PoolManager ERC-6909 claim (`poolManager.mint` to the hook) instead of `take()`, so no input balance is needed when `afterSwap` runs. A permissionless `collect(currency)` later burns the claim and takes the tokens to the treasury. Exact-input swaps keep `take()`, since the output is already in the manager.

We used the deferred route rather than returning the delta alone: without a `take` or `mint`, the hook's own `+fee` delta stays unsettled and the unlock reverts.

Test added for exact-out in a fresh pool whose manager holds none of the input token (fails on the previous code); the existing exact-out test now checks the claim and the `collect()` payout.

[L-02] Intermediate rounding permanently discards accrued YT interest

Low

Fixed

LOCATION

YieldVault::_accrue:169-183

SUMMARY

Each YT holder should receive the share-denominated yield accrued between their last checkpoint and the current vault index. The interest math truncates in an intermediate step before the final conversion to

shares:

```
uint256 interest = Math.mulDiv(
    Math.mulDiv(bal, idx - prev, prev),
    ONE,
    idx
);
userIndex[user] = idx;
```

When the inner `mulDiv(bal, idx - prev, prev)` rounds down to zero, the function books no interest but still advances `userIndex[user]` to `idx`. A later call then starts from the newer index, so the skipped yield can never be recovered.

For example, a victim holding 99 YT with `userIndex = 0.01e18` and a rate rising to `0.0101e18` yields `floor(99 * 0.0001e18 / 0.01e18) = 0`, even though the fully scaled rational expression would credit roughly 98 share units.

This is distinct from the final, unavoidable raw-token rounding: it is the avoidable intermediate floor that discards value. Two holders with identical balances and total index movement can end up with different interest purely because their checkpoint schedules differ, and the discarded interest becomes unallocated protocol surplus rather than being credited to the holder.

IMPACT

YT holders can permanently lose accrued share-denominated interest. The loss repeats across many small index increases and is most pronounced for low-decimal tokens or sub-WAD rate changes.

Because `claimInterest` accepts any user address and is permissionless, any account can trigger the zero-rounded checkpoint against a victim, repeatedly stripping small share claims.

EXPLOIT PATH

1. A victim holds 99 YT with `userIndex[victim] = 0.01e18`; the share rate increases to `0.0101e18`.
2. Any caller invokes `claimInterest(victim)`.
3. `_accrue` computes `floor(99 * 0.0001e18 / 0.01e18) = 0`, records no interest, and sets `userIndex[victim] = 0.0101e18`.
4. The victim cannot claim the roughly 98 share units for that interval, and repeated small increases cause the same loss again.

PREREQUISITES

A supported share token and rate configuration that produces sub-unit intermediate accruals while the fully scaled interest is nonzero, and a victim holding YT with a prior `userIndex` checkpoint.

RECOMMENDATION

Compute the complete expression before dividing, avoiding the raw-unit truncation in the first `mulDiv`. One approach multiplies the YT balance and WAD scale first and uses a full-precision denominator:

```
uint256 interest = Math.mulDiv(
    bal * ONE,
    idx - prev,
    prev * idx
);
```

If overflow bounds make this direct form unsuitable, use OpenZeppelin's full-precision `mulDiv` in two carefully ordered stages, or an equivalent 512-bit arithmetic implementation that preserves the complete numerator while avoiding overflow.

RESOLUTION

Pare

Fixed in <https://github.com/pare-audit/contracts/commit/c71e5ae>

Both `_accrue` and the `pendingInterest` view now compute the interest in one full-precision step: `mulDiv(bal, (idx - prev) * ONE, prev * idx)`, so only the final result rounds down.

Test added with the report's example: 99 YT with the index moving 0.01 -> 0.0101 now books 98 shares of interest (96 after the 3% fee) instead of 0, and `pendingInterest` matches the claim. The test fails on the previous code.

The third-party trigger described (`claimInterest` on a victim's behalf) is also closed by the M-02 fix, which restricts `claimInterest` to the holder.

[L-03] Legacy Chainlink lagging rounds are accepted without `answeredInRound` validation

Low

Fixed

LOCATION

`ChainlinkRateAdapter:rate:48-56`, `PtLinearOracle:_read:148-153`

SUMMARY

Both `ChainlinkRateAdapter.rate()` and `PtLinearOracle._read()` read a Chainlink feed but validate only the answer value, timestamp, and adapter bounds. They ignore the round identifiers, so a legacy FluxAggregator-style feed that returns an `answeredInRound` lower than the current `roundId` is accepted as long as its timestamp falls within the freshness window.

The `answeredInRound < roundId` condition means the current round has not actually produced a fresh answer and the returned value comes from an earlier round. Modern OCR feeds report equal round identifiers and are unaffected, but a legacy feed can still be configured, leaving the integration unsafe.

For example, a feed returning `roundId = 5`, `answeredInRound = 3`, `answer = 1.00` with a recent `updatedAt` passes the positivity and staleness checks even though the answer belongs to round 3.

IMPACT

A legacy feed can supply a price from an earlier answered round that still looks recent under the timestamp check. That stale intermediate value then flows into the vault index update or the PT oracle backing and price calculation, so consumers act on a non-current price instead of rejecting the incomplete current round.

EXPLOIT PATH

1. A legacy-style feed returns `roundId = 5`, `answeredInRound = 3`, `answer = 1.00`, and a recent `updatedAt`.



2. `ChainlinkRateAdapter.rate()` or `PtLinearOracle._read()` checks only positivity and staleness, so the response is accepted.
3. The accepted answer updates the vault index or feeds the backing and PT price calculation.
4. Consumers act on a value from an earlier answered round.

PREREQUISITES

The configured feed must expose legacy FluxAggregator-style round semantics and return `answeredInRound < roundId` while its timestamp still passes the configured freshness check. Modern OCR proxy feeds are not affected.

RECOMMENDATION

Validate the round relationship in both feed-reading paths when legacy feeds are permitted:

```
(uint80 roundId, int256 answer,, uint256 updatedAt, uint80 answeredInRound) =  
f.latestRoundData();  
if (answeredInRound < roundId) revert IncompleteRound(roundId, answeredInRound);
```

Preferably, additionally restrict deployment to approved modern Chainlink proxy feeds and make that policy explicit in the factory configuration.

RESOLUTION

Pare

Fixed in <https://github.com/pare-audit/contracts/commit/89ef198>

Both `ChainlinkRateAdapter.rate()` and `PtLinearOracle._read()` now revert `IncompleteRound(roundId, answeredInRound)` when `answeredInRound < roundId`, as recommended. A refused rate in the adapter freezes the vault index at its last value and leaves every exit open.

Tests added for both readers: round 5 answered in round 3 is refused, and the same feed is accepted once answered in the current round. Both tests fail on the previous code. On the feed policy: our deployments use modern Chainlink proxy feeds only.

[L-04] previewSplit does not reflect a supply-cap-closed vault

Low

Fixed

LOCATION

YieldVault:previewSplit:312-320

SUMMARY

`previewSplit()` is meant to return zero whenever a split cannot actually be performed. The documentation treats a supply cap set below the current PT supply as equivalent to pausing new splits.

The implementation never inspects `supplyCap`, so it can report a positive mint estimate even though

`split()` will always revert with `CapExceeded` :

```
if (isMatured() || splitsPaused) return 0;
(bool ok, uint256 r) = _tryRate();
if (!ok || r < pyIndexStored) return 0;
return Math.mulDiv(shares, r, ONE);
```

The same mismatch also occurs when the cap is full or nearly full and the requested split would exceed only the remaining capacity. A correct check must account for both the current supply and the requested minted amount, not merely whether the cap is nonzero. This returns an incorrect preview but does not transfer funds or bypass the cap.

IMPACT

Users and integrators receive a positive split preview while the corresponding `split()` transaction necessarily reverts.

This drives incorrect UI, routing, and automation decisions, and causes avoidable integration errors and wasted gas when a positive preview is acted on before submitting the split.

EXPLOIT PATH

1. The vault has 1,000 PT outstanding and the owner calls `setSupplyCap(900)` .
2. The vault is still before maturity, not manually paused, and its adapter reports a valid rate.
3. A caller invokes `previewSplit(100)` , which returns a positive mint estimate because it ignores `supplyCap` .
4. The caller invokes `split(100, receiver)` , which reverts because `pt.totalSupply() + minted` exceeds `supplyCap` .

PREREQUISITES

The vault owner has set a nonzero `supplyCap` at or below the current PT total supply (or close enough that the requested split exceeds remaining capacity), while the vault is unmatured, not paused, and the adapter reports a valid rate.

RECOMMENDATION

Return zero when the cap leaves no capacity for the requested split. Since `previewSplit` accepts an amount rather than returning a maximum, the simplest compatible fix is:

```
if (supplyCap != 0 && pt.totalSupply() >= supplyCap) return 0;
```

For a more precise preview, cap the returned value to the remaining PT capacity after converting it to the corresponding share amount, or provide a separate `maxSplittable` view.



RESOLUTION

Pare

Fixed in <https://github.com/pare-audit/contracts/commit/ca0f70e>

previewSplit now applies the same cap check as split(): with a nonzero cap, it returns 0 if pt.totalSupply() + minted would exceed supplyCap. This covers both the closed case (cap at or below the current supply) and a request larger than the remaining capacity, so the cap can never make the preview and the split disagree.

Test added: with a 100 cap, 100 previews in full and 101 previews 0; after 60 is split, 40 previews in full and 41 previews 0; a cap moved below the supply previews 0 and split reverts. The test fails on the previous code.

[L-05] Protocol fee share is lost when total fee rounds to one pip

Low

Fixed

LOCATION

PtPoolHook:feeSplitPips:128-133

SUMMARY

The hook splits the configured total swap fee between LPs and the treasury, but `feeSplitPips()` floors the protocol portion only after `totalFeePips()` has already been reduced to integer pips.

When the total fee clamps to one pip, `protocolPips` floors to zero for every protocol share below 100% (including the maximum permitted 50%), while `lpPips` stays at one.

As a result, `beforeSwap` charges the one-pip fee entirely to LPs, and `afterSwap()` sees `protocolPips = 0` and skips `poolManager.take`, so the treasury's configured share is dropped for every swap in this fee regime.

IMPACT

The treasury receives no protocol fee while the total fee sits at one pip, despite a nonzero configured protocol share.

The missing amount is bounded per swap but accumulates over large trading volume. Because the boundary only occurs during portions of a time-dependent fee schedule, treasury collection becomes discontinuous as the scheduled total fee moves across the one-pip level.

EXPLOIT PATH

1. Deploy the hook with `protocolShareBps = 5,000`, `minFeePips = 1`, and a valid `maxFeePips`.
2. While the time-based fee is below one pip, `feeSplitPips()` returns `lpPips = 1` and `protocolPips = 0`.
3. A swap applies the one-pip LP fee, and `afterSwap` recomputes the zero protocol allocation.
4. `afterSwap` returns without `poolManager.take`, so the treasury earns nothing from that swap; repeated swaps compound the omitted revenue.



PREREQUISITES

The hook must be deployed with `minFeePips = 1` and `protocolShareBps` greater than 0 but less than 10,000, and swaps must occur while the clamped total fee equals one pip. Validating fee bounds at configuration time also prevents the issue.

