

Pashov Audit Group

Pare Security Review

October 11th 2026 - October 12th 2026

Contents

1. About Pashov Audit Group 3
2. Disclaimer 3
3. Risk Classification 3
4. Methodology 4
5. Executive Summary 4
6. Findings 4
Info Severity 5
[I-01] Dust-tick spray makes every YtRouter trade and quote unbounded in gas 5
[I-02] vault addresses are fetched twice per trade, and the approval reset after split is a redundant 0->0 write 6
[I-03] Front-Runner Can Take The Surplus Of A YT Buyer When PT Trades Above Par 7
[I-04] YtRouter constructor does not check that the periphery uses the same PoolManager and factory 9
[I-05] buyYt returns and emits sharesPaid = 0 while sending a share surplus to to 10
[I-06] The hook's non-decaying fee floor is charged on PT notional, so YT sold through PARE's own router loses a large fraction of its remaining value in the final days of every series 11
[I-07] Buy path requires share tokens to accept approve(0) 12
[I-08] buyYt can exhaust the PoolManager's share balance before the exact-input hook fee is taken 13
[I-09] buyYt lacks a minimum YT-output or price-per-YT bound across vault index resets 15

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 9 issues across the reviewed codebase, reported by 0x37, Afriauditor, Hals, naman, Omar, Slayer Security, TheWeb3Mechanic, Valves Security.

SCOPE

https://github.com/pare-audit/contracts/tree/yt-router/src/yield

- YtRouter.sol

REVIEW COMMIT HASH

`2b6c9c34bece9f2e7911fee423877ef1f43bccbb`

FIXES REVIEW COMMIT HASH

`14dad5b5272b1dbbb0e75cbe5e1bbd4c96462f04`

6. Findings

ID	TITLE	SEVERITY	STATUS
I-01	Dust-tick spray makes every YtRouter trade and quote unbounded in gas	Info	Fixed
I-02	vault addresses are fetched twice per trade, and the approval reset after split is a redundant 0->0 write	Info	Fixed
I-03	Front-Runner Can Take The Surplus Of A YT Buyer When PT Trades Above Par	Info	Fixed
I-04	YtRouter constructor does not check that the periphery uses the same PoolManager and factory	Info	Fixed
I-05	buyYt returns and emits sharesPaid = 0 while sending a share surplus to to	Info	Fixed

I-06	The hook's non-decaying fee floor is charged on PT notional, so YT sold through PARE's own router loses a large fraction of its remaining value in the final days of every series	Info	Fixed
I-07	Buy path requires share tokens to accept approve(0)	Info	Fixed
I-08	buyYt can exhaust the PoolManager's share balance before the exact-input hook fee is taken	Info	Acknowledged
I-09	buyYt lacks a minimum YT-output or price-per-YT bound across vault index resets	Info	Fixed

Info Severity

[I-01] Dust-tick spray makes every YtRouter trade and quote unbounded in gas

Info Fixed

SUMMARY

Uniswap v4 charges a swap for every initialized tick it crosses, regardless of how much liquidity sits on that tick. When a swap passes an initialized tick (any tick where a liquidity position begins or ends), `crossTick` writes tick state to storage (fee-growth and liquidity-net updates plus bitmap bookkeeping), and that cost is paid per crossed tick whether the tick holds 1e19 of liquidity or 1 wei.

`YtRouter` runs every trade and every quote as a single swap whose price limit is parked at the far end of the range, so the swap is free to sweep as many ticks as the fill needs. The swap params come from one helper:

```
sqrtPriceLimitX96: zeroForOne ? TickMath.MIN_SQRT_PRICE + 1 : TickMath.MAX_SQRT_PRICE - 1
```

Correctness is instead enforced afterwards by the exact-fill checks (`_in` / `_out`) and the caller's `maxSharesPaid` / `minSharesOut` . Those checks bound the value of the trade, but nothing bounds the tick span the swap sweeps, and therefore nothing bounds its gas.

This applies across all router entry points: buys, sells, and quotes all route through the same swap construction. Any liquidity provider can permissionlessly initialize dust ticks (one 1-wei position per tick), and every subsequent router trade or quote that sweeps across them pays roughly 30,500 gas per tick.

