yAudit

July 2026

Prepared for
Promis

Audited by
Panda
fedebianu

Promis Smart Contracts

Smart Contract Security Assessment

Contents

1 Review Summary

1.1 Protocol Overview

Promis is an upgradeable yield-bearing token protocol centered on proUSD, whose USD-denominated value is designed to appreciate as deposited yAssets earn yield through integrations such as Aave V3 and Morpho. Minting and redemption use off-chain-authorized request and finalization flows, with queued redemptions settled in batches. The protocol also includes proUSD+, a tiered locking product backed by StrategyVault, and a Chainlink adaptor that normalizes external prices to 18 decimals.

1.2 Audit Scope

This audit covers 11 implementation contracts and 8 supporting state files totaling approximately 3,921 lines of code across 10 days of review. The scope comprises the listed contracts from the core, Chainlink oracle adaptor, ProTokenPlus vault, and yield-handler modules.

```
contracts/core/ProToken.sol
contracts/core/ProTokenOperations.sol
contracts/core/ProTokenSettings.sol
contracts/core/ProTokenUnmintHandler.sol
contracts/core/YAssetOperationsHandler.sol
contracts/core/state/ProTokenOperationsState.sol
contracts/core/state/ProTokenSettingsState.sol
contracts/core/state/ProTokenUnmintHandlerState.sol
contracts/core/state/YAssetOperationsHandlerState.sol
contracts/oracles/OracleChainlinkPushAdaptor.sol
contracts/vaults/ProTokenPlus/ProTokenPlus.sol
contracts/vaults/ProTokenPlus/ProTokenPlusOperations.sol
contracts/vaults/ProTokenPlus/StrategyVault.sol
contracts/vaults/ProTokenPlus/state/ProTokenPlusState.sol
contracts/vaults/ProTokenPlus/state/StrategyVaultState.sol
contracts/yields/AaveV3YieldHandler.sol
contracts/yields/MorphoYieldHandler.sol
contracts/yields/state/AaveV3YieldHandlerState.sol
contracts/yields/state/MorphoYieldHandlerState.sol
```

1.3 Risk Assessment Framework

1.3.1 Severity Classification

Severity	Description	Potential Impact
Critical	Immediate threat to user funds or protocol integrity	Direct loss of funds, protocol compromise
High	Significant security risk requiring urgent attention	Potential fund loss, major functionality disruption
Medium	Important issue that should be addressed	Limited fund risk, functionality concerns
Low	Minor issue with minimal impact	Best practice violations, minor inefficiencies
Undetermined	Findings whose impact could not be fully assessed within the time constraints of the engagement. These issues may range from low to critical severity, and although their exact consequences remain uncertain, they present a sufficient potential risk to warrant attention and remediation.	Varies based on actual severity
Gas	Findings that can improve the gas efficiency of the contracts.	Increased transaction costs
Informational	Code quality and best practice recommendations	Reduced maintainability and readability

Table 1: severity classification

1.4 Key Findings

Breakdown of Finding Impacts

Impact Level	Count
Critical	0
High	5
Medium	5
Low	14
Informational	9

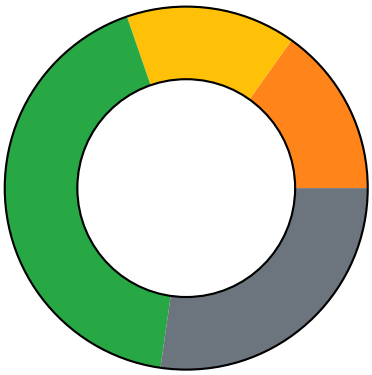

Figure 1: Distribution of security findings by impact level

1.5 Overall Assessment

The review identified four high-severity pricing and accounting issues affecting proUSD minting, redemption value, reserve solvency, and StrategyVault withdrawals. It also identified a medium-severity oracle-deviation configuration issue, five low-severity configuration and upgrade risks, eight informational observations, and two gas optimizations. The modular architecture and use of established libraries are positive, but the high- and medium-severity findings should be remediated and thoroughly retested before deployment; all developer responses in the reviewed report remain pending.

2 Audit Overview

2.1 Project Information

Protocol Name: Promis
Repository: https://github.com/promis-finance/promis-contracts
Commit Hash: 4cd2d43122bf10c173b16d9a017415f35839c003
Final Commit Hash: 63ca73515f5c9a8f6ef1ea74b9261c534fe53efa
Commit URL: https://github.com/promis-finance/promis-contracts/tree/4cd2d43122bf10c173b16d9a017415f35839c003

2.2 Audit Timeline

The audit was conducted from July 6 to 17, 2026.