RECOMMENDATION

Carry the unrounded protocol numerator across swaps using a fractional accumulator scoped per pool/currency, so the dropped fraction is retained until it reaches one full pip.

```
uint256 numerator = abs * protocolPips + feeRemainder[poolId][feeCurrency];
uint256 fee = numerator / PIPS;
feeRemainder[poolId][feeCurrency] = numerator % PIPS;
if (fee != 0) poolManager.take(feeCurrency, treasury, fee);
```

RESOLUTION

Pare

Fixed in <https://github.com/pare-audit/contracts/commit/dcaa2b0>

Took the configuration-time route the finding mentions: the constructor now reverts `MinFeeTooSmallToSplit` when `protocolShareBps != 0` and `minFeePips * protocolShareBps < BPS`, so the floored protocol share can never be zero while LPs still take a pip. Since the fee never goes below the floor, the treasury's share is at least one pip on every swap. We did not add the per-pool remainder accumulator: it would add a storage write to every swap, and `afterSwap`'s early return on `protocolPips == 0` would bypass it at the floor anyway. Our deployment (floor 100 pips, 50% share) was not in the affected regime.

Test added: floor 1 with a 50% share is refused, floor 2 is accepted and splits 1/1 at the floor, floor 1 with no treasury share is accepted. The refusal fails on the previous code.

[L-06] Rate outage causes matured PT redemptions to transfer unallocated post-maturity yield

Low

Acknowledged

LOCATION

YieldVault:_payMaturedPT:242-251

SUMMARY

Each matured PT is meant to receive the share amount for one unit of underlying at the current nondecreasing rate, with any appreciation after maturity on unredeemed PT credited to the treasury. When the adapter rate read fails, the code falls back to the stored `expiryIndex` instead of preserving this

distinction:

```
uint256 reserved = Math.mulDiv(amount, ONE, expiryIndex);
(bool ok, uint256 r) = _tryRate();
shares = Math.mulDiv(amount, ONE, ok ? Math.max(r, expiryIndex) : expiryIndex);
uint256 swept = reserved - shares;
if (swept != 0) feesOwed += swept;
```

When `ok` is false, `shares` equals `reserved` and `swept` is zero. If the real rate is actually higher, the redeemed PT receives shares worth more than the intended principal, and the excess post-maturity yield is swept to nobody.

For example, with `expiryIndex = 1` and a true current rate of 2, redeeming 100 PT pays 100 shares instead of the intended 50, with no credit to `feesOwed`.

IMPACT

During a rate outage after maturity, PT holders can redeem at the expiry-rate share amount even though the underlying share has appreciated. This transfers post-maturity yield that should go to the treasury to redeemers, with losses proportional to the redeemed PT.

Because the PT is burned during the outage, a later successful rate read cannot recover the missed sweep. The loss is permanent once the redemption completes.

Treasury fee accounting therefore depends on whether the feed happens to be live at the exact moment of each individual redemption. A redeemer can intentionally wait for an outage window to capture treasury-designated value.

EXPLOIT PATH

1. The vault settles at `expiryIndex = 1` while a user holds 100 PT.
2. After maturity the underlying share rate rises to 2, but the adapter begins reverting (for example, a stale or ceiling-rejected feed answer).
3. The user calls `redeemPT(100, receiver)`; `_tryRate()` fails and `_payMaturedPT()` computes 100 shares using `expiryIndex` instead of 50 at the current rate.
4. The PT is burned, `feesOwed` receives no post-maturity yield, and the user keeps roughly 50 excess shares of value once the feed recovers.

PREREQUISITES

The series must have a nonzero `expiryIndex` and the underlying share must have appreciated above that index after maturity. The adapter rate read must revert during `redeemPT()` or `merge()`, such as from a stale or invalid feed. This is an external-condition accounting issue, not attacker-controlled oracle manipulation.

RECOMMENDATION

The safest minimal fix is to require a valid post-maturity rate for PT payouts rather than treating an



unavailable rate as `expiryIndex` :

```
(bool ok, uint256 r) = _tryRate();
if (!ok) revert RateUnavailable();
uint256 effectiveRate = Math.max(r, expiryIndex);
shares = Math.mulDiv(amount, ONE, effectiveRate);
feesOwed += reserved - shares;
```

If exits must remain open during outages, add explicit per-PT post-maturity reserve accounting and reconcile the reserve once a valid rate is available; do not transfer the full expiry-rate reserve to the redeemer without recording the unresolved difference.

RESOLUTION

Pare

Thanks. we'll take option 2 and keep the code as it is. Our keeper will call `updateIndex()` daily on every Chapter 2 vault after maturity, alongside the `settle()` calls it already makes at maturity and when the settling window closes, so the outage fallback is never more than about a day old. `updateIndex()` is permissionless, so anyone can refresh it as well. Happy for this to be marked acknowledged.

[L-07] Zero-value YT transfers can permanently discard truncated interest

Low

Fixed

LOCATION

YieldToken:_update:45-48

SUMMARY

YT transfers run a hook that checkpoints accrued interest before balances change, so yield earned by the old holder does not travel with the token. The hook advances `userIndex` and does floor-rounded interest accounting.

The problem is that this hook runs even when the ERC-20 transfer amount is zero. On a zero-value transfer the index is still advanced, interest is computed with floor rounding, and when that result is zero no remainder is retained.

Because a zero-value `transferFrom` needs no allowance, anyone can repeatedly force a victim's checkpoint at unfavorable intermediate indices. Each forced checkpoint moves `userIndex` forward while contributing zero interest, so the dust that would have accumulated over a larger interval is thrown away.

IMPACT

A permissionless caller can repeatedly checkpoint any YT holder across small index movements, discarding rounding dust that would otherwise accrue over a larger interval. This is a real accounting loss of interest for affected holders, bounded by integer precision.

EXPLOIT PATH

1. A victim holds YT with `userIndex[victim] = i0` , and the adapter later reports `i1 > i0` with a

difference small enough that `_accrue` floors interest to zero.

2. The attacker calls `yt.transferFrom(victim, victim, 0)` with no approval.
3. The hook runs `_accrue(victim, i1)`, writes `userIndex[victim] = i1`, and computes `interest = 0` after flooring.
4. The victim's later claims omit the yield between `i0` and `i1`, even though their YT balance never changed.

PREREQUISITES

The victim must hold YT with a prior checkpoint, and the adapter index must increase by an amount small enough to floor to zero interest. The attacker needs only the public YT token; no allowance is required for a zero-value transfer.

RECOMMENDATION

Skip the bookkeeping hook for zero-value updates:

```
function _update(address from, address to, uint256 value) internal override {
    if (isYield && value > 0) {
        IYieldVaultHook(vault).beforeYieldTransfer(from, to);
    }
    super._update(from, to, value);
}
```

Alternatively, preserve fractional accrual in a per-user remainder and include it in the next calculation:

```
interestOwed[user] += interest;
interestRemainder[user] = rawInterest % SCALE;
```

RESOLUTION

Pare

Fixed in <https://github.com/pare-audit/contracts/commit/cdd6258>

Applied the first recommendation as written: `YieldToken._update` calls `beforeYieldTransfer` only when `value != 0`, so a zero-value transfer moves nothing and books nothing, and a stranger's `transferFrom(holder, holder, 0)` no longer advances the holder's checkpoint.

Test added: a holder of 1 gwei YT, a 1-wei index step, then a zero `transferFrom` by an unrelated account; `userIndex` stays at 1.0, the balance is unchanged, and the later claim pays the full rise from 1.0. The test fails on the previous code (the checkpoint moved).



Info Severity

[I-01] Contracts compiled for Cancun cannot be deployed on chains without its opcodes

Info

Fixed

SUMMARY

The build targets the Cancun EVM version via `evm_version = "cancun"` in `foundry.toml`. This causes the compiler to emit Cancun's `MCOPY` opcode in `YieldToken` (in its constructor and in the string encoding behind `name()` and `symbol()`) as well as in `YieldVault`, `YieldFactory`, and `YieldPeriphery`. It also emits Shanghai's `PUSH0` opcode in every in-scope contract.

On a chain that does not support these opcodes, the first `MCOPY` or `PUSH0` to execute fails as an invalid instruction.

IMPACT

The team plans to deploy the same contracts on other EVM chains, including L2s. On a chain lacking these opcodes, `YieldFactory.createSeries()` reverts, because the vault and its PT and YT are created with code that uses `MCOPY`. Contracts compiled with `PUSH0` fail at deployment or on their first call.

No funds are at risk, since the failure surfaces before any deposit, but listing on such a chain requires a different build.

RECOMMENDATION

Build for each target chain at the newest EVM version that chain supports, using a dedicated profile in `foundry.toml` for chains that lack Shanghai or Cancun:

```
[profile.default]
evm_version = "cancun"

[profile.paris]
evm_version = "paris"
```

Build and deploy those chains with `FOUNDRY_PROFILE=paris`.

RESOLUTION

Pare

Fixed in 285518c: the default profile now sets `evm_version = "cancun"` explicitly, so the build matches the reviewed bytecode under any Foundry version. The paris profile is unchanged.



[I-02] The hook's mid-swap `take()` makes a pay-first router charge the user the protocol fee twice on exact-out swaps

Info

Fixed

SUMMARY

On exact-out swaps the fee currency is the input currency, and `afterSwap` collects the protocol fee by moving tokens out of the PoolManager mid-swap:

```
poolManager.take(feeCurrency, treasury, fee);
```

The hook's own accounting is correct here: `take` debits the hook by `fee` and the returned delta credits it back by `fee`.

The problem only surfaces for a router that pays before swapping (`sync(input)` then transfer `maxIn`, then `swap`, then `settle()`, then refund the excess). Its `settle()` is measured against a PoolManager balance that the hook's `take` has just reduced, so it is credited `maxIn - fee` instead of `maxIn` while the swap still charges `input + fee`. The refund comes out `fee` short.

The user ends up paying the fee once to the treasury and a second time into the PoolManager, where no delta records it and nobody can recover it.

IMPACT

Users swapping exact-out through a pay-first router lose one extra protocol fee per swap (0.1% of input in the PoC), and that amount is permanently stranded in the PoolManager.

This is a compatibility hazard rather than a flaw in the protocol's own logic. Mainstream routers that `sync`, transfer, and settle in one step (before or after the swap) are unaffected, and native-currency inputs are unaffected because `settle` uses `msg.value`.

PREREQUISITES

Requires a non-standard router that keeps a `sync` / `settle` window open across the swap, combined with an exact-out swap paid in a non-native input currency.

RECOMMENDATION

Collect the fee as ERC-6909 claims instead of `take`, so no tokens move during the swap and router ordering no longer matters:

```
poolManager.mint(treasury, feeCurrency.toId(), fee);
```

RESOLUTION

Pare

Addressed by the L-01 fix, <https://github.com/pare-audit/contracts/commit/f4f08af>, with a test added in <https://github.com/pare-audit/contracts/commit/ab322e7>.