IMPACT

An attacker can arbitrarily inflate the gas cost of router transactions by renting out initialized ticks for pennies. Legitimate buys, sells, and quotes that sweep across the sprayed ticks become far more expensive, up to the point where transactions fail or callers overpay heavily in gas fees, effectively a denial of service on the router.

EXPLOIT PATH

1. The attacker opens many 1-wei liquidity positions, each initializing a distinct tick across the pool's range.
2. A user calls a router buy, sell, or quote that executes a swap with its price limit at the far end of the range.
3. The swap crosses every sprayed initialized tick, paying ~30,500 gas per tick.
4. The user's transaction either reverts out of gas or completes at a greatly inflated gas cost.

RECOMMENDATION

Bound the tick span a router swap can cross instead of relying solely on value-based fill checks. Options include setting a tighter `sqrtPriceLimitX96` derived from the expected execution price rather than the extreme range end, or splitting fills and capping the number of ticks crossed per swap so a dust-tick spray cannot make router operations unbounded in gas.

RESOLUTION

Pare

Fixed in 91a3811 (#8).

buyYt and sellYt now take a sqrtPriceLimitX96 from the caller, and the swap stops there instead of at the far end of the range. A fill that comes up short is still refused whole (PoolTooShallow). quoteBuyYt and quoteSellYt now also return sqrtPriceAfterX96, the price the swap leaves the pool at, so the caller sets the limit from the quote plus their slippage.

Quotes keep the far-end limit: they only run through eth_call, cost nothing, and are how the end of the fill is found. Dust ticks inside a fill's own price range still cost gas to cross, as they do for any swap in the pool through any router; the limit bounds that range to what the caller accepted.

Tests (test/YtRouter.t.sol): with 300 dust ticks sprayed past the limit, an oversized sale costs 207k gas, against 229k with no dust (before the fix: 9.3M and 2.0M). New tests cover a buy and a sale stopping at the limit, a limit on the wrong side or zero being refused, and the quote's end price matching where the trade leaves the pool.

Added in commit fc48a35: quoteBuyYt and quoteSellYt now carry a NatSpec warning that they are meant for off-chain use. Their swap runs to the far end of the range, so a contract calling them on-chain pays for every initialized tick crossed, dust included, and should trade with buyYt / sellYt and a price limit instead. The note on _params now points to that warning rather than saying quotes cost nothing. Comments only, no logic change.

[I-02] vault addresses are fetched twice per trade, and the approval reset after split is a redundant 0->0 write

Info Fixed

SUMMARY

`_check` fetches the vault's `pt` and `share` through external calls before unlock. The callback (`_buy`, `_sell`, `_quoteSell`) then repeats the same external calls for `share`, `pt`, and `yt`, even though all of these values are immutable on the vault. Every trade pays for these lookups twice.

In `_buy`, the router approves exactly `o.amount` to the vault and `split` pulls exactly that amount, so the allowance is already 0 afterwards. The following line therefore does nothing useful:

```
share.forceApprove(o.vault, 0)
```

This performs an external call, a 0 to 0 storage write, and emits an `Approval` event for no effect.

IMPACT

A few thousand gas is wasted on every buy and sell. There is no effect on correctness.

RECOMMENDATION

Read `pt`, `share`, and `yt` once in the entry function and pass them to the callback inside `Order`.

Drop the approval reset after `split`. Removing that line also resolves the known M-15 issue with tokens that reject `approve(0)`.

RESOLUTION

Pare

Fixed in f847bb1.

The approval reset after split is removed: split pulls exactly the approved amount, so the allowance is already 0. _buy now reads the PT once instead of twice. Every router test now also asserts the router leaves no allowance to the vault.

Measured with each call in isolation: a buy costs about 3,900 gas less; a sale is unchanged.

On passing share, pt and yt through the Order: we built and measured it. It gives the same buy saving and 31 gas on a sale, because the three addresses then travel encoded through unlock and back, which costs about what the repeated reads did. We kept the smaller change.