2.3 Audit Resources

- Promis contracts

Category	Mark	Description
Access Control	Average	Privileged entry points generally enforce the intended admin, operator, authority, strategist, or protocol-contract checks, and no direct access-control bypass was identified. However, the custom admin pattern is duplicated throughout the codebase, and several roles can directly move protocol backing or alter configuration in ways that can disrupt pending requests and accounting.
Mathematics	Low	Core pricing and accounting require substantial remediation. Four high-severity findings show that oracle prices, usdCap handling, nominal 1:1 redemption caps, and the StrategyVault high-water accounting model can overmint proUSD, overpay or underpay redemptions, and leave withdrawal reserves underfunded.
Complexity	Average	The contracts are separated into logic, state, interfaces, and types, which aids review and upgrade safety. Nonetheless, asynchronous request and batch lifecycles, shared reserve accounting, live global configuration, and delegatecalls from ProTokenPlus to a version-coupled operations handler create significant cross-contract state complexity.
Libraries	Good	The codebase makes appropriate use of established OpenZeppelin upgradeable contracts, SafeERC20 utilities, and official Aave and Morpho interfaces. The review did not identify a vulnerability originating in an external library.
Decentralization	Low	The system relies heavily on trusted off-chain and privileged actors. The admin, operator, and externalBusiness roles can withdraw yAsset backing, the strategist can route StrategyVault backing to arbitrary destinations without on-chain repayment enforcement, and request finalization depends on an off-chain authority.
Documentation	Average	The repository provides a detailed architectural README, extensive interfaces and NatSpec, and a ProTokenPlus upgrade guide. Documentation quality is reduced by several behavior mismatches, including the meaning of zero oracle deviation, optional aToken configuration, unused oracle payloads, and an upgrade guide that does not reflect the executable version-mismatch window.
Testing and verification	Average	The repository contains broad Hardhat tests across core, oracle, vault, yield, and integration flows. However, several high-impact price-floor, price-decrease, redemption-cap, reserve-accounting, configuration-migration, and version-mismatch cases were not covered until focused audit proof-of-concept tests were added, indicating a need for stronger edge-case and invariant testing.

Table 2: Code Evaluation Matrix

2.4 Critical Findings

None.

2.5 High Findings

2.5.1 Missing oracle price floor lets capped deposits overmint proUSD

Technical Details

`ProToken.setUSDPrice()` enforces a minimum non-zero proUSD price of `1e18`:

```
if (_price < MIN_USD_PRICE && _price != 0) revert InvalidPrice();
```

However, `ProTokenOperations._getUsdRepresentationProToken()` skips this invariant when `ProTokenSettings.proTokenPriceSettings.oraclePriceSource` is configured. In oracle mode, it reads the proUSD oracle and uses the returned price directly:

```
usdRepresentation.assetUSD = _aggregateOraclePrices(
    _proToken,
    oracles
);

usdRepresentation.assetAmountUSD =
    (_amount * usdRepresentation.assetUSD) /
    (10 ** _decimals);
```

During mint finalization, `_calculateMintingAmount()` divides the capped yAsset USD value by `proTokenUsdRep.assetAmountUSD`. If the proUSD oracle reports a price below 1 USD, the denominator is reduced and the user receives more proUSD than the deposited collateral should mint.
Example: with a capped stable yAsset at `usdCap == 1e18`, a 100 USDC deposit finalized while the proUSD oracle reports `$0.80` mints `125 proUSD`. Once the oracle returns to `$1`, the attacker can redeem the excess proUSD through the normal unmint flow and withdraw more backing than they deposited.

Impact

The impact is high if oracle-mode proUSD pricing is enabled. A temporary or manipulated sub-1 USD proUSD oracle price lets users overmint against capped deposits, undercollateralizing the shared yAsset reserves.

Recommendation

Apply the same non-zero `>= 1e18` floor to oracle-sourced proUSD prices before mint and unmint math. If oracle-mode proUSD pricing is required, also consider supporting multiple proUSD oracle sources and deviation checks instead of relying on a single oracle address.

Developer Response

Fixed in commit f9d1a56: - Added a price floor

2.5.2 Capped yAsset redemptions can overpay when the live price exceeds `usdCap`

Technical Details

`ProTokenOperations._calculateUnmintAmount()` applies `usdCap` to unmint/redemption pricing by using the lower of the live yAsset price and the configured cap:

```
uint256 cappedYAssetUSD = yAssetUsdRepresentation.assetUSD >
    yAssetSettings.settings.priceSettings.usdCap
    ? yAssetSettings.settings.priceSettings.usdCap
    : yAssetUsdRepresentation.assetUSD;

return
    (proTokenUsdRepresentation.assetAmountUSD * oneYAssetUnit) /
    cappedYAssetUSD;
```

This is unsafe for redemptions. If a yAsset is trading above its cap, the denominator is reduced below the live price, so the contract returns too many yAsset units for the proUSD being burned.
Example: if a capped USDC-like yAsset has a live price of `$1.03` and `usdCap == $1.00`, burning `100 proUSD` returns `100` yAsset units. At the live price, those units are worth `$103`; the fair redemption for `$100` of proUSD would be about `97.087` yAsset units.
An attacker can mint proUSD against a fairly priced yAsset, then unmint into the premium capped yAsset and extract the difference. The overpayment happens in both instant redemptions and queued redemptions, because the same inflated amount is either paid immediately by `YAssetOperationsHandler.payOut()` or recorded as a later liability in `ProTokenUnmintHandler`.

Impact

High. When an enabled redemption yAsset trades above its configured `usdCap`, the protocol can transfer more live USD value than the proUSD being burned. This directly drains yAsset reserves or records inflated queued liabilities that must later be funded.

Recommendation

Do not use `usdCap` as a lower redemption denominator. For unminting, price the output yAsset using its live price, or ensure the denominator cannot be lower than the live price. Keep `usdCap` as a mint-side credit limit if the goal is to avoid over-crediting premium assets on deposit.

Developer Response

Fixed in commit 93f8bfa: - The unminting side now uses a floor instead of a cap (max instead of min)

2.5.3 proUSD price decreases can break StrategyVault withdrawal accounting

Technical Details

`StrategyVault._accrueGrowth()` only moves `lastPrice` upward. When `currentPrice > lastPrice`, it subtracts freed proUSD from `depositProUSD` and `withdrawProUSD`, then books the freed amount into `growthProUSD`. When `currentPrice <= lastPrice`, it returns without undoing any previously banked growth. This is unsafe because `ProToken.setUSDPrice()` permits price decreases as long as the new price is still at least `1e18`. `ProTokenPlusOperations.executeCompleteWithdraw()` converts a user's base-denominated unbonding amount into proUSD using the current proUSD price, but `StrategyVault.take()` only allows payment from the already-reduced `withdrawProUSD` reserve. Example:

1. The withdrawal reserve holds `withdrawBase = 200e18` and `withdrawProUSD = 200e18` while proUSD is priced at `$1`.
2. proUSD is set to `$2` and any vault interaction accrues growth. The reserve now only keeps `100e18` in `withdrawProUSD`, while `100e18` is moved into `growthProUSD`.
3. `claimGrowth()` can transfer that `100e18` out of the vault.
4. proUSD is later set back to `$1`, which is allowed by `ProToken`.
5. Completing a `200e18` base withdrawal requires `200e18` proUSD at the current price, but `take()` reverts because `withdrawProUSD` is only `100e18`.