On exact-out swaps afterSwap no longer calls take(): it mints the protocol fee as an ERC-6909 claim and a permissionless collect(currency) later burns the claim and takes the tokens to the treasury in its own unlock. No tokens leave the PoolManager during the swap, so a pay-first router's settle() is measured against an untouched balance. The claim is minted to the hook rather than straight to the treasury only so the treasury keeps receiving plain tokens; the mid-swap behaviour is as recommended.

The test builds the router from the finding (sync, transfer maxIn, swap, settle, refund) and checks the user is charged exactly the swap delta's cost, identical to the settle-after router from the same state. On the previous take() path the user paid one extra fee (0.1% of input).

[I-03] Gas burning rate adapter failure freezes pre maturity merge for every holder

Info

Fixed

SUMMARY

`YieldVault._tryRate()` wraps `adapter.rate()` in a `try / catch` so a failing rate source falls back to the stored index instead of blocking exits. The call forwards all available gas, and under the EVM's 63/64 rule a rate source that fails by consuming its gas (an `INVALID` opcode, an out-of-gas loop, or a very large revert payload) leaves the caller with only 1/64 of its gas.

`YieldVault.merge()` reads the rate twice. The first read happens through `_updateIndex()`. The second happens when `yt.vaultBurn()` runs the YT transfer hook, which calls `YieldVault.beforeYieldTransfer()` and `_updateIndex()` again.

```
// YieldVault._tryRate()
try adapter.rate() returns (uint256 v) {
  if (v != 0) return (true, v);
} catch {}
return (false, 0);

// YieldVault.merge()
uint256 idx = _updateIndex();
yt.vaultBurn(msg.sender, amount);
```

After the first failed read the merge keeps about 1/64 of its gas, and after the nested second read about 1/4096, far below the roughly 50,000 gas its remaining work needs. Merge therefore reverts at every available gas limit on mainnet, including the 16,777,216 gas per-transaction cap and a 60,000,000 gas block. The same rate source failing with an ordinary revert leaves merge working.

IMPACT

While the rate source keeps failing this way, no holder can merge PT and YT back into shares before maturity. Merging is the only pre-maturity exit, so this breaks the documented promise that exits never

depend on a live rate.

No funds are lost. PT and YT remain transferable and `YieldVault.claimInterest()` keeps paying. After maturity, `YieldVault.settle()` followed by `YieldVault.redeemPT()` still works because each performs a single rate read.

PREREQUISITES

Requires a listed rate source that fails by burning its gas rather than by reverting. Standard rate sources do not behave this way.

RECOMMENDATION

Cap the gas forwarded to the adapter, and revert instead of falling back when the caller left too little gas for the read:

```
uint256 internal constant RATE_GAS = 1_000_000;

function _tryRate() internal view returns (bool ok, uint256 r) {
    try adapter.rate{gas: RATE_GAS}() returns (uint256 v) {
        if (v != 0) return (true, v);
    } catch {
        if (gasleft() < RATE_GAS / 63) revert RateUnavailable();
    }
    return (false, 0);
}
```

RESOLUTION

Pare

Fixed in <https://github.com/pare-audit/contracts/commit/87e2740>

Applied the recommendation as written: `_tryRate` calls `adapter.rate{gas: RATE_GAS}()` with `RATE_GAS = 1,000,000`, and the catch reverts `RateUnavailable` when `gasleft() < RATE_GAS / 63`, so a caller that left too little gas for the capped read gets a clean failure to retry rather than a silent fallback.

Test added with an adapter that fails by executing `INVALID`: after it starts burning, merge still finishes at the stored index through both reads, `previewSplit` returns 0, `split` reverts `RateUnavailable`, and a merge sent with 1.2M gas reverts `RateUnavailable` with data. On the previous code the same merge consumed over 1e9 gas and reverted empty.

[I-04] `split()` makes two independent `adapter.rate()` calls

Info

Fixed

SUMMARY

`split()` reads the adapter rate twice through two separate external calls. First `_updateIndex()` reads the rate internally (line 201), then `_tryRate()` is called again at line 202.



```
_updateIndex();  
_tryRate();
```

Within a single transaction both reads return the same value, since storage cannot change mid-call, so there is no exploitable divergence today. The issue is architectural: the `RateBelowIndex` guard and the mint math reference two independent rate reads. Any future adapter whose rate is stateful or non-deterministic would silently split the invariant across those two reads.

IMPACT

No impact under current EVM atomicity. This is a code-quality risk for future adapter integrations, where the module's core design goal is to support any adapter.

RECOMMENDATION

Read the rate once and thread it through. Have `_updateIndex` return `(idx, r)` and use both values consistently instead of making a second `_tryRate()` call.

RESOLUTION

Pare

Fixed in <https://github.com/pare-audit/contracts/commit/10bcc7a>

As recommended: `_updateIndex` now returns the index together with the one rate read it made (`_updateIndexAndRate`), and `split` uses that single reading for both the `RateBelowIndex` guard and the mint. `previewSplit` takes its index and rate from one `_nextIndex()` read the same way, and the settlement branch no longer re-reads the rate to note the post-maturity rate.

Test added: `split` performs exactly two adapter calls, its own and the one the YT mint hook makes for its index update, where the previous code performed three. The hook's read is a separate nested update rather than a second input to the same decision.

[I-05] ERC-4626 Empty-State Rate Reset Can Permanently Block New Splits

Info

Fixed

SUMMARY

YieldVault stores `pyIndexStored` as a non-decreasing high-water mark and rejects any new split when the current rate falls below it.

Many real ERC-4626 implementations (for example Convex `cvxFXB` and Gearbox `PoolV3`) fall back to a 1:1 conversion rate when their share supply drops to zero. If the underlying vault is fully emptied after PARE has already recorded an index above 1, the reported rate resets to 1.

Because `pyIndexStored` is never reset, every later `split()` call reverts once `r < pyIndexStored`.

IMPACT

The YieldVault can no longer accept new split deposits for the remainder of the series lifetime, a denial-of-service for new users.

Existing user funds are generally not directly affected, since the reset condition only arises after all PARE



positions have exited.

RECOMMENDATION

Permit an index reset only when the vault has no remaining obligations: zero PT supply, zero YT supply, zero underlying share balance, and no unclaimed interest or fees.

Alternatively, explicitly prevent reopening an emptied series and require deployment of a new series instead.

RESOLUTION

Pare

Fixed in <https://github.com/pare-audit/contracts/commit/1f5601f>

Took the first option: split (and previewSplit, so the two agree) reset `pyIndexStored` to today's rate when the rate is below it and both `pt.totalSupply()` and `yt.totalSupply()` are zero; with any PT or YT outstanding the `RateBelowIndex` refusal stands. We used those two conditions rather than the full list because nothing index-denominated can be owed once no PT or YT exists, while `interestOwed` and `feesOwed` are share amounts that a reset does not touch, and a zero share balance would rarely hold (merge rounding leaves dust).

Test added: a split at 1.0, the rate credited to 1.1, everyone merges out, the share rate resets to 1.0; `previewSplit` quotes and `split` mints at 1.0 and the index becomes 1.0. With a holder still in, a lower rate is refused as before. The test fails on the previous code (`preview 0`, `split` refused).

[I-06] Chainlink adapter accepts future-dated observations

Info

Fixed

LOCATION

`ChainlinkRateAdapter:rate:30-32`

SUMMARY

The Chainlink rate adapter is meant to reject stale observations, but it only enforces the lower freshness bound. A configured nonzero `maxAge` should impose a two-sided time check: the observation must be neither too old nor dated in the future.

The implementation checks only staleness, so a future timestamp makes the condition false and the answer is accepted:

```
if (answer <= 0) revert BadAnswer(answer);
if (updatedAt + maxAge < block.timestamp) revert Stale(updatedAt);
r = Math.mulDiv(uint256(answer), 1e18, unit);
```

With no future-timestamp validation, an invalid future-dated observation flows into `YieldVault` and `PtLinearOracle`. This is an external-condition failure (a feed returning a future timestamp) rather than an attacker-controlled permissionless mutation, but the adapter has no compensating validation.



IMPACT

An invalid future-dated observation bypasses the configured freshness bound and is consumed as if valid.

If the answer is materially wrong, it can distort new splits, interest accounting, and oracle valuation. A sufficiently high answer permanently advances `pyIndexStored` to a new high.

Because the index is monotonic, an overstated future-dated rate can prevent later, lower valid rates from restoring it, leaving the index overstated and potentially disabling future splits. The inflated index can also drive `_accrue()` and `claimInterest()` to transfer shares based on unearned value, inflating PT/YT issuance and creating undercollateralized claims.

`PtLinearOracle._read()` relies on the same timestamp freshness and can likewise consume the invalid observation on direct oracle reads.

EXPLOIT PATH

1. The configured rate feed returns a positive answer with `updatedAt` set to a future timestamp.
2. A user calls `split()` while the series is still immature.
3. `ChainlinkRateAdapter.rate()` sees that `updatedAt + maxAge` is not less than `block.timestamp` and returns the answer.
4. If the answer exceeds `pyIndexStored`, `YieldVault` records it as the new high index, distorting subsequent splits and interest accounting.

PREREQUISITES

The configured feed must return a positive answer with an `updatedAt` later than the current block timestamp. `maxAge` is nonzero but by itself does not prevent future timestamps.

RECOMMENDATION

Add an explicit future-timestamp check and use a subtraction-based age comparison:

```
if (updatedAt > block.timestamp) revert FutureAnswer(updatedAt);
if (block.timestamp - updatedAt > maxAge) revert Stale(updatedAt);
```

Apply the same validation to every feed-reading component that relies on timestamp freshness, including `PtLinearOracle._read()`.

RESOLUTION

Pare

Fixed in <https://github.com/pare-audit/contracts/commit/d687a51>

Applied the recommendation as written to both feed-reading paths: `ChainlinkRateAdapter.rate()` and `PtLinearOracle._read()` revert `FutureAnswer(updatedAt)` when `updatedAt > block.timestamp`, and the staleness check is now the subtraction form `block.timestamp - updatedAt > maxAge`.

Tests added for both readers: an observation dated one second in the future is refused, and the same feed is accepted once dated at the current block. Both tests fail on the previous code.



[I-07] claimFees does not settle the vault on the first post-maturity interaction

Info

Fixed

LOCATION

YieldVault:claimFees:266-270

SUMMARY

The vault lifecycle expects the first interaction at or after maturity to fix `expiryIndex`, after which YT interest stops accruing against a live post-maturity rate.

`claimFees()` is a permissionless vault interaction, but it only consumes `feesOwed` and transfers shares. It never calls `_updateIndex()`, so maturity can pass while `expiryIndex` stays `0`.

If the adapter rate rises before a later YT checkpoint, that later checkpoint fixes `expiryIndex` at the newer rate and books the intervening growth as YT interest.

The same gap applies to `skim`, which can move unrelated reward or airdrop tokens while leaving the maturity transition incomplete.

IMPACT

YT holders can receive extra interest for rate growth that happens after maturity but before any action settles the vault, because the maturity index stays unfixed.

This affects all YT holders of a series whenever `claimFees` (or `skim`) is the first post-maturity interaction.