[I-03] Front-Runner Can Take The Surplus Of A YT Buyer When PT Trades Above Par

Info Fixed

SUMMARY

`YtRouter.buyYt()` takes shares from the pool, splits them into PT and YT, and sells the PT back. When the PT sale fetches more than the split cost, `_buy()` sends the surplus to `to`. This is intended behaviour, documented in the NatSpec and exercised by `test_buyWithPtAbovePar`.

The only slippage control is `maxSharesPaid`. In `_buy()`, `sharesPaid` is clamped to `0` whenever `net >= 0`, so the check `sharesPaid > o.limit` passes for every surplus from zero upward, even with `maxSharesPaid = 0`.

```
int256 net = poolManager.currencyDelta(address(this), shareC);
uint256 sharesPaid = net < 0 ? uint256(-net) : 0;
if (sharesPaid > o.limit) revert TooExpensive(sharesPaid, o.limit);
...
} else if (net > 0) {
    poolManager.take(shareC, o.to, uint256(net));
}
```

The buyer has no way to bound the surplus: `quoteBuyYt()` reverts with the same clamped value, and `buyYt()` and the `YtBought` event report only `sharesPaid`. By contrast, `_sell()` has no such gap because `sharesOut < o.limit` bounds its signed result.

IMPACT

A front-runner who sells PT before the buy and buys it back after captures the buyer's surplus.

With a limit of `L`, the buyer's result can move from `+surplus` to `-L`, so the real loss bound is `surplus + L`, not `L`.

In the PoC the buyer loses 245 of 268 shares of surplus on a 10,000-share split (2.4% of the amount split), and the attacker nets 221 shares after fees. The loss is confined to a single trade and does not touch vault funds.

RECOMMENDATION

Let the buyer bound the signed net result of the buy so the limit covers the surplus branch, the way `minSharesOut` does for `sellYt()`. Also return the surplus from `quoteBuyYt()`, `buyYt()` and `YtBought` so the caller can read the value they want to protect.

```
     struct Order {
         Action action;
         address vault;
         PoolKey key;
         uint256 amount; // buy: shares to split; sell: YT to sell
         uint256 limit; // buy: most shares the buyer pays; sell: least shares the seller
receives
+        uint256 minSurplus; // buy: least shares the buyer receives back when the PT sale
covers the split
         address payer;
         address to;
     }
```

```
         int256 net = poolManager.currencyDelta(address(this), shareC);
         uint256 sharesPaid = net < 0 ? uint256(-net) : 0;
+        uint256 surplus = net > 0 ? uint256(net) : 0;
         if (o.action == Action.QuoteBuy) revert QuoteResult(minted, sharesPaid);
         if (sharesPaid > o.limit) revert TooExpensive(sharesPaid, o.limit);
+        if (surplus < o.minSurplus) revert TooLittle(surplus, o.minSurplus);
```

RESOLUTION

Pare

Fixed in 01cd8d9, as recommended.

buyYt now takes minSurplus and reverts with TooLittle(surplus, minSurplus) when the PT sale returns less than that, so the buyer bounds the surplus branch the way minSharesOut bounds a sale. quoteBuyYt, buyYt and YtBought now all report surplus alongside sharesPaid (one of the two is always 0), so the caller can read the value to protect. The split and PT sale in _buy moved into their own block to stay within the stack limit; that code is unchanged.

New test test_buySurplusBelowTheCallersBoundIsRefused: a front-runner's buy shrinks the surplus the buyer quoted while still costing them nothing (so maxSharesPaid = 0 alone would let it through), and the buyer's trade is refused. test_buyWithPtAbovePar now checks that the quote, the trade and the event report the same surplus.