Impact

High.

Recommendation

Enforce the intended non-decreasing price invariant in `ProToken.setUSDPrice()` by reverting any non-zero price update below the previous non-zero price. If `0` must remain supported as a temporary disabled-price state, it should not reset the monotonic floor; the next non-zero price should still be required to be at least the last accepted non-zero price.

Developer Response

Fixed in commit bdb17ad6: - Separated the role for setting the proUSD price into two roles: `onlyAdmin`, with no specific price restrictions, and `onlyPriceOperator`, with only the ability to increase the price subject to step restrictions. - The `onlyAdmin` role is useful for decreasing the price in case of slashing in third-party protocols such as Aave/Morpho - `StrategyVault` now has `markdownPrice()`, which is used in case of slashing in third-party protocols. This function records deficits resulting from slashing - `StrategyVault` now has `cover()`, which is used to cover the deficits after `markdownPrice()` succeeds

2.5.4 1:1 unmint cap under-redeems proUSD when proUSD price is above 1 USD

Technical Details

`ProTokenOperations` prices minting and unminting from the current proUSD USD price, but the unmint path then caps the output by the nominal proUSD amount instead of the USD value being burned.
For minting, a deposit of 100 yAsset at a proUSD price of 2 USD correctly mints 50 proUSD:

```
calculatedAmount = (yAssetUsdRep.assetAmountUSD * USD_PRECISION) / proTokenUsdRep.
assetAmountUSD;
```

For unminting, those 50 proUSD are still worth 100 USD, so the user should receive 100 yAsset at a 1 USD yAsset price. `_calculateUnmintAmount` initially calculates that amount, but then replaces it with the nominal 50 proUSD amount:

```
if (calculatedAmount > decimalAdjustedAmount) {
    return decimalAdjustedAmount; // Cap at 1:1
}
```

This makes proUSD appreciation unrecoverable through the normal unmint path for yAssets configured with `usdCap == 0`, which is also the default test/helper configuration.
A temporary PoC confirmed the behavior: after setting the proUSD price to 2 USD, minting 100 yAsset produced 50 proUSD, and unminting those 50 proUSD returned only 50 yAsset.

Impact

High. proUSD is intended to accrue USD value over time, but the default unmint path prevents holders from redeeming that increased value. Users either accept under-redemption or set a fair `minAmountOut` and cannot unmint. New minters after price appreciation can also lose principal on a mint/unmint round trip.

Recommendation

Do not cap unmint output by the nominal proUSD token amount. Calculate yAsset redemption from the USD value of the proUSD being burned, applying an explicit yAsset price cap only to the yAsset price if depeg protection is required. Alternatively, require a non-zero `usdCap` for redeemable stable yAssets and remove the `usdCap == 0` 1:1 amount cap path.

Developer Response

Fixed in commit d360232: - Removed the `usdCap == 0` else branch

2.5.5 `StrategyVault.claimYield()` leaves `withdrawProUSD` overstated, breaking vault solvency

`StrategyVault.claimYield()` withdraws protocol rewards held inside `withdrawProUSD` but transfers the proUSD out **without decrementing `withdrawProUSD`**. The ledger then overstates the reserve, so even a correctly-sized claim breaks the solvency invariant and bricks user withdrawals.

Technical Details

Per the design confirmed by the team, when the strategist calls `repay()`, the repaid amount bundles three things: user principal, user rewards, and protocol rewards, all booked into the withdraw reserve:

```
withdrawProUSD += proUSDMinted;   // user principal + user rewards + protocol rewards
withdrawBase   += usdAmount;
```

`claimYield()` is meant to let the admin or operator withdraw only the protocol-reward portion, whose size is tracked by an off-chain dashboard and passed as `amount`. The bug is that it transfers `claimed` proUSD out but never reduces `withdrawProUSD`:

```
uint256 obligations = depositProUSD + growthProUSD;       // withdrawProUSD excluded (holds the claimable protocol rewards)
uint256 available = held > obligations ? held - obligations : 0;
if (amount > available) revert ExceedsClaimableYield(amount, available);
claimed = amount;
IERC20(proToken).safeTransfer(recipient, claimed);        // no `withdrawProUSD -= claimed`
```

The tokens leave the vault but the ledger still counts them. The solvency invariant enforced by `take()` (`held >= depositProUSD + growthProUSD + withdrawProUSD`) is now violated because `held` dropped by `claimed` while `withdrawProUSD` did not. The next user withdrawal reverts `InsufficientVaultBalance`.
This happens even when the operator claims exactly the correct protocol-reward amount reported by the dashboard: the claim is sized right, but the reserve ledger is left overstated relative to the tokens actually held.

Impact

High. The issue manifests in normal operation with an honest operator. After any successful `claimYield()` call, `withdrawProUSD` overstates the proUSD actually held, so `held < depositProUSD + growthProUSD + withdrawProUSD` and both `take()` and `borrow()` revert on their solvency checks. All user withdrawals and strategist borrows are blocked. Recovery requires a proUSD transfer into the vault equal to the claimed amount. A single claim bricks the vault; repeated claims compound the deficit.