EXPLOIT PATH

1. The series reaches maturity with `expiryIndex = 0` and `feesOwed > 0`.
2. Any caller invokes `claimFees()`; fees transfer to the treasury but `expiryIndex` remains zero.
3. The adapter rate increases while no settling action occurs.
4. `claimInterest(holder)` calls `_updateIndex()` at the later rate and books the post-maturity increase as the holder's YT interest.

PREREQUISITES

The series must be matured with `expiryIndex` still zero, `feesOwed` nonzero, no prior post-maturity call to `_updateIndex`, and the adapter rate must increase between `claimFees` and the next YT checkpoint.

RECOMMENDATION

Call `_updateIndex()` before reading or transferring fees so `claimFees` participates in the maturity transition. Apply the same fix to `skim`.

```
function claimFees() external nonReentrant returns (uint256 shares) {
    _updateIndex();
    shares = feesOwed;
    if (shares == 0) return 0;
    feesOwed = 0;
    share.safeTransfer(treasury, shares);
}
```



RESOLUTION

Pare

Fixed in <https://github.com/pare-audit/contracts/commit/0f963ac>

Applied the recommendation as written: `claimFees()` and `skim()` both call `_updateIndex()` before anything else, so whichever interaction comes first at or after maturity fixes `expiryIndex`.

Tests added for both: a rate rise shortly before maturity that the index has not yet credited, then `claimFees` (or `skim`) as the first call at maturity; `expiryIndex` is fixed at 1.0 and a later `claimInterest` books nothing for that rise. Both tests fail on the previous code (`expiryIndex` stayed 0).

[I-08] Downward rounding can permanently strand coarse-decimal deposits

Info

Fixed

LOCATION

YieldVault:split:205-214; YieldVault:merge:218-227

SUMMARY

A split mints PT/YT by rounding the issuance conversion down and only checks that at least one PT is minted. It never verifies that the inverse conversion (PT back to shares) yields a nonzero amount.

With a zero-decimal share worth 1.5 underlying units (`idx = 1.5e18`), one deposited share mints `floor(1 * 1.5) = 1` PT, but one PT converts back to `floor(1e18 / 1.5e18) = 0` shares because the payout uses floor division.

The zero-payout can happen on a merge even when both PT and YT inputs are positive, so a nonzero-input check alone is insufficient.

IMPACT

A user can lose an entire deposited share when the token decimals and exchange rate cause issuance to round up to a positive PT amount while redemption rounds down to zero shares.

Both PT and YT are burned while the vault keeps the deposited share, and the share token cannot be skimmed, so there is no recovery path. The loss can be economically significant for high-value, low-decimal share tokens. This affects both the merge path and the PT redemption path.

EXPLOIT PATH

1. A user calls `split(1, user)` for a zero-decimal share token with `idx = 1.5e18`.
2. The vault receives one share and mints `floor(1 * 1.5) = 1` PT and 1 YT.
3. The user calls `merge(1, user)`, burning both tokens and computing `floor(1e18 / 1.5e18) = 0` shares.
4. The vault transfers zero shares, keeping the deposited share while the user's position is destroyed.

PREREQUISITES

A configured supported share token must have coarse enough decimals and a rate such that `floor(received * idx / 1e18) > 0` while the inverse payout rounds to zero. A zero-decimal token with a

rate greater than `1e18` is sufficient.

RECOMMENDATION

Reject a split whenever the resulting PT amount would not be redeemable for at least one share unit:

```
minted = Math.mulDiv(received, idx, ONE);
if (minted == 0 || Math.mulDiv(minted, ONE, idx) == 0) {
    revert ZeroAmount();
}
```

A stronger design is to add explicit minimum-output protections to merge and redemption and ensure issuance cannot create a position below that minimum. Apply this consistently to both pre- and post-maturity payout paths.

RESOLUTION

Pare

Fixed in <https://github.com/pare-audit/contracts/commit/4469a70>

Applied the recommendation as written: split reverts ZeroAmount when `minted == 0` or `Math.mulDiv(minted, ONE, idx) == 0`, and `previewSplit` returns 0 in the same case so the preview and the split agree.

Test added with the report's example at index 1.5: a one-unit deposit is refused (`previewSplit 0`, split reverts ZeroAmount), while two units mint 3 PT that merge back to 2 shares. The test fails on the previous code, where the one-unit deposit minted 1 PT.

[I-09] ERC-4626 rate sampling truncates sub-base-unit share appreciation

Info

Fixed

LOCATION

ERC4626RateAdapter:rate:26-29

SUMMARY

The adapter is meant to report an ERC-4626 share's underlying value at WAD precision so the YieldVault can split principal owed to PT holders from yield owed to YT holders. To do this, `rate()` first converts exactly one share to assets, then rescales:

```
uint256 assets = vault.convertToAssets(oneShare);
r = Math.mulDiv(assets, 1e18, assetUnit);
if (r == 0) revert ZeroRate();
```

ERC-4626 `convertToAssets` rounds down to whole underlying base units, and the rescale to WAD happens only after that truncation. When a share's appreciation stays below one underlying base unit, the conversion returns the same integer and the reported index does not move.

For example, if true per-share value grows from 1.1 to 1.9 underlying base units, `convertToAssets(oneShare)` still returns 1, so the index is unchanged even though real yield accrued.

A related edge case: a one-share conversion can round to zero and trigger `ZeroRate()` even when larger share quantities convert to positive assets, making the PT oracle unavailable for collateral that otherwise has value.

IMPACT

Yield attributable to YT holders is never booked by `_accrue()` because the index stays flat. That unrecognized appreciation is instead paid out to PT holders at maturity against the stale expiry index, or remains in vault backing.

This is a real accounting transfer from YT holders to PT redemptions, not merely a display-precision loss. On supported vaults with low per-share values and coarse underlying decimals it can cause persistent drift.

Because the probe moves only when small source-vault value changes cross an underlying base-unit boundary, the rate also changes in discontinuous jumps. If settlement freezes a rounded-down pre-maturity index, a later integer crossing can be misclassified as post-maturity treasury yield.

EXPLOIT PATH

1. A user splits one ERC-4626 share while `convertToAssets(oneShare)` returns 1 base unit; the adapter records the corresponding WAD index and mints PT/YT.
2. Yield raises the share's true value (for example 1.1 to 1.9 base units), but the conversion still returns 1 due to floor rounding.
3. The YieldVault sees no index increase, so YT transfers and `claimInterest()` book zero interest.
4. After maturity, the PT holder redeems against the unchanged expiry index and receives the full reserved share, including the unreported appreciation.

PREREQUISITES

A supported ERC-4626 share with a per-share value whose increase stays below one underlying base unit after `convertToAssets(oneShare)` rounds down (low per-share value and coarse underlying decimals), with a user having split into PT and YT before the share appreciates.

RECOMMENDATION

Sample a larger number of shares so the conversion retains fractional-share information before rescaling:

```
uint256 assets = vault.convertToAssets(sampleShares);
r = Math.mulDiv(assets, 1e18, assetUnit * sampleMultiplier);
if (r == 0) revert ZeroRate();
```

Alternatively, use an adapter-specific exchange-rate interface that returns a higher-precision rate directly, while keeping the YieldVault's nondecreasing-index checks.



RESOLUTION

Pare

Fixed in <https://github.com/pare-audit/contracts/commit/f51af77>

Took the first recommendation: `rate()` now samples `oneShare * 1e18` and rescales with `assetUnit * 1e18` as the denominator, so the conversion keeps 18 decimals of fractional share value before the floor. The `ZeroRate` edge is covered by the same change: a share worth less than one base unit now reports its rate.

Test added with a 4626 share over a 6-decimal asset: a per-share value of 1.5000004 reports 1.5000004e18 where the previous code reported 1.5e18, and a per-share value of 0.4 base units reports 4e11 where the previous code reverted `ZeroRate`.

[I-10] Extremely small backing ratios round to a zero PT oracle price

Info

Fixed

LOCATION

`PtLinearOracle:backing:105-111`

SUMMARY

The oracle expresses what fraction of each PT is still backed by the current share rate. Any nonzero share rate should yield a nonzero backing ratio whenever the arithmetic can represent it.

Instead, the ratio floors to zero when remaining backing is below one WAD unit relative to the stored index:

```
uint256 r = adapter.rate();
return r >= idx ? WAD : Math.mulDiv(r, WAD, idx);
```

For example, with `idx = 2e18` and `r = 1`, `Math.mulDiv(1, 1e18, 2e18)` returns zero. Once backing is zero, `ptUsd()` and `price()` are also zero even though the adapter still reports positive backing.

IMPACT

A severe but still positive share-rate loss makes the Morpho oracle report a zero PT price rather than a small positive value.

PT-backed positions can become liquidatable as a result, though in practice the underlying collateral value is already near zero when this occurs.

EXPLOIT PATH

1. A vault has `expiryIndex = 2e18`, and its supported share adapter later returns `r = 1`.
2. `backing()` evaluates `Math.mulDiv(1, 1e18, 2e18)` and stores zero.
3. A Morpho health check calls `price()`, which propagates the zero backing into a zero PT price.
4. PT-backed borrowers can be marked liquidatable even though the adapter rate is technically nonzero.

PREREQUISITES

A Morpho market must use this oracle, the supported share rate must fall to a positive value satisfying $r * 1e18 < expiryIndex$ (or $r * 1e18 < pyIndexStored$), and PT-backed debt positions must remain open when the oracle is queried.

RECOMMENDATION

Preserve a nonzero result for every nonzero adapter rate, recognizing this only affects values below the current fixed-point resolution:

```
uint256 ratio = Math.mulDiv(r, WAD, idx);
if (ratio == 0 && r != 0) ratio = 1;
return ratio;
```

Alternatively, compute the oracle ratio at a larger internal precision and scale down only at the final Morpho boundary.

RESOLUTION

Pare

Fixed in <https://github.com/pare-audit/contracts/commit/7c13c8d>

Applied the recommended floor in `backing()`: a nonzero rate now yields at least one WAD unit. On its own that would still have left `price()` at zero, because `price()` computed `curve x backing / WAD` before applying the Morpho scale factor, so we also folded the scale factor into that single division (the alternative the recommendation describes: scale down only at the final Morpho boundary). Ordinary prices are unchanged apart from the trailing digits the earlier double floor dropped.

Test added: with the index at 2.0 and the adapter reporting a rate of 1 wei, `backing()` is 1 and `price()` is 850,000 ($0.85 \text{ curve} \times 1e-18 \text{ backing} \times 1e6 \text{ scale}$) both with and without feeds, where the previous code returned 0 for both; a 50% loss still prices exactly as before.

[I-11] Immutable Chainlink decimal scaling becomes stale after feed decimal changes

Info

Fixed

LOCATION

ChainlinkRateAdapter:constructor:20-25

SUMMARY

The adapter caches the feed's decimal precision once at construction and reuses it forever. The constructor stores the initial precision in the immutable `unit`, and `rate()` always divides by that cached value.

Chainlink feed addresses stay constant across aggregator migrations, but a migration can change the number of decimals the feed reports. When that happens the adapter keeps using the old scale, so answers are interpreted against the wrong precision.