[I-04] YtRouter constructor does not check that the periphery uses the same PoolManager and factory

Info Fixed

SUMMARY

`YtRouter` stores three immutable addresses (`poolManager`, `factory`, `periphery`) and its constructor only rejects zero addresses, never checking they come from the same deployment:

```
constructor(IPoolManager poolManager_, IYtFactory factory_, IYtPeriphery periphery_) {
    if (address(poolManager_) == address(0) || address(factory_) == address(0) ||
address(periphery_) == address(0)) revert ZeroAddress();
    poolManager = poolManager_;
    factory = factory_;
    periphery = periphery_;
}
```

`YieldPeriphery` carries its own immutable `factory` and `poolManager` (lines 19-20). It deploys every `PtPoolHook` against its own `poolManager` and records hooks only for vaults of its own `factory`, yet the router validates pools against both its own `factory` and the periphery's view:

```
if (!factory.isVault(vault)) revert UnknownVault();
address hook = periphery.hookOf(vault);
if (!pair || hook == address(0) || address(key.hooks) != hook) revert NotThePool();
```

Two mismatch cases break trading. With a different factory, `factory.isVault(vault)` passes but `periphery.hookOf(vault)` returns `address(0)`, so `buyYt`, `sellYt`, `quoteBuyYt` and `quoteSellYt` all revert with `NotThePool`. With a different PoolManager, each hook stores the periphery's `poolManager` and rejects any other via `PtPoolHook.onlyPoolManager`, so pools cannot be initialized or swapped on the router's `poolManager` and every trade reverts.

IMPACT

A router deployed with inconsistent dependencies can never complete a trade. The YT market is unreachable through that router until it is redeployed with matching addresses.

RECOMMENDATION

Expose the periphery's `factory` and `poolManager` on `IYtPeriphery` and require them to equal the router's own `factory` and `poolManager` in the constructor, reverting otherwise.

RESOLUTION

Pare

Fixed in 0b31619, as recommended.

IYtPeriphery now exposes factory() and poolManager() (the public immutables YieldPeriphery already has), and the constructor reverts with NotOneDeployment() unless both equal the router's own factory and poolManager. The zero-address check still runs first.

New test test_constructorRefusesAPeripheryOfAnotherDeployment: a periphery built for another factory and a router naming another PoolManager are both refused; the matching set deploys.

[I-05] buyYt returns and emits sharesPaid = 0 while sending a share surplus to to

Info Fixed

SUMMARY

`_buy` collapses the signed net amount into a single unsigned value, so the reported shares paid is zeroed out whenever the trade produces a surplus:

```
sharesPaid = net < 0 ? -net : 0;
```

When `net > 0`, the surplus is still transferred out via `take(shareC, o.to, net)`, but that amount appears in neither the function's return data nor the `YtBought` event.

IMPACT

A contract that calls `buyYt` with itself as `to` cannot learn the surplus from the call. It records a zero-cost buy while the extra shares sit untracked.

Off-chain indexers relying on the return value or event record an incorrect cost basis.

Note that M-29 only addresses the quote's revert payload, and its fix (a new quote error) leaves the live return value and the emitted event unchanged.

RECOMMENDATION

Return and emit the surplus as well, or expose the signed net directly so callers and indexers can account for the surplus shares.

RESOLUTION

Pare

Fixed in 01cd8d9, together with I-03.

buyYt now returns (ytOut, sharesPaid, surplus) and YtBought emits surplus alongside sharesPaid; exactly one of the two is non-zero. quoteBuyYt returns it too, so a contract buying for itself and an indexer reading the event both see the surplus sent to to. test_buyWithPtAbovePar checks that the return value, the event and the quote report the same surplus.

[I-06] The hook's non-decaying fee floor is charged on PT notional, so YT sold through PARE's own router loses a large fraction of its remaining value in the final days of every series

Info Fixed

SUMMARY

The router is the venue PARE provides for trading YT, and it prices every YT trade through the PT pool: selling `n` YT buys `n` PT and merges, buying `n` YT splits and sells `n` PT. Each trade therefore pays the pool's fees on the full PT notional, while the YT it delivers is worth only the yield remaining to maturity.

`PtPoolHook` scales its fee down as time to maturity shrinks, but floors it at `minFeePips` (100 pips, 0.01% per swap, in the sponsor's fixture). The fee stops decaying while the YT's value keeps falling toward zero.