Recommendation

Decrement the reserve ledger when protocol rewards are claimed, so the tokens and the ledger stay in sync and `take()`'s solvency invariant holds:

```
withdrawProUSD -= claimed;
withdrawBase   -= (claimed * lastPrice) / USD_PRECISION;   // keep the base mirror consistent
IERC20(proToken).safeTransfer(recipient, claimed);
```

Developer Response

Fixed in commit 1953f96: - `claimYield()` now decrements `withdrawProUSD` and `withdrawBase` - `claimYield()` now makes sure the claim does not eat into `earmarkedWithdrawBase` territory

2.6 Medium Findings

2.6.1 Median-relative validation permits oracle spreads above the configured maximum

Technical Details

`OracleAggregationSettings.maxPriceDeviation` is documented as the maximum allowed deviation between oracle prices. However, `_validatePriceDeviation()` does not compare the sources to one another. It compares every source to the aggregated median.
For an even number of sources, `_calculateMedianPrice()` defines the median as the arithmetic mean of the two central values. With exactly two prices `L` and `H`, this gives:

```
M = (L + H) / 2
deviation(L, M) = deviation(H, M) = (H - L) / (H + L)
```

Therefore, a configured threshold `T` accepts any pair satisfying:

```
H / L <= (1 + T) / (1 - T)
```

At the deployed 10% threshold, the two sources can have a price ratio of approximately `1.2222`. This is materially wider than the documented 10% maximum between the oracle responses. With only two sources, one faulty source can also move the returned arithmetic mean while neither response fails validation.
This can be monetized through `_calculateUnmintAmount()`, where an underestimated yAsset price is used as the redemption denominator.

Impact

Medium. A response pair whose direct spread exceeds the configured limit can still pass validation and produce a biased aggregate price. For redeemable assets, an underestimated aggregate price increases the number of yAsset units paid per burned proUSD, allowing repeated extraction from immediate reserves or creation of oversized queued liabilities.
Example scenario:

```
oracle A:             1.00 USD
oracle B:             0.82 USD
maxPriceDeviation:    1000 BPS (10%)
calculated median:    0.91 USD
```

Both deviations from the median round to at most 10%, so `_aggregateOraclePrices()` succeeds even though the sources differ far more than 10% relative to the `$1` response.
For a yAsset with a `$1` `usdCap`, a `$1` proUSD price, and no unmint fee, redeeming 100 proUSD uses the `$0.91` aggregate as the denominator and returns approximately 110 yAsset units. If the `$1` source is correct, the redeemer extracts about `$10` of excess collateral.

Recommendation

Validate a directly defined spread between `minPrice` and `maxPrice` in addition to, or instead of, per-source deviation from the median.

Developer Response

Fixed in commit c5f6fae: - The deviation check now includes the spread

2.6.2 Queued unmint burns proUSD before the yAsset payout is guaranteed

A queued unmint irrevocably burns the user's proUSD upon finalization, but the yAsset payout then depends on a permissioned actor processing and funding the batch. If that never happens, the user loses their proUSD with no on-chain recourse as the asset is destroyed, not merely escrowed.

Technical Details

The unmint has two stages. In stage 1, `createUnmintRequest()` escrows the user's proUSD (still recoverable via a `PROOF_OF_RETURN`). Stage 2 is where this issue lives: when `finalizeUnmintRequest()` is called with `PROOF_OF_APPROVE`, `_executeUnmint()` burns the proUSD before deciding how to pay out (line 407):

```
1  IProToken(proTokenInfo.proToken).burn(address(this), _amount);   // proUSD burned
2
3  if (yOps.previewPayOut(yAssetToReceiveAmount)) {
4      yOps.payOut(_recipient, yAssetToReceiveAmount);              // enough liquidity → instant, fine
5  } else {
6      IProTokenUnmintHandler(...).createUnmintRequest(_recipient, _yAsset, yAssetToReceiveAmount); // queue
7  }
```

In the queued branch (taken when liquidity is insufficient), the proUSD is already burned but the user receives only a claim in `ProTokenUnmintHandler`. To be paid, two steps outside the user's control must happen:

1. `processNextUnmintBatch()` marks the batch `processed` and pulls the yAsset from the caller to fund it.
2. Only then can the user call `claimUnmintRequests()`.

There is no permissionless fallback, timeout, cancellation, or refund. If the privileged actor never processes/funds the batch, the user holds no proUSD (burned) and no yAsset (unfunded).

Impact

Medium. In a sound redemption design, burn and payout are atomic; here they are separated across time and across a trust boundary. Aggravating factors:

- The queued path is taken precisely under liquidity stress (`previewPayOut` false), exactly when the operator's solvency/reliability is most in question.
- `claimUnmintRequests*()` is `whenNotPaused`, so even an already-funded batch cannot be claimed while the protocol is paused.
- `emergencyWithdraw()` can pull the yAsset earmarked for a processed batch, stranding claims.

Recommendation

Do not burn until payout is guaranteed: in the queued path, escrow the proUSD so an unfunded request can be refunded in proUSD.