For example, if a feed reports 8 decimals at deployment, `unit` becomes `1e8`. After a migration to 18-decimal answers, `rate()` still divides by `1e8` instead of `1e18`, producing a rate off by a power of ten. The adapter has no mechanism to detect or accommodate the change.

IMPACT

A post-deployment change in the feed's decimals mis-scales the reported exchange rate by a power of ten.

Depending on the direction of the change, this can propagate into incorrect PT/YT minting, index updates, interest accounting, redemption, and downstream PT oracle valuations.

EXPLOIT PATH

1. The factory deploys a `ChainlinkRateAdapter` for a proxy whose `decimals()` returns 8, so `unit` becomes `1e8`.
2. The proxy is migrated to an aggregator returning equivalent answers with 18 decimals.
3. A user calls `split()` or another operation that invokes `adapter.rate()`; the adapter divides the 18-decimal answer by `1e8` instead of `1e18`.
4. The vault records or uses a rate off by 10 orders of magnitude, resulting in incorrect accounting or oracle valuation.

PREREQUISITES

Requires an external event: the configured feed must later expose an aggregator whose `decimals()` differs from the value returned at adapter construction, and a vault or oracle operation must then consume the adapter rate.

RECOMMENDATION

Read the current decimals at call time and reject any change from the configured precision:

```
uint256 expectedUnit = unit;
uint256 currentUnit = 10 ** feed.decimals();
if (currentUnit != expectedUnit) revert DecimalChange();
r = Math.mulDiv(uint256(answer), 1e18, currentUnit);
```

Alternatively, if the deployment environment guarantees a stable feed interface, configure the adapter against a feed wrapper whose decimal precision is immutable.



RESOLUTION

Pare

Fixed in <https://github.com/pare-audit/contracts/commit/27e8d48>

Applied the recommendation: `rate()` reads `feed.decimals()` on every call and reverts `DecimalChange(unitNow)` when `10 ** decimals` differs from the unit cached at construction. The `YieldVault` reads that revert as an unavailable rate, so a migration that changes the feed's precision stops new splits and freezes the index rather than mis-scaling, while every exit stays open.

Test added: an 8-decimal feed rates normally, is refused once its decimals change to 18, and rates normally again when they return to 8. The test fails on the previous code (the 18-decimal answer was accepted against the 8-decimal scale).

[I-12] Loan and peg feed decimal migrations can mis-scale the PT oracle

Info

Fixed

LOCATION

`PtLinearOracle:constructor:76-105`

SUMMARY

The PT oracle caches each Chainlink feed's scale once at construction and reuses it for every future answer, instead of applying the scale that matches the live answer.

The loan path fixes the unit at deploy time:

```
loanUnit = address(loanFeed_) == address(0) ? 0 : 10 ** loanFeed_.decimals();

if (address(loanFeed) != address(0))
    v = Math.mulDiv(v, loanUnit, _read(loanFeed, loanMaxAge));
```

If the proxy later migrates to an aggregator reporting a different precision (for example 18 to 8 decimals), the fresh answer is combined with the stale unit. An 18-to-8 change leaves the old `1e18` unit multiplying an 8-decimal answer, producing roughly a `1e10` scale error. The same stale-scale mismatch can cause a peg feed's `depeg` to be ignored or exaggerated.

IMPACT

A decimals-changing Chainlink proxy migration can materially overvalue or undervalue the PT price feeding a dependent Morpho market.

That mis-scaled magnitude flows directly into collateral valuation, borrowing capacity, and liquidation thresholds, potentially creating bad debt or triggering erroneous liquidations.

EXPLOIT PATH

1. The oracle is deployed while the loan (or peg) feed reports 18 decimals, fixing its cached unit at `1e18`.
2. The Chainlink proxy is migrated to an aggregator reporting the same price with 8 decimals.



3. `price()` reads the fresh 8-decimal answer but still multiplies by `1e18`.
4. The resulting PT price is materially mis-scaled, driving excessive borrowing capacity or erroneous liquidations in the dependent market.

PREREQUISITES

A configured loan or peg Chainlink proxy must migrate to an aggregator implementation with a different `decimals` value while the immutable oracle stays in use, and a dependent Morpho market must keep consuming the oracle output. The proxy address and answers remain otherwise valid, so the stale precision arises purely from the aggregator implementation change.

RECOMMENDATION

Read the current decimals and reject any deviation from the expected deployment scale before applying the answer:

```
function _readExpected(IFeed f, uint8 expectedDecimals, uint256 maxAge)
internal view returns (uint256 answer)
{
    if (f.decimals() != expectedDecimals) revert BadDecimals();
    answer = _read(f, maxAge);
}
```

Alternatively, compute the unit from the current `f.decimals()` on every read and add monitoring or a circuit breaker for unexpected metadata changes. For an immutable oracle, rejecting a change is safer than silently recalculating against an unreviewed scale.

RESOLUTION

Pare

Fixed in <https://github.com/pare-audit/contracts/commit/98681cd>

Applied the recommendation's rejecting form: `_read` now takes the unit cached at construction, reads `f.decimals()` on every call, and reverts `DecimalChange(feed, unitNow)` when `10 ** decimals` differs. Both the peg path (`peg()`, `price()`) and the loan path go through `_read`, so one check covers both feeds.

Test added: an oracle with both feeds prices normally, reverts `DecimalChange` for the loan feed once it reports 18 decimals, reverts for the peg feed (on both `price()` and `peg()`) once it reports 6, and prices normally again when both return to 8. The test fails on the previous code, which kept pricing against the cached scale.

[I-13] Long-dated PT oracle configurations cause an extended price outage

Info

Fixed

LOCATION

PtLinearOracle:curve:68-74

SUMMARY

The constructor accepts a discount slope and maturity that are each individually valid but jointly produce a discount of 100% or more before maturity. A 50% annual slope (`baseDiscountPerYear = 0.5e18`) is accepted because it equals, rather than exceeds, `MAX_DISCOUNT_PER_YEAR` , and a maturity several years out is likewise accepted.

`curve()` then reverts whenever the computed discount reaches `WAD` :

```
uint256 left = maturity > block.timestamp ? maturity - block.timestamp : 0;
uint256 discount = Math.mulDiv(left, baseDiscountPerYear, YEAR);
if (discount >= WAD) revert CurveBelowZero();
return WAD - discount;
```

For example, a three-year series with a 50% annual slope yields an initial discount of ~150%, and a 40% slope over three years yields 120%. Because `price()` depends on `ptUsd()` , which depends on `curve()` , the oracle reverts on every read until the remaining tenor drops below the threshold (under two years for the 50% case). The oracle is immutable, so there is no parameter to repair the deployment.

IMPACT

The oracle is unavailable for the initial portion of a valid long-dated series, potentially for months or years.

Every price-dependent consumer breaks during that window. Dependent Morpho-compatible markets cannot obtain a PT price, so collateral valuation, ordinary lending operations, and safety-critical liquidation flows all revert.

Recovering requires waiting until the remaining term falls below the discount threshold, or deploying a replacement market/oracle. This is a configuration-validation gap, not attacker-controlled price manipulation.

EXPLOIT PATH

1. Deploy a series with maturity roughly three years away and its oracle with `baseDiscountPerYear = 0.5e18` .
2. The constructor succeeds because the slope equals `MAX_DISCOUNT_PER_YEAR` and the maturity passes its own check.
3. A Morpho-compatible consumer calls `price()` while the computed discount still exceeds `WAD` .
4. `curve()` reverts, suspending all oracle-dependent market operations until the remaining tenor falls below the threshold.

PREREQUISITES

A deployed oracle whose remaining maturity and slope satisfy `left * baseDiscountPerYear / YEAR ≥ 1e18` , with a dependent market relying on it. No malicious input is required; accepted constructor parameters alone trigger the outage.

RECOMMENDATION

Validate the full curve domain at deployment time with overflow-safe arithmetic, so a series that would

ever exceed 100% discount is rejected:

```
uint256 left = maturity_ - uint64(block.timestamp);
if (Math.mulDiv(left, baseDiscountPerYear_, 1) >= YEAR * WAD) {
    revert CurveBelowZero();
}
```

Alternatively, if the pricing policy allows, clamp the discount rather than reverting:

```
uint256 discount = Math.mulDiv(left, baseDiscountPerYear, YEAR);
if (discount >= WAD) discount = WAD - 1;
return WAD - discount;
```

RESOLUTION

Pare

Fixed in <https://github.com/pare-audit/contracts/commit/af5928a>

Took the first recommendation: the constructor evaluates the discount at deployment with the same expression `curve()` uses, `Math.mulDiv(maturity - now, baseDiscountPerYear, YEAR)`, and reverts `CurveBelowZero` when it is at least `WAD`. The discount only decreases from there, so a slope-and-tenor pair that passes at deployment can never reach 100% later.

Test added: a series maturing in three years refuses a 50% slope at construction and accepts a 30% slope, which prices at 0.1 from day one. The test fails on the previous code, which deployed the 50% oracle and then reverted on every `price()` read.

[I-14] Missing L2 sequencer status and recovery grace period in Chainlink pricing

Info

Fixed

LOCATION

PtLinearOracle::_read:144-150

SUMMARY

PT pricing validates only the age of a Chainlink answer, with no check on L2 sequencer status or a post-recovery grace period. A feed answer can be recent relative to `maxAge` yet have been published before a sequencer outage.

```
(, int256 answer,, uint256 updatedAt,) = f.latestRoundData();
if (answer <= 0) revert BadAnswer(address(f), answer);
if (maxAge != 0 && updatedAt + maxAge < block.timestamp)
    revert StaleAnswer(address(f), updatedAt);
return uint256(answer);
```

Because no sequencer uptime feed or recovery timestamp is consulted, a pre-outage price that is still within `maxAge` is accepted during the recovery window.

IMPACT

On an affected L2, PT collateral can be valued using a stale pre-outage peg, loan-token, or backing price during recovery.

If the asset lost value while the sequencer was unavailable, borrowing and liquidation decisions use an overstated collateral value, allowing borrowing against overvalued PT collateral and leaving lenders exposed to bad debt.

EXPLOIT PATH

1. A Morpho-compatible market uses PtLinearOracle with a peg, loan-token, or Chainlink-backed share-rate input on an L2.
2. The L2 sequencer goes down after the last feed update, and the underlying asset loses value during the outage.
3. The sequencer recovers before the configured `maxAge` expires, but the feed has not yet published a reliable post-recovery answer.
4. A borrower uses the still-accepted pre-outage price to borrow more than the collateral is actually worth, creating potential lender bad debt.

PREREQUISITES

The market is deployed on an L2 where sequencer availability affects feed reliability, a configured feed's pre-outage answer remains within `maxAge` after recovery, and the relevant asset loses value during the outage or recovery window before a fresh feed update.

RECOMMENDATION

Add a sequencer uptime feed and enforce both an outage check and a recovery grace period before accepting prices:

```
(uint80 roundId, int256 answer, , uint256 updatedAt, ) = feed.latestRoundData();
(, int256 status, , uint256 statusUpdatedAt, ) = sequencer.latestRoundData();
if (status != 0 || block.timestamp - statusUpdatedAt <= recoveryGrace)
    revert SequencerUnavailable();
if (answer <= 0 || block.timestamp - updatedAt > maxAge)
    revert InvalidOrStaleAnswer();
```

Apply equivalent validation to the adapter used by `backing()` , not only the peg and loan-token feeds.