At 10% APY, a YT one day from maturity is worth about 0.027% of notional, and the floor alone takes 0.01% before LP spread and price impact. Measured in the fixture one day out, `quoteSellYt` returns 0 at a PT discount of 0.02% or less, and executed sales deliver only 40 to 80% of the YT's fair value at discounts of 0.05 to 0.14% (about 0.016 to 0.08% of notional per sale).

`quoteSellYt` reports these as ordinary prices. Nothing in the quote, the router, or the docs tells a seller, or an integrator using the quote as fair value, that holding to maturity and calling `claimInterest` pays more. Pendle avoids this by letting its fee decay to zero at expiry precisely because YT trades through PT; PARE's floor stops that decay.

IMPACT

Each affected sale loses 20 to 60% of the asset's remaining value, demonstrated against the real PoolManager artifact with the sponsor's own parameters. The amount is small in notional terms, but it is the entire economic content of the asset the router exists to trade.

The loss is systematic, not situational. It is set by a protocol parameter and the calendar, not by market conditions or an attacker, and it recurs in the last days of every series for every YT sold through the protocol's venue.

Integrators cannot see it. Strategy vaults, aggregators, and liquidation paths that treat `quoteSellYt` as fair value execute at it, and the quote offers no reference against which the haircut is visible.

No principal is at risk: a holder who declines the price can hold to maturity and claim. `minFeePips` is owner-set.

RECOMMENDATION

Let the hook's fee decay to zero at expiry as Pendle does, or size the floor against the YT's remaining value rather than the full PT notional. Either approach removes the haircut and carries no security trade-off.

RESOLUTION

Pare

Fixed in 9cb7348, taking the first recommendation.

PtPoolHook now accepts minFeePips = 0 together with a treasury share, so the fee follows feePerYear × timeLeft all the way to zero at maturity, as Pendle's does. A non-zero floor must still yield at least one treasury pip (the L-05 check is unchanged for it). In the last hours before

maturity, a total fee of 1 pip pays the LPs alone, but that window is a few hours, not the life of the floor that L-05 addressed. afterSwap already returns early when the treasury's part is 0. New series deploy with a floor of 0.

The live pUSDx-MAR27 hook was deployed with minFeePips = 100 and is immutable. With its 2,000 pips a year, that floor binds only in the series' last ~18 days.

New test test_aZeroFloorDecaysToNothingAtMaturity: a zero floor with a 50% treasury share deploys, charges 5 pips one day out (split 3/2) and nothing at maturity; a 100-pip floor still holds.

On each point, in 14dad5b (comments only):

Rounding: agreed and intended. With a floor of 0 the fee reaching 0 slightly early is consistent with decaying to zero. totalFeePips now documents it: 0 from 1 year / feePerYearPips before maturity, about 4.4 hours at the 2,000 pips we deploy with, longer with a smaller feePerYearPips. We have left feePerYearPips unbounded: a smaller value only brings that zero-fee window forward, and maxFeePips caps the other end.
quoteSellYt now says that near maturity a sale still buys the full PT amount through the pool, so fees and price impact can take most of a YT's small remaining value, and that holding to maturity and calling claimInterest may pay more.
MAR27: yes, accepted as a residual. The hook is immutable, the floor binds only in the series' last ~18 days, and no principal is at risk: a holder who declines the price holds to maturity and claims. Every series after MAR27 deploys with a floor of 0.

[I-07] Buy path requires share tokens to accept approve(0)

Info Fixed

LOCATION

`YtRouter:_buy:194-196`

SUMMARY

After each buy, the router clears its allowance by calling `forceApprove(o.vault, 0)`, even though the vault's `split` consumes the exact approved amount via `transferFrom` and leaves the allowance at zero already.

`forceApprove` handles the USDT-style requirement of clearing an allowance before setting a new nonzero one, but it cannot work around a token that rejects zero approvals outright.

```
share.forceApprove(o.vault, o.amount);
uint256 minted = v.split(o.amount, address(this));
forceApprove(o.vault, 0);
```

For a share token that reverts on `approve(vault, 0)`, this cleanup call reverts and rolls back the whole buy.