Developer Response

Acknowledged. It is intended. - A finalized unmint MUST be guaranteed and has to be completed. - An async request has to be handled by an operator. - A finalized async path should never be able to be refunded. - The instant flow was added later

2.6.3 `StrategyVault.regive()` and `StrategyVault.rotate()` can underfund withdrawals

Technical Details

`StrategyVault` serves user exits from a shared reserve tracked by `withdrawProUSD` / `withdrawBase`. The reserve is prefunded by the strategist via `repay()` ~1–2 weeks before positions unlock (per the SDD), sized against expected aggregate exits. Crucially, **nothing earmarks a portion of the reserve to a specific pending exit** when an unbonding is finalized in `_executeInitiateWithdraw()`: it sets `unbondingEnd` (line 314) but touches no vault reserve ledger. The reserve is consumed only later, at `completeWithdraw()` → `take()`, which requires `proUSDAmount <= withdrawProUSD` and otherwise reverts `WithdrawReserveUnderfunded` (lines 228-229).
Between the start of unbonding and `completeWithdraw`, the reserve is unprotected, and two paths can drain it:

1. Permissionless `regive()` via `relock()`. Any user with an unlocked position can call `relock()` (`executeRelock` → `regive(amount)` at line 449). `regive()` only checks whether the reserve covers *this* relock (line 258), not whether the reserve remains for matured unbondings:

```
if (proUSDAmount > withdrawProUSD || worthBase > withdrawBase) {
    emit RegivenAsync(...); return;          // defer
}
withdrawProUSD -= proUSDAmount;              // else drain, no check for pending exits
```

2. Privileged `rotate()`. `rotate()` (admin/operator) performs the same `withdraw → deposit` move bounded only by `withdrawProUSD` (line 370), with no on-chain link to the deferred-regive debt it is meant to settle. It can move the reserve past what pending exits require just as easily.

Neither path drives `withdrawProUSD` negative (both check the aggregate), but both greedily consume reserve without subtracting already-matured exit obligations. Because accounting is a single aggregate with no per-obligation reservation, a matured withdrawal and a relock (or a rotate) race for the same pool on a first-come basis.

Impact

Medium. A user who has completed the full unbonding period can be blocked from `completeWithdraw()` because another user's `relock()` (or an operator `rotate()`) drained the shared reserve first. There is no theft or permanent loss, but there is a real cross-user liveness failure. The permissionless vector is capital-bounded (the relocker locks its own funds into the new tier, and the moved reserve becomes strategist-borrowable), so it is more of an innocent race plus repeatable grief that forces additional strategist `repay()` calls than a cheap attack—but it can persistently delay honest exits.

Recommendation

Earmark reserve for matured/pending exits and let reserve-movers consume only the surplus above it:

```
uint256 public earmarkedWithdrawBase; // committed to active unbonding obligations

// when an unbonding is finalized in _executeInitiateWithdraw:
earmarkedWithdrawBase += request.amount;
// (and decrement it in take() when the exit is paid)

// in regive() and rotate(), gate on surplus, not the full reserve:
uint256 surplusBase = withdrawBase > earmarkedWithdrawBase
    ? withdrawBase - earmarkedWithdrawBase : 0;
if (worthBase > surplusBase) { /* defer (regive) or revert (rotate) */ }
```

This single change fixes both consumers: permissionless `regive` can no longer take reserve owed to a matured exit, and `rotate` is likewise bounded to genuine surplus (subsuming the `rotate`-unbounded concern in `medium-06`). Track the earmark in the same denomination used by `take()`'s check so the guarantee is exact.

Developer Response

Fixed in commit bd447c9: - Added earmarking of withdrawal funds

2.6.4 Algebra TWAP adaptor misdetects staleness via tick-cumulative equality

`OracleAlgebraAdaptor` treats equal tick cumulatives as "stale/no trades", but equal cumulatives simply mean the average tick over the window is 0, i.e., a price of ~1.0, which is the most common case for a stable/stable pair. The adaptor discards the correct price and can revert with `StaleOracleData` exactly at parity.

Technical Details

`_tryGetTWAP()` marks a window successful only if the two tick cumulatives differ:

```
if (tickCumulatives[0] != tickCumulatives[1]) {
    tick = int24((tickCumulatives[1] - tickCumulatives[0]) / int56(uint56(period)));
    success = true;
}
```

The tick cumulative is the running sum of the pool's tick over time.
`tickCumulatives[1] - tickCumulatives[0] == 0` means the *average tick over the window is 0*. For a well-pegged stable/stable pool at parity, this is a legitimate and expected reading, not a staleness signal. Conversely, the check does not detect genuine staleness at all: on a pool with no recent trades, the cumulative keeps growing linearly using the last tick, so a stale-but-nonzero average still passes as "fresh".
When the average is 0, `_getPoolTWAP()` falls through `twapPeriod` → `twapPeriodMiddle` → `twapPeriodLongest`, and if the average remains 0 across all three windows it reverts `StaleOracleData` (line 211).

Impact

Medium. For the protocol's primary use case, the adaptor systematically rejects the correct price and, at exact parity across all fallback windows, reverts, bricking mint/unmint for that asset while nothing is actually wrong. Because `_aggregateOraclePrices()` requires all oracles to succeed, this is a DoS risk.

PoC (if any)

1. A USDT/USDC Algebra pool trades symmetrically around tick 0, so the TWAP average tick is 0 over 15m, 2h, and 24h.
2. `_tryGetTWAP` returns `success = false` for all three windows.
3. `_getPoolTWAP` reverts `StaleOracleData`; every mint/unmint using this oracle reverts, even though a price of approximately 1.0 is valid and current.

Recommendation

Do not use cumulative equality as a freshness proxy. Derive freshness from the oracle's own last-observation timestamp, and accept a zero average tick as the valid price 1.0. Compute the TWAP tick unconditionally from the cumulative delta over `period`.

Developer Response

Fixed in commit f1e4faaf: - Removed the `OracleAlgebraAdaptor` contract

2.6.5 Pull oracle adapters are unusable

The two pull-based oracle adaptors (`OraclePromisPullAdaptor`, `OracleRedStoneAdaptor`) can never return a price through the production call path, because `ProTokenOperations` always calls them with empty `bytes`. Configuring either as a price source permanently reverts every mint and unmint.