RESOLUTION

Pare

Fixed in <https://github.com/pare-audit/contracts/commit/601768e>

Added SequencerGuardedFeed, a wrapper for L2 deployments: it forwards decimals() and latestRoundData() from the wrapped feed, but reverts SequencerDown while the chain's sequencer uptime feed reports the sequencer down (or has no round yet) and SequencerGrace for a configured period after it comes back, per Chainlink's L2 uptime feed convention (answer 0 = up, startedAt = when it came up). PtLinearOracle's peg and loan feeds and ChainlinkRateAdapter's rate feed (the backing() input) all read through the same two functions, so wrapping each feed applies the check on all three paths, as requested, without changing the immutable oracle or adapter; an L1 deployment passes the raw feeds.

Test added: wrapped peg, loan and rate feeds answer normally; while the sequencer is down the still-fresh pre-outage answers are refused by both the oracle and the adapter; during the grace period they are refused; after it they answer again.

[I-15] Per-call merge rounding can strand backing shares

Info

Fixed

LOCATION

YieldVault:merge:218-227

SUMMARY

Burning equal PT and YT amounts should return the share-denominated backing of the position. The implementation rounds every merge call independently and discards the remainder, so when the index sits above one, a small per-call merge can return zero shares even though several such units merged together would be redeemable.

```
yt.vaultBurn(msg.sender, amount);
pt.vaultBurn(msg.sender, amount);
shares = expiryIndex == 0
? Math.mulDiv(amount, ONE, idx)
: _payMaturedPT(amount);
share.safeTransfer(receiver, shares);
```

Chunking a position can therefore burn all of its PT and YT while the backing shares stay trapped in the vault with no remaining ownership claim.

IMPACT

A holder can permanently lose rounding remainders by merging in chunks. For high-decimal tokens this is dust, but for coarse-denomination share tokens it can amount to an entire position. The lost value is not transferred to another user, it is simply stranded in the vault.

EXPLOIT PATH

1. The user splits two raw share units at index $1.5e18$ and receives three PT and three YT units.



2. The user calls `merge(1, receiver)` ; one PT and one YT are burned and zero shares are transferred.
3. The user repeats `merge(1, receiver)` twice; the remaining PT/YT are burned and each call again transfers zero shares.
4. The entire position is gone while the vault still holds the original two share units with no claim assigned to the user.

PREREQUISITES

A pre-maturity position must use a share token whose raw denomination is coarse enough for per-call floor rounding to be material, and the holder must merge the position across multiple calls rather than in a single call.

RECOMMENDATION

Maintain a per-user fractional remainder in share precision and carry it into the next merge payout:

```
uint256 gross = amount * ONE + mergeRemainder[msg.sender];
shares = gross / idx;
mergeRemainder[msg.sender] = gross % idx;
```

Alternatively, reject zero-payout merges so users cannot accidentally destroy a position without receiving shares:

```
shares = Math.mulDiv(amount, ONE, idx);
if (shares == 0) revert ZeroAmount();
```

RESOLUTION

Pare

Fixed in <https://github.com/pare-audit/contracts/commit/ca6d4e0>

Took the second option: merge reverts ZeroAmount when the share payout floors to zero, so PT and YT are never burned for nothing and a holder is steered to merge at least a whole share unit's worth at a time. redeemPT gets the same check, since its post-maturity payout floors the same way.

Test added with the report's example at index 1.5: two units mint 3 PT and 3 YT; merge(1) is refused and the position is untouched, merge(3) returns the 2 units; after maturity redeemPT(1) is refused and redeemPT(3) returns 2. The test fails on the previous code, where merge(1) burned the tokens and returned 0.

[I-16] Per-holder interest-fee rounding allows fee evasion by splitting YT

Info

Fixed

LOCATION

YieldVault:_accrue:169-181

SUMMARY

The protocol intends `interestFeeBps` of all YT interest to be booked to `feesOwed` regardless of how YT ownership is distributed. YT transfers correctly checkpoint balances so accrued interest stays with the old holder, but the fee calculation does not preserve the same aggregate result across equivalent ownership splits.

`_accrue()` rounds the fee down independently for every address. With a 5% fee, each address earning 10 shares pays zero because `Math.mulDiv(10, 500, 10000)` rounds down, while a single address earning 1,000 shares pays 50 shares.

Because each recipient is checkpointed independently, a holder can fragment YT among controlled addresses before yield accrues so that every small per-holder accrual falls below one fee unit and rounds to zero. The same effect arises from ordinary YT transfers or repeated checkpoints (including repeated zero-value transfers) that split one yield period into many small accrual intervals. Once `userIndex` advances, the lost fee fractions can never be recovered.

IMPACT

YT holders can permanently retain interest that should have gone to the treasury. The avoided fee is repeatable and scales with the number of recipient accounts and the amount of interest.

This creates a real accounting shortfall in `feesOwed` and makes protocol revenue depend on the number of holder addresses rather than the aggregate interest generated. The loss is bounded rounding dust per checkpoint, but it accumulates across many small accruals.

The issue also exists as a general dust-fee waiver whenever any individual accrual is below one fee unit, even without deliberate fragmentation.

EXPLOIT PATH

1. A user splits 2,000 zero-decimal share units at index $1e18$, receiving 2,000 PT and 2,000 YT.
2. Before the index changes, the user transfers 20 YT to each of 100 controlled addresses; each recipient is checkpointed at index $1e18$.
3. The share rate rises and the index becomes $2e18$. Each 20-YT balance accrues 10 shares of interest, and each 5% fee rounds to zero.
4. The 100 addresses claim 1,000 shares in total while the treasury receives none of the 50-share fee that would have been charged on a consolidated position.

PREREQUISITES

The series must charge a nonzero interest fee, and the holder must be able to distribute YT across multiple addresses (or trigger repeated checkpoints via permissionless `claimInterest()` / transfers) before an index increase. The individual interest amounts must be small enough that per-holder fee rounding removes the intended fee.

RECOMMENDATION

Track fractional fee remainders per holder and carry them into subsequent fee calculations so rounding



cannot be reset by transfers or address fragmentation:

```
mapping(address => uint256) public feeRemainder;

uint256 feeNumerator = interest * interestFeeBps + feeRemainder[user];
uint256 fee = feeNumerator / BPS;
feeRemainder[user] = feeNumerator % BPS;
feesOwed += fee;
interestOwed[user] += interest - fee;
```

Note that for this to fully hold, remainders must follow the YT position during transfers; otherwise partitioning still creates new rounding buckets.

Alternatively, calculate fees against an aggregate series-level interest amount before distributing net interest, if the desired economics are series-wide rather than holder-specific.

RESOLUTION

Pare

Fixed in <https://github.com/pare-audit/contracts/commit/497e7a6>

We took a different route from the per-holder remainder, for the reason the finding itself gives: a remainder does not follow YT across transfers, so fragmentation would still open fresh rounding buckets. Instead the fee is now rounded up (`Math.Rounding.Ceil`) in `_accrue`, and identically in the `pendingInterest` view. Every accrual therefore pays at least one fee unit, and splitting a position across addresses or checkpoints can only increase the fee paid, never reduce it. The over-collection is bounded at one share unit per checkpoint and goes to the treasury.

Test added with the report's example: 2000 units split as 20 YT to each of 100 addresses, index 1 - > 2; the treasury books at least the fee a single holder would pay on the same interest, and the sum of interest paid and fee booked equals the 1000 shares accrued. The test fails on the previous code, which booked 0 fee.

[I-17] Per-redemption rounding can strand post-maturity yield outside treasury accounting

Info

Fixed

LOCATION

YieldVault:_payMaturedPT:244-252

SUMMARY

After maturity, yield earned by unredeemed PT is meant to accrue to the treasury while the PT holder receives only the principal-valued share amount. The problem is that the reserved principal and the

redemption payout are computed and floor-rounded independently on every call:

```
uint256 reserved = Math.mulDiv(amount, ONE, expiryIndex);
shares = Math.mulDiv(amount, ONE, rate);
uint256 swept = reserved - shares;
if (swept != 0) feesOwed += swept;
```

Because `feesOwed` only ever receives each chunk's rounded difference, splitting a redemption into small pieces can drive every chunk's `swept` to zero even when the aggregate reserved-minus-paid value is positive. The omitted shares stay in the vault with no treasury claim.

For example, a series settled at `expiryIndex = 1.1e18` with a post-maturity rate of `1.2e18` : redeeming 36 PT in one call books $\text{floor}(36/1.1) - \text{floor}(36/1.2) = 32 - 30 = 2$ shares for the treasury, but six separate 6-PT redemptions each floor to zero and book nothing.

IMPACT

Fragmented PT redemption can permanently leave post-maturity yield in the vault without assigning it to `feesOwed` or the treasury. The PT holder's payout is not increased, so this is not a user theft, but treasury entitlement and vault solvency accounting become inaccurate.

The residual persists even after the final PT is burned, because the configured share token is protected from skim and there is no final reconciliation path. The result is a permanent share-token accounting imbalance once all PT obligations are redeemed.

A sufficiently small partial redemption can also pay zero shares while still burning the PT, making redemption granularity directly destructive. Conversely, the same defect makes redemption batching economically meaningful, since a single aggregate redemption books yield that fragmented redemptions omit.

EXPLOIT PATH

1. Settle a series at `expiryIndex = 1.1e18` ; after maturity the adapter reports `1.2e18` and a holder owns 36 PT.
2. Redeem 6 PT: the vault burns the PT, transfers 5 shares, and records zero post-maturity yield.
3. Repeat the 6-PT redemption five more times; total payout is 30 shares and `feesOwed` is unchanged.
4. Compare against one 36-PT redemption, which also pays 30 shares but records 2 shares for the treasury.

PREREQUISITES

The series must be post-maturity with the live rate above `expiryIndex` , and PT redemption amounts must be small enough that per-call floor rounding eliminates the reserved-minus-paid difference.

RECOMMENDATION

Track the aggregate reserved obligation and derive the sweep from aggregate values rather than per-call independent floors:

```
uint256 reservedBefore = remainingPT * ONE / expiryIndex;
remainingPT -= amount;
uint256 reservedAfter = remainingPT * ONE / expiryIndex;
uint256 paid = amount * ONE / currentRate;
uint256 expectedSweep = reservedBefore - reservedAfter - paid;
feesOwed += expectedSweep;
```



Alternatively, use a higher-precision fixed-point accumulator for redemption rounding and carry residual fractions across redemptions until they form a whole share, reconciling any final residual when all PT has been redeemed.

RESOLUTION

Pare

Fixed in <https://github.com/pare-audit/contracts/commit/47f6042>

Took the first recommendation, using the PT token's own supply as the aggregate. `_payMaturedPT` now computes the shares reserved for the whole outstanding PT supply before and after this redemption and books the difference, minus what the holder is paid, to `feesOwed`. The per-call floor rounding no longer hides the sweep across chunked redemptions.

Test added with the report's example: expiryIndex 1.1, post-maturity rate 1.2, 36 PT redeemed as six calls of 6. The holder receives 30 shares and the treasury is booked 2, the same as one 36 PT redemption. The previous code booked 0 for the chunked case.