IMPACT

Every buy through a vault whose share token rejects zero-value approvals reverts, making those vaults unusable for buyers.

This is an availability problem only. Because the outer transaction rolls back atomically, no partial state is left behind.

EXPLOIT PATH

1. The router calls `forceApprove(vault, amount)`, which succeeds for the affected share token.
2. `YieldVault.split` pulls the approved amount and mints PT and YT, consuming the full allowance.
3. `_buy` calls `forceApprove(vault, 0)`, which the token rejects.
4. The unlock reverts atomically, so no buyer can complete a buy through this vault.

PREREQUISITES

A registered vault must use an ERC-20-like share token that accepts the initial nonzero approval but rejects `approve(vault, 0)`.

RECOMMENDATION

Drop the cleanup approval, since the exact allowance is already consumed by `split`:

```
share.forceApprove(o.vault, o.amount);
uint256 minted = v.split(o.amount, address(this));
// Do not call forceApprove(o.vault, 0) here.
```

If a cleanup policy is needed for tokens that may not consume the full allowance, use a token allowlist or a compatibility-specific adapter rather than unconditionally calling `approve(0)` on arbitrary registered shares.

RESOLUTION

Pare

Fixed in f847bb1, together with I-02.

The cleanup forceApprove(o.vault, 0) after split is removed, as recommended: split pulls exactly the approved amount, so the allowance is already 0, and a share token that rejects approve(0) no longer blocks buys. The only approval left is forceApprove(o.vault, o.amount) before the split, made from a zero allowance every time. Every router test now asserts the router leaves no allowance to the vault (_empty()).

[I-08] buyYt can exhaust the PoolManager's share balance before the exact-input hook fee is taken

Info Acknowledged

LOCATION

`YtRouter:_buy:189-201`

SUMMARY

A YT buy checks only the gross `sharesToSplit` reserve and then immediately removes that entire balance from the PoolManager, leaving nothing behind for downstream operations.

During the subsequent exact-input PT sale, `PtPoolHook.afterSwap` charges its protocol fee on the unspecified output currency, which is the share currency on this path. The hook needs share liquidity available before the router settles its own net share payment, but the initial `take` has already drained it.

```
if (share.balanceOf(address(poolManager)) < o.amount) {
    revert PoolTooShallow();
}
```

For example, if the PoolManager holds exactly 100 shares and a buyer calls `buyYt` with `sharesToSplit = 100`, the reserve check passes, `take` transfers all 100 shares out, and the hook's later share-denominated fee transfer has no balance to draw from, reverting the whole buy.

IMPACT

Otherwise solvent YT buys can revert when the PoolManager's share inventory is exactly enough for the split reserve but short of the hook's immediate share-denominated fee. The buyer never exceeds the configured price limit, yet the trade fails.

The same boundary condition also affects exact-input buys and quotes sitting near the manager's available share balance. The failure is deterministic whenever the configured hook fee is nonzero and the initial withdrawal consumes the manager's physical share reserve, creating a temporary availability and composability failure for affected trades. It follows the normal configured `PtPoolHook` path and requires no malicious hook or malformed pool key.

EXPLOIT PATH

1. The PoolManager holds 100 shares and the PT pool uses a nonzero-fee `PtPoolHook`.
2. A buyer calls `buyYt` with `sharesToSplit = 100`; the reserve check passes.
3. `poolManager.take` transfers all 100 shares to the router for splitting.
4. The router sells the minted PT via an exact-input swap and the hook attempts to take its share-denominated protocol fee.
5. No share balance remains to satisfy the fee, so the swap and the entire buy revert despite the buyer being able to fund the net cost.

PREREQUISITES

The vault's configured PT pool must use a `PtPoolHook` with a nonzero protocol share, and the PoolManager's physical share balance after the initial `take` must be below the hook fee.

RECOMMENDATION

Include an estimated or configured fee reserve in the precondition and leave that reserve in the manager:

```
uint256 feeReserve = _maxHookShareFee(o.key, o.amount);
if (share.balanceOf(address(poolManager)) < o.amount + feeReserve) {
    revert PoolTooShallow();
}
```

Preferably, refactor fee settlement so the hook's fee is accounted as a delta and settled after the router's buyer payment rather than requiring physical share inventory during the callback.