Technical Details

The only production call site of the adaptor interface is `ProTokenOperations._aggregateOraclePrices()`, which invokes every configured oracle as:

```
1 uint256 price = IOracleAdaptor(_oracles[i]).getOraclePriceForAsset(_asset, "");
```

The `bytes data` parameter is hardcoded to `""`, but both pull adaptors require it:

- `OraclePromisPullAdaptor.getOraclePriceForAsset()` expects `price (32 bytes) + n × [timestamp (32) + signature (65)]` in `data` and reverts immediately with `InvalidInputs` when `data.length < 32`.
- `OracleRedStoneAdaptor.getOraclePriceForAsset()` calls RedStone's `getOracleNumericValuesAndTimestampFromTxMsg()`, which parses the signed payload from the **calldata of the current external call**. Since `ProTokenOperations` builds that calldata through a typed interface call, the RedStone payload can never be present, and the base contract reverts (`CalldataMustHaveValidPayload`).

Both adaptors therefore revert unconditionally when reached through the real price flow. `_aggregateOraclePrices` additionally requires **all** configured oracles to answer successfully, so a single pull adaptor in `oraclePriceSources` (or as the proToken `oraclePriceSource`) bricks `_calculateMintAmount` / `_calculateUnmintAmount`.

Impact

Medium. If either pull adaptor is configured for a yAsset or for proUSD, all mints and unmints for that asset revert until an admin reconfigures the oracle settings. This causes a DoS of the core protocol through the use of components shipped in this same codebase. If they are never configured, they are dead code.

Recommendation

Either remove the pull adaptors from the deployable set, or redesign the price plumbing so callers can forward per-oracle payloads end-to-end.

Developer Response

Fixed in commit f1e4faa: - Removed the `bytes` parameter - Removed the `OraclePromisPullAdaptor` and `OracleRedStoneAdaptor` contracts

2.7 Low Findings

2.7.1 Duplicate yield handlers double-count liquidity and can block redemptions

Technical Details

`YAssetOperationsHandler.setYProtocolHandlers()` validates array lengths, nonzero addresses, and total allocation, but it does not require handler addresses to be unique. A configuration such as `[H, H]` is therefore accepted.
Deposits through `_allocateToHandlers()` remain conserved: the allocation portions are deposited into the same handler in two calls, so handler `H` ultimately holds only the actual total deposited amount. However, `previewPayOut()` iterates over both entries and adds `H.getBalance()` twice. `getYAssetInfo()` has the same double-counting problem.
This creates a deterministic false positive in the ProTokenOperations instant-liquidity branch. If `previewPayOut()` returns true from the duplicated balance, `payOut()` withdraws the real balance from the first occurrence. The second occurrence then has no remaining balance, and `payOut()` ultimately reverts with `InsufficientBalance`. `_executeUnmint()` propagates the revert instead of entering the asynchronous queue.

Impact

Low. Once configured, reported backing and previewed liquidity are inflated without any corresponding assets. Unmint and strategic-unmint transactions for amounts between the real and double-counted balance repeatedly revert instead of falling back to the queue. Any off-chain NAV or monitoring process consuming `getYAssetInfo()` can also overstate protocol backing.

Recommendation

Reject duplicate handler addresses in `setYProtocolHandlers()`. The function already clears every old handler's `isProtocolHandler` flag before inserting the new set, so at insertion time the flag is `true` only if the same address appeared earlier in the new array. Checking it before writing is sufficient:

```
    for (uint256 i = 0; i < handlersLength; i++) {
        address handlerAddr = _handlers[i];
        uint256 allocation = _allocations[i];

+       if (isProtocolHandler[handlerAddr]) revert DuplicateHandler();
        isProtocolHandler[handlerAddr] = true;
```

Developer Response

Fixed in commit 05c72e1: - Added a duplicate check

2.7.2 The unmint fee is unbounded

Technical Details

`ProTokenSettings.setYAsset()` stores `unmintFeePer` without any bound. The field is documented in wad precision, where `1e18` represents 100%.
The fee is applied in `ProTokenOperations._executeUnmint()`:

```
uint256 feeAmount = (yAssetToReceiveAmount *
    yAssetSettings.settings.unmintFeePer) / USD_PRECISION;
yAssetToReceiveAmount -= feeAmount;
```

In addition, `_executeUnmint()` runs during `finalizeUnmintRequest()` and reads the yAsset settings fresh, so the fee charged is the one configured at finalization time, not the one in effect when the user created the request. A fee update therefore applies retroactively to every pending request, and the request's `minAmountOut` — fixed at creation and covered by the authority proof — is the user's only protection.

Impact

Low. A single unvalidated parameter can silently turn redemptions into near-total value capture for requests without slippage protection, or disable all approved unmints into the affected yAsset. Retroactive application to already-pending requests aggravates the issue. An explicit on-chain fee bound also removes this lever from the trust assumptions a proUSD holder must accept.

Recommendation

Introduce a `MAX_UNMINT_FEE` constant at a defensible business bound (e.g. `1e17` = 10%).