[I-18] Permissionless pool initialization permits hostile initial PT prices

Info

Fixed

LOCATION

PtPoolHook:beforeInitialize:123-128

SUMMARY

The hook that guards PT pool initialization authenticates neither the initializer nor the initial price. In `beforeInitialize` it only checks that one of the pool currencies is the configured PT and that the pool uses the dynamic fee flag.

Because PoolManager treats initialization as a one-time operation per PoolKey, any untrusted caller can initialize the intended pool first with any valid `sqrtpPriceX96`, including an economically extreme value. The legitimate initialization then reverts as already initialized, and the pool stays anchored at the attacker's price.

IMPACT

An attacker can permanently consume the intended PT pool key and force the pool to launch at an attacker-selected price.

The deployer is forced to either abandon that key or operate a pool that started at a hostile price. Liquidity providers, routers, and integrations that assume the intended launch price may face adverse execution or downstream losses.

This does not alter the vault's oracle, but it permanently disrupts the intended pool deployment.

EXPLOIT PATH

1. The attacker observes a pending `initialize(PoolKey, intendedPrice)` transaction for an uninitialized PT and dynamic-fee pool.
2. The attacker submits `initialize(PoolKey, hostilePrice)` first, where `hostilePrice` is a valid but

economically extreme `sqrPriceX96` .

3. `beforeInitialize` passes (a currency equals the immutable PT and the fee is `DYNAMIC_FEE_FLAG`), so `PoolManager` records `hostilePrice` .
4. The intended transaction reverts as already initialized, leaving the deployer with a hostile pool or an abandoned key.

PREREQUISITES

A pending initialization transaction must expose a not-yet-initialized `PoolKey`, and the attacker must be able to submit a competing `PoolManager.initialize` with that same key ahead of it.

RECOMMENDATION

Restrict initialization to an authorized initializer and bind the exact expected price via a deployment commitment, rather than relying only on numerical price bounds:

```
mapping(bytes32 => uint160) public committedPrice;

function beforeInitialize(address sender, PoolKey calldata key, uint160 sqrPriceX96)
external
view
onlyPoolManager
returns (bytes4)
{
    if (sender != authorizedDeployer) revert Unauthorized();
    if (sqrPriceX96 != committedPrice[keccak256(abi.encode(key))]) revert BadInitialPrice();
    // existing PT and dynamic-fee checks
    return IHooks.beforeInitialize.selector;
}
```

Alternatively, deploy and initialize through a trusted factory in a single transaction and have the factory verify the recorded initial price before adding liquidity. A bounded acceptable-price check is a weaker fallback that mitigates extreme prices but does not prevent the launch race.

RESOLUTION

Pare

Fixed in <https://github.com/pare-audit/contracts/commit/e9b7a53>

Took the first recommendation, restricting initialization to an authorized initializer.

`beforeInitialize` now requires the initializer to be the vault's owner, read live from the vault. That is the account that creates the series, the oracle and the hook, and the deploy script already initializes the pool from it. Any other caller reverts with `NotInitializer`, so the intended key can no longer be taken first at a hostile price.

We did not add a per-pool price commitment. With a single trusted initializer, the price it submits is the committed price, and a commitment mapping would need a setter the hook has no owner for.

Test added: an attacker's initialize at a hostile price reverts with the wrapped `NotInitializer` error, then the owner initializes the same key at the intended price. The previous code let the attacker's initialize through.



[I-19] PT oracle accepts feed observations dated in the future

Info

Fixed

LOCATION

PtLinearOracle:_read:139-143

SUMMARY

The oracle's `_read()` validates that a feed observation is positive and not stale, but never checks that `updatedAt` is no later than the current block. A future-dated observation passes the staleness check and is returned to the caller.

```
if (answer <= 0) revert BadAnswer(address(f), answer);
if (maxAge != 0 && block.timestamp - updatedAt > maxAge) {
    revert StaleAnswer(address(f), updatedAt);
}
return uint256(answer);
```

Because `peg()` and `price()` consume this value directly, a malformed feed response produces an invalid PT valuation rather than reverting. A future timestamp can also keep an otherwise-stale answer eligible beyond the intended freshness interval as later blocks advance.

IMPACT

The Morpho-compatible oracle can return a price derived from an observation that has not occurred at the current block time, yielding invalid PT valuation or oracle disruption until the feed response is corrected.

Both optional feeds are exposed because they share the same `_read()` validation path:

- A future-dated loan-token feed can supply an erroneously low denominator, inflating the denominator-based conversion in `price()` and overvaluing collateral.
- A future-dated peg observation can keep a depegged asset priced at par.

These two paths can distort collateral pricing independently or together.

EXPLOIT PATH

1. A configured peg or loan feed returns a positive answer with `updatedAt` later than the current block timestamp.
2. A Morpho-compatible consumer calls `PtLinearOracle.price()`.
3. `_read()` evaluates `updatedAt + maxAge < block.timestamp` as false and accepts the answer.
4. The future-dated value is incorporated into the peg or loan-token conversion, returning an invalid PT price.

PREREQUISITES

A configured peg or loan feed must return a positive answer with `updatedAt > block.timestamp`. The relevant feed may have `maxAge` set to zero or a nonzero value.

RECOMMENDATION

Reject future-dated timestamps before the age calculation:

```
function _read(IFeed f, uint256 maxAge) internal view returns (uint256) {
    (, int256 answer, , uint256 updatedAt) = f.latestRoundData();
    if (answer <= 0) revert BadAnswer(address(f), answer);
    if (updatedAt > block.timestamp) revert FutureAnswer(address(f), updatedAt);
    if (maxAge != 0 && block.timestamp - updatedAt > maxAge) {
        revert StaleAnswer(address(f), updatedAt);
    }
    return uint256(answer);
}
```

Apply the same future-timestamp check consistently to every feed adapter that consumes external timestamps.

RESOLUTION

Pare

Fixed in <https://github.com/pare-audit/contracts/commit/d687a51> (the I-06 commit), with the loan-feed test added in <https://github.com/pare-audit/contracts/commit/8e2635b>

`PtLinearOracle._read` now reverts with `FutureAnswer` when `updatedAt > block.timestamp`, before the age calculation, as recommended. Both the peg feed and the loan feed go through `_read`, so both paths are covered. The same check is in `ChainLinkRateAdapter`, which is the only other place a feed timestamp is consumed. The ERC-4626 adapter reads no external timestamp, and `SequencerGuardedFeed` forwards `latestRoundData` unchanged to those consumers.

Tests: a future-dated peg observation (from I-06) and a future-dated loan observation (this commit) each revert with `FutureAnswer`, and the price is back once the feed's timestamp is current. Without the check, both fail.

[I-20] Settlement can lock a lagging observation before maturity yield is published

Info

Fixed

LOCATION

YieldVault:_updateIndex:151-161

SUMMARY

At maturity the fixed index should capture share value at or after maturity, so that all yield earned through maturity stays with YT holders and only later appreciation goes to treasury.

Instead, `_updateIndex()` accepts the latest fresh observation regardless of when it was published. Once `block.timestamp` reaches maturity, it permanently stores whatever the adapter returns in `expiryIndex`, even if that observation predates maturity. A later feed update cannot change `expiryIndex`, and the subsequent increase is processed by `_payMaturedPT` as treasury yield.

Because settlement is permissionless and feed heartbeat lag is a normal external condition, the allocation

depends purely on whether settlement runs before or after the next feed publication. It needs neither a malicious administrator nor an invalid feed answer.

IMPACT

Pre-maturity appreciation that is published after settlement can be diverted from YT interest accounting to post-maturity treasury yield. This can materially shift the economic allocation between YT holders and PT holders.

EXPLOIT PATH

1. The feed publishes rate 1.0 shortly before maturity, while the underlying share's economic rate rises to 1.1 before maturity.
2. Immediately after maturity a caller invokes `settle()` ; the adapter still returns the fresh 1.0 observation, so `expiryIndex` becomes 1.0.
3. The feed later publishes 1.1, but `expiryIndex` stays fixed because `_updateIndex()` returns the existing value.
4. Redeeming PT at 1.1 books the 1.0 to 1.1 increase as post-maturity treasury yield instead of YT accrual.

PREREQUISITES

A supported rate source must publish a fresh-but-lagging observation across maturity, and a permissionless caller must settle before the next observation is available.

RECOMMENDATION

Expose the observation timestamp from the adapter and require a post-maturity observation before fixing `expiryIndex` :

```
(bool ok, uint256 r, uint256 updatedAt) = _tryRateWithTimestamp();
if (block.timestamp >= maturity && updatedAt < maturity) {
    return pyIndexStored;
}
expiryIndex = ok ? Math.max(r, pyIndexStored) : pyIndexStored;
```

Alternatively, introduce a settlement delay or an explicit oracle checkpoint that guarantees the rate used for `expiryIndex` was observed at or after maturity.

RESOLUTION

Pare

Fixed in ee07f8a, the same change as under M-01. The settlement change you suggested there is what it does: the first reading taken at or after maturity opens a window, and once `RATE_DELAY` has run the lowest reading in it is credited and becomes `expiryIndex`, never lower than the index already credited. The value fixed is now a rate observed at or after maturity, not the credited index from before it. `test_settlementWaitsForAReadingTakenAtOrAfterMaturity` now settles at the feed's post-maturity 1.1 rather than 1.0, and `test_continuousGrowthReadDailyPaysYTToMaturity` covers the 180-day case: `expiryIndex` equals the maturity reading, YT is paid every bit of yield to maturity, and only post-maturity yield is swept. The one fallback to an earlier value is the I-24 time limit (b777172), where the feed gives no readings at all.



[I-21] skim does not settle the vault on the first post-maturity interaction

Info

Fixed

LOCATION

YieldVault:skim:290-297

SUMMARY

The first vault interaction at or after maturity is supposed to fix `expiryIndex`, which stops YT from earning against adapter rate changes beyond the fixed-maturity boundary.

`skim()` is a permissionless interaction that can transfer reward or airdrop tokens to the treasury, but it never calls `_updateIndex()`. A nonzero post-maturity skim can therefore complete while `expiryIndex` stays zero.

The next YT checkpoint then fixes the index at the later live adapter rate, folding the intervening rate increase into accrued interest. The reward token is not the source of the error; the defect is that a vault state transition is omitted from a permissionless post-maturity path.

IMPACT

YT holders can receive post-maturity interest produced by the gap between a skim and the next settling action.

The issue needs a non-share token balance in the vault but requires no attacker privilege and no control over the rate source.

EXPLOIT PATH

1. The series matures with `expiryIndex == 0`, and the vault holds a non-share reward token with a nonzero balance.
2. Any caller invokes `skim(rewardToken)`; the reward is sent to the treasury, but no index settlement occurs.
3. The adapter rate increases while `expiryIndex` remains zero.
4. A later `claimInterest(holder)` or YT transfer settles at the later rate and books the intervening post-maturity growth as YT interest.