RESOLUTION

> **Pare**
>
> Acknowledged.
>
> The failure needs a buy that takes from the PoolManager nearly all of its share balance, leaving less than the hook's protocol fee (at most 0.5% of the sale's output with the live parameters). On mainnet the PoolManager's sUSDx balance includes every v4 pool that holds sUSDx. A buy that size also sells its PT into a pool left with almost no share depth, and quoteBuyYt runs the same path and reverts the same way, so our app never offers such a trade. When it does occur, the whole buy reverts atomically and nothing moves.
>
> A fee reserve in the precondition would refuse these buys slightly earlier with a clearer error, and would also refuse some that succeed today. Booking the exact-input fee as a claim, as the hook already does for exact-output, would remove the dependency on physical inventory. But it changes how the treasury receives fees (a collect() step), and the live series' hook is immutable, so it would not cover that pool. We prefer to keep the current behaviour.

[I-09] buyYt lacks a minimum YT-output or price-per-YT bound across vault index resets

Info Fixed

LOCATION

`YtRouter:_buy:193-217`

SUMMARY

A buy should let the caller bound both the maximum shares paid and the minimum YT received. The router only accepts `maxSharesPaid`, so the YT output can change materially between a user's quote and execution.

`YieldVault.split` intentionally resets an emptied series to a lower rate when the underlying share rate has fallen. `YtRouter._buy` accepts the resulting `minted` amount and checks only whether the share payment is within `o.limit`; it never checks that the buyer received a minimum YT amount.

Because minted PT and YT depend on the execution-time vault index, a buyer can pay an allowed share amount while receiving substantially fewer YT after a downward index reset. For example, a buyer quotes against an index of `2e18` expecting 20 YT for 10 shares, but if the series empties and the live rate drops to `1e18`, `split` mints only 10 YT for the same 10 shares.

IMPACT

A buyer can receive fewer YT than anticipated after an emptied series resets its index, producing a materially worse effective price per YT. `buyYt` does not revert as long as the share payment stays within the supplied limit, so there is no protection against a stale cross-transaction quote being silently filled at a worse rate.

EXPLOIT PATH

1. A buyer observes a vault index of `2e18` and submits a buy for 10 shares with a payment limit based on an expected 20 YT output.
2. Before execution, all outstanding PT and YT are merged or redeemed, leaving both supplies at zero.
3. The underlying share rate is now `1e18`, so `YieldVault.split` resets the effective index and mints only 10 PT and 10 YT from the same 10 shares.
4. The PT sale costs no more than the buyer's share limit, so `buyYt` succeeds and transfers only 10 YT with no minimum-output protection.

PREREQUISITES

The vault must have zero PT and YT supply, and its current underlying rate must be below the stored index so `YieldVault.split` performs its documented reset. No privileged attacker is required; this is a reachable empty-series state.

RECOMMENDATION

Add and enforce a minimum YT amount:

```
function buyYt(
    address vault, PoolKey calldata key, uint256 sharesToSplit,
    uint256 maxSharesPaid, uint256 minYtOut, address to, uint256 deadline
) external nonReentrant returns (uint256 ytOut, uint256 sharesPaid) {
    // ... execute _buy
    if (ytOut < minYtOut) revert TooLittle(ytOut, minYtOut);
}
```

The check may also be placed inside `_buy` immediately after `split`, before any user-facing transfer. A separate protected entry point could be added to preserve backward compatibility.

RESOLUTION

Pare

Fixed in dc6969c, as recommended.

buyYt now takes minYtOut and reverts with TooLittle(ytOut, minYtOut) when the split mints fewer YT, after the unlock returns, so the whole buy is undone. Callers take it from quoteBuyYt's ytOut. Together with maxSharesPaid, minSurplus and the price limit, a buy is now bounded on every side.

New test test_buyRefusedBelowTheCallersMinimumYt: one YT more than the quote is refused and the buyer's balances are untouched; the quote's YT as the bound fills.