Developer Response

Fixed in commit bb00e84: - The unmint fee is now bounded at 10%

2.7.3 The module pause controls are disconnected from each other and `StrategyVault` has no pause mechanism of its own

Technical Details

Core and yield modules implement their circuit breaker by reading `ProTokenSettings.isPaused()`. For example, `ProTokenOperations.whenNotPaused` uses the central setting.
The vault subsystem diverges in two ways:

- `ProTokenPlus` is pausable, but only locally: it inherits its own `PausableUpgradeable` state.
- `StrategyVault` has no pause mechanism of its own: it neither inherits a pausable implementation nor checks `ProTokenSettings.isPaused()`. Its functions are stopped only where the call path happens to be gated elsewhere: `borrow()` / `repay()` pass through `strategicUnmint()` / `strategicMint()`, which check the central pause, and the `onlyProTokenPlus` functions are covered by the local ProTokenPlus pause. The directly callable admin/operator functions `rotate()`, `claimGrowth()`, and `claimYield()` have no pause gate of any kind.

Impact

Low. Incident response is fragmented across two disconnected pause controls, and StrategyVault's direct operator functions answer to neither: with both `ProTokenSettings` and `ProTokenPlus` paused, `rotate()`, `claimGrowth()`, and `claimYield()` remain callable. In particular, `claimGrowth()` transfers proUSD to an arbitrary recipient, so containment of a compromised operator key relies solely on rotating the role via `ProTokenSettings.setOperator()`.

Recommendation

Both `ProTokenPlus` and StrategyVault's directly callable privileged functions should consult one canonical circuit breaker: the central `ProTokenSettings.isPaused()`.

Developer Response

Acknowledged. It is intended.

2.7.4 Rotating StrategyVault in settings can freeze existing ProTokenPlus withdrawals

Technical Details

`ProTokenPlus` resolves the active vault address live from `ProTokenSettings.getStrategyVault()` at the moment each deposit, relock, or withdrawal completion executes. No position, pending deposit, or unbonding request stores the vault address that actually holds its backing. When `ProTokenSettings.setStrategyVault()` swaps the global vault pointer, every subsequent `ProTokenPlus` operation immediately switches to the

new vault, but the old vault continues to hold all pre-rotation user backing and withdrawal reserves.
After rotation, `ProTokenPlusOperations._executeDeposit()` sends new deposits to the replacement vault, while `executeCompleteWithdraw()` attempts to pull matured unbondings from the same replacement vault. A user with a pre-rotation unbonding that matured before the pointer change will find their `completeWithdraw()` call routed to the new vault, which has no record of their withdrawal and holds no reserve to cover it. The call reverts even though the old vault still holds the funded reserve.
The old vault also becomes cut off from normal protocol settlement.
`ProTokenOperations.onlyStrategyVault` checks `IProTokenSettings(proTokenSettings).getStrategyVault()` to authorize strategist mint and unmint operations. Once the settings pointer moves to the new vault, the old vault's `repay()` can no longer call `strategicMint()` because the authorization check fails. The old vault cannot replenish its withdrawal reserve or unwind its backing through the standard strategist path.
The same split can occur from the vault side: `StrategyVault.setProTokenPlus()` allows independent rebinding of the vault's frontend reference with no migration or quiescence check.

Impact

Low. Changing the active vault pointer while live ProTokenPlus obligations exist can freeze pre-rotation user withdrawals. A legitimate unbonding that matured before rotation will revert when the user calls `completeWithdraw()` because the call is routed to the new vault, which holds no reserve for that cohort. The old vault retains the user's backing and funded withdrawal reserve but is no longer reachable through the live protocol entrypoints. New deposits after rotation flow into the replacement vault, splitting a single product's liabilities across multiple unrelated custody backends.
The old vault loses access to strategist settlement operations because `ProTokenOperations` authorization follows the live settings pointer. Without a built-in migration flow, governance must either roll the pointer back (potentially stranding users who interacted with the new vault in the meantime) or construct a bespoke recovery procedure. This leaves no safe on-chain recovery path and can strand user funds across vaults until manual intervention restores the correct linkage.

Recommendation

Prevent live vault rebinding unless an explicit migration flow is active. At minimum, make `setStrategyVault()` revert while ProTokenPlus has active positions, pending deposits, or active unbondings. If vault rotation must be supported, pause user actions, migrate or drain old-vault assets and ledgers in a single transaction, then update the pointer. Apply the same guard to `StrategyVault.setProTokenPlus()` to close the analogous unverified path. Consider binding each position and unbonding request to the vault address that holds its backing at creation time, so that completion operations can resolve the correct custodian regardless of subsequent pointer changes.

Developer Response

Acknowledged. `StrategyVault` rotation is very highly unlikely to happen once the protocol is set. Additionally, it will be documented.

2.7.5 Admin proUSD address rotation desynchronizes StrategyVault from proUSD+ flows

Technical Details

`StrategyVault` caches the proUSD token address once during initialization by reading `proToken` from `ProTokenSettings.getProTokenInfo()` and storing it in local state. All subsequent vault operations—deposits, withdrawals, strategist borrows, and repayments—reference this cached value for token transfers, balance checks, and price lookups. Meanwhile, `ProTokenSettings.setProToken()` and `ProTokenPlus.setProUSD()` allow governance to change the canonical proUSD token address at runtime. After such a change, `StrategyVault` continues using the old cached token while the rest of the system operates on the new token:

- `ProTokenPlusOperations.executeCompleteWithdraw()` prices withdrawals using the new token via `_convertFromBase()` and expects `StrategyVault.take()` to return that token, but `take()` transfers the old cached token back to `ProTokenPlus`. When `ProTokenPlus` then attempts `IERC20(proUSD).safeTransfer(caller, totalAmount)` with the new token, the transfer fails because it received the old token.
- Fresh deposits fail because `ProTokenPlus` approves and transfers the new token to the vault, but `StrategyVault.give()` tries to pull the old cached token via `IERC20(proToken).safeTransferFrom()`.
- Strategist flows break because `StrategyVault.borrow()` and `repay()` account against the old cached token, while `ProTokenOperations.strategicMint()` and `strategicUnmint()` mint or burn the new token from settings.

`StrategyVault` exposes `setProTokenOperations()` to update the operations handler but provides no corresponding setter to refresh the cached `proToken`. Recovery requires rolling the token address back or performing a coordinated upgrade and migration.

Impact

Low. A live token-address rotation through `setProUSD()` or `setProToken()` renders proUSD+ deposits, matured withdrawals, and strategist reserve-management flows unavailable. Users who completed unbonding cannot complete withdrawals even though the unbonding period elapsed. The vault remains locked to the old token while the wider system operates on the new token, and no on-chain path exists to resynchronize `StrategyVault` without rolling back the configuration change or upgrading the contract. The failure persists until governance intervenes with a rollback or migration.

Recommendation

Treat proUSD address replacement as an unsupported operation unless a coordinated migration flow exists. The safest fix is to make the token reference immutable for live deployments by removing `setProToken()` and `setProUSD()` or by adding an admin-gated `setProToken(address)` function to `StrategyVault` that can resynchronize the cached token. If token-address rotation remains a supported admin action, enforce atomic updates across `ProTokenSettings`, `ProTokenPlus`, and `StrategyVault` with an explicit migration procedure that updates all three modules in a single transaction or sequence, preventing desynchronization. At minimum, document that changing the proUSD address requires an upgrade or migration of the vault itself.

Developer Response

Acknowledged. Changing the proUSD address is very highly unlikely to happen once the protocol is set. Additionally, it will be documented.

2.7.6 Pending mint escrows can be stranded by operations rotation or yAsset removal

Technical Details

The `createMintRequest()` function escrows the user's yAsset into the contract and stores only basic request data: `user`, `receiver`, `yAsset`, `amount`, and `minAmountOut`. The request does not capture the settlement-critical routing addresses needed to complete the mint later.
When a user finalizes a pending request via `finalizeMintRequest()` on the approve path, the code re-reads the current configuration from `ProTokenSettings`. The `_prepareMintProToken()` call inside `_mintProToken()` loads fresh yAsset settings, and `_executeMint()` transfers the escrowed yAsset to the handler address found in those settings (`settings.yAssetSettings.settings.yOperationsHandler`), then calls `distributeYAsset()` on that handler.
The handler's `distributeYAsset()` method accepts the no-pull transfer path only when `msg.sender == getProTokenInfo().proTokenOperations`. This check resolves `proTokenOperations` from live settings at execution time, not from the proxy that originally created and holds the escrow.
As a result, two maintenance actions can brick pending mint requests:

1. **Operations rotation**: `ProTokenSettings.setProTokenOperations()` updates the registered `proTokenOperations` address without checking for outstanding escrows. After rotation, the old proxy that holds the escrowed yAsset is no longer recognized by `distributeYAsset()`, which reverts with `Unauthorized()`.
2. **yAsset removal**: `ProTokenSettings.setYAsset()` and its counterpart removal path do not verify that pending mint escrows for that yAsset have been cleared. After a yAsset is deregistered, `_getYAssetSettings()` reverts, blocking the approve settlement path.

The only on-chain recovery mechanism for these stranded escrows is the `PROOF_OF_RETURN` branch in `finalizeMintRequest()`, which refunds the original user without touching the handler. However, this requires the authority to sign a fresh return proof and the original requester to submit the finalization transaction. There is no admin-controlled timeout or forced-refund path for stale mint escrows.

Impact

Low. A user's deposited yAsset can become stuck in the legacy `ProTokenOperations` proxy if governance changes core configuration before that mint request is finalized. The user does not lose funds immediately, but approval settlement is bricked until governance restores compatibility, upgrades the legacy proxy, or the authority and user cooperate on a return-proof refund. Address-based operations migrations and asset removals are not safe in the presence of pending mint escrows. This undermines maintenance flexibility and incident response by coupling recovery to user cooperation or manual configuration rollback.

Recommendation

Make pending mint requests configuration-stable or block unsafe maintenance while they exist. Snapshot the settlement-critical addresses (`yOperationsHandler`, `proToken`) into each request at creation time so that later config changes do not invalidate already-escrowed requests. Alternatively, prevent `setProTokenOperations()` and `removeYAsset()` for affected assets until all pending mint escrows for that asset or operations instance are cleared. Adding an admin timeout-based refund path for stale mint escrows would further reduce recovery risk and decouple incident response from user availability.

Developer Response

Acknowledged. We will disallow deposits before removing support for a yAsset. Pause functionality for the yAsset already exists to disallow deposits.

2.7.7 Missing handler-asset validation lets a miswired yAsset route fail open

Technical Details

The protocol expects each `YAssetOperationsHandler` instance to manage exactly one token, stored in its `yAsset` field at initialization. However, `ProTokenSettings.setYAsset()` never verifies that the configured handler actually manages the same asset being registered. When asset A is paired with a handler initialized for asset B, the mint and unmint flows operate on different tokens.
During a mint, `ProTokenOperations._executeMint()` transfers the settings-configured asset (A) to the handler and calls `distributeYAsset(_amount)` without passing a token address. The handler's internal logic in `_allocateToHandlers()` then uses its own stored `yAsset` (B) for all operations: `IERC20(yAsset).forceApprove(...)` and `IYieldProtocolHandler.depositYieldAsset(...)` move B into downstream yield protocols, while the deposited A sits untracked on the handler address. The mint only succeeds if the handler can complete allocation using its direct B balance or has no downstream handlers configured; otherwise the transaction reverts atomically.
If the mint does succeed, A accumulates as an invisible balance. Later, when a user burns proUSD through the asset-A route, `_executeUnmint()` calculates the redemption amount using A's price data but delegates the payout to