RECOMMENDATION

Make `skim` settle the index before processing the token balance and transfer.

```
function skim(IERC20 token) external nonReentrant returns (uint256 amount) {
    _updateIndex();
    if (address(token) == address(share)) revert CannotSkimShare();
    amount = token.balanceOf(address(this));
    if (amount == 0) return 0;
    token.safeTransfer(treasury, amount);
}
```



RESOLUTION

Pare

Fixed in <https://github.com/pare-audit/contracts/commit/0f963ac> (the I-07 commit)

`skim` now calls `_updateIndex()` first, before the token check and transfer, as recommended. `claimFees` got the same in that commit, so every permissionless path takes part in the maturity transition.

The test from that commit covers this exact scenario: the rate rises before maturity without being credited, `skim` is the first interaction at maturity, `expiryIndex` is fixed there, and a YT holder's claim five days later pays nothing for the rise after it. On the previous code the claim settled at the later rate.

[I-22] Skimming vault-held YT strands interest booked to the vault address

Info

Fixed

LOCATION

YieldVault:skim:290-298

SUMMARY

When the vault sends YT it holds to the treasury via `skim`, it should also move or settle the interest that YT accrued while held by the vault. It does not.

`skim` transfers the YT while the interest claim remains indexed under `address(vault)`. The transfer hook may book additional interest, but no code reassigns `interestOwed[address(vault)]` to the treasury. Calling `claimInterest(address(vault))` clears the claim through a self-transfer, so the shares move from the vault to itself rather than to the treasury.

As a result, the treasury receives the YT but cannot receive the interest accrued while the YT was held by the vault.

IMPACT

Interest earned by YT while held by the vault can become permanently unclaimable by the treasury after the YT is skimmed. The affected value is limited to the interest associated with vault-held YT.

EXPLOIT PATH

1. YT is transferred to the YieldVault, and the vault records a checkpoint for its YT balance.
2. The adapter rate rises from $1e18$ to $2e18$; a later YT transfer hook accrues interest to `interestOwed[address(vault)]`.
3. Anyone calls `skim(address(yt))`; the hook checkpoints the vault and the YT moves to the treasury, but the accrued claim stays under the vault address.
4. `claimInterest(address(vault))` clears the claim while transferring shares from the vault to itself, so the treasury keeps the YT without receiving its accrued interest.

PREREQUISITES

YT must be held by the vault and accrue interest before it is skimmed. The YT must be transferred after



the index increase so the transfer hook checkpoints the vault balance.

RECOMMENDATION

Before transferring vault-held YT, settle the vault's claim directly to the treasury and clear it only after the transfer succeeds:

```
if (address(token) == address(yt)) {
    _accrue(address(this), _updateIndex());
    uint256 interest = interestOwed[address(this)];
    interestOwed[address(this)] = 0;
    if (interest != 0) share.safeTransfer(treasury, interest);
}
```

Alternatively, reject PT/YT in `skim` and provide a dedicated recovery function that transfers the token and its associated claim atomically.

RESOLUTION

Pare

Fixed in <https://github.com/pare-audit/contracts/commit/314778a>

Took the first recommendation. When `skim` moves YT the vault holds, the YT transfer hook books what that YT earned under the vault's own address, and since `claimInterest` is holder-only (L-04) nobody could ever claim it. `skim` now pays that booked interest to the treasury, in shares, together with the YT. The payout comes after the share-balance guard, so the guard is unchanged. No separate accrue is needed before the transfer: the hook accrues the vault's balance at the index `skim` has just updated to, so `interestOwed[address(this)]` holds everything by the time the payout runs.

Test added: 40 YT sent to the vault, the rate rises to 1.1, anyone skims the YT. The treasury receives the YT and the interest it had pending, the vault's own claim is zero afterwards, and the rest of the holder's position merges as before. The previous code paid none of the interest.

[I-23] Stale transfer checkpoints reassign outage-period YT yield to buyers

Info

Fixed

LOCATION

YieldVault:beforeYieldTransfer:186-191

SUMMARY

A YT transfer should settle the seller's accrued interest up to the transfer time and start the buyer at that same valid transfer-time index, keeping pre-transfer yield with the seller.

During an adapter outage, `beforeYieldTransfer()` accepts the stored index as if it were the transfer-time

checkpoint and updates both accounts, even though the real rate may have risen:

```
uint256 idx = _updateIndex();
if (from != address(0)) _accrue(from, idx);
if (to != address(0) && to != from) _accrue(to, idx);
```

Because the checkpoint is committed against a stale index, no intervening interest is booked at transfer time. When the feed recovers, the buyer's balance accrues from that stale index across the entire outage period. The seller's balance has already moved and cannot receive the earlier yield. This is lazy accumulation assigning the entire recovered index delta to the current holder.

For example, a seller is checkpointed at index 1.0 and the underlying appreciates to 1.2 while the feed is stale. After recovery the buyer claims the full 1.0 to 1.2 increase on the transferred balance.

IMPACT

A YT seller permanently loses yield earned before the transfer, and the buyer receives it once the feed recovers.

The misallocation is not reversible by any later settlement or claim.

EXPLOIT PATH

1. A seller holds YT checkpointed at index 1.0; the underlying appreciates to 1.2 while the adapter feed is stale.
2. The seller transfers the YT during the outage; `beforeYieldTransfer()` uses stored index 1.0 and books no intervening interest.
3. The feed recovers and the adapter reports index 1.2.
4. The buyer claims interest and receives the full 1.0 to 1.2 increase on the transferred balance, while the seller cannot claim the pre-transfer yield.

PREREQUISITES

A YT balance must be transferred while the adapter cannot provide a current rate and before the feed later reports a higher index. Ordinary YT transfer access is sufficient; no special role is needed.

RECOMMENDATION

Require a valid rate for yield-token transfers so checkpoints are never committed at an unverified outage fallback:

```
function beforeYieldTransfer(address from, address to) external {
    if (msg.sender != address(yt)) revert OnlyYieldToken();
    (bool ok, uint256 r) = _tryRate();
    if (!ok) revert RateUnavailable();
    uint256 idx = _updateIndex();
    if (from != address(0)) _accrue(from, idx);
    if (to != address(0) && to != from) _accrue(to, idx);
}
```

Alternatively, implement an explicit pending-transfer mechanism that leaves ownership of the balance's accrued yield with the seller until a valid checkpoint can be established.



RESOLUTION

Pare

Fixed in <https://github.com/pare-audit/contracts/commit/2282027>

Took the recommendation, with the one read the index update already makes rather than a second `_tryRate().beforeYieldTransfer` now reverts with `RateUnavailable` when there is no rate, so a transfer is never checkpointed on an unverified index.

Two points of scope, both deliberate. Mints and burns are not transfers: split already needs a rate, and merge burns YT as an exit, so it stays open through an outage as exits always have (the rule L-06 relies on). And after settlement the index is fixed, so a transfer during an outage changes nothing and needs no rate.

One note on the example: since the M-01 fix a fresh reading is only a candidate, so a buyer cannot collect a full 1.0 to 1.2 jump in either version. The residual case is a rise that has held for `RATE_DELAY` and is due for credit: during an outage the index cannot move, so that credit would land at the next live read, with the buyer. The transfer now waits, so it lands with the seller.

Tests added: during an outage a transfer reverts while a merge pays at the recorded index; once the feed is back, the transfer's read credits the rise to the seller's remaining YT and the buyer claims nothing for it. A transfer after settlement goes through without a rate. The outage case passes on the previous code without reverting.

[I-24] Temporary rate outage at maturity permanently misallocates intervening yield

Info

Fixed

LOCATION

YieldVault:_updateIndex:150-161

SUMMARY

At maturity the vault should record a valid share-rate index that separates pre-maturity YT yield from post-maturity yield allocated to treasury. The implementation finalizes maturity even when the adapter read fails.

In `_updateIndex()`, a failed `_tryRate()` falls back to `pyIndexStored`, and the maturity branch permanently writes that value to `expiryIndex`:

```
(bool ok, uint256 r) = _tryRate();
idx = ok ? Math.max(r, pyIndexStored) : pyIndexStored;
if (block.timestamp >= maturity) {
    expiryIndex = idx;
    emit Settled(idx);
}
```

Once set, the fixed `expiryIndex` masks all later rate increases, and the assignment is irreversible: later calls return early even after the adapter recovers. Rate growth that recovers to a higher value, which may have accrued before maturity, is then classified by `_payMaturedPT()` as post-maturity yield.

For example, with `pyIndexStored = 1e18` at maturity during an outage, `expiryIndex` locks to `1e18` ; when the feed recovers to `1.2e18` , the `0.2e18` difference is swept to treasury fees instead of being paid to YT holders.

IMPACT

If the rate feed is unavailable at maturity and later recovers at a higher rate, yield that should belong to YT holders is instead treated as post-maturity PT yield and credited to `fees0wed` for the treasury.

This produces a series-wide accounting loss for YT holders and an unintended redistribution of value. The stale maturity denominator can also cause PT holders to redeem for excess shares, overpaying them at the recovered rate.

An ordinary, unprivileged caller can trigger this irreversible misstatement without controlling the feed.

EXPLOIT PATH

1. A series reaches maturity with `pyIndexStored = 1e18` ,but the adapter feed is stale and `_tryRate()` reverts.
2. Any caller invokes `settle()` , permanently setting `expiryIndex` to `1e18` .
3. The feed recovers and reports `1.2e18` ; YT checkpoints and claims continue using the fixed `1e18` index.
4. PT holders redeem at the recovered rate, and the difference between the `1e18` reserve and the redemption amount is added to `fees0wed` instead of being paid as YT yield.

PREREQUISITES

A series has outstanding PT/YT positions and the adapter is unavailable when a permissionless caller invokes `settle()` at or after maturity. The adapter later becomes available and reports a rate higher than the last stored index. No privileged access is required.

RECOMMENDATION

Require a successful adapter read before finalizing maturity:

```
function _updateIndex() internal returns (uint256 idx) {
    if (expiryIndex != 0) return expiryIndex;
    (bool ok, uint256 r) = _tryRate();
    if (!ok) revert RateUnavailable();
    idx = Math.max(r, pyIndexStored);
    if (idx != pyIndexStored) pyIndexStored = idx;
    if (block.timestamp >= maturity) expiryIndex = idx;
}
```

Alternatively, preserve the current fallback for non-finalizing operations but have `settle()` explicitly require `adapter.rate()` to succeed before writing `expiryIndex` . A pending-maturity state that defers finalization until a valid read is available, while still allowing other supported exit paths, also addresses the issue.



RESOLUTION

Pare

Fixed in <https://github.com/pare-audit/contracts/commit/b777172>. SETTLE_LIMIT puts a time limit on the wait: two RATE_DELAY windows plus two days after maturity (30 days for sUSDx). If by then there is no reading taken at or after maturity, or a settling window opened and the feed stopped before it could close, the series settles at the index credited so far. From that point PT alone redeems at it, and YT transfers no longer need a rate (the I-23 gate only applies while expiryIndex is 0), so a PT-only holder can always exit and a YT holder can always transfer. Your three permanent refusals (deprecated feed, rate above maxRate, DecimalChange) all land here. Tests: test_settlementHasATimeLimit and test_settlementTimeLimitWithTheWindowOpen, both failing on the previous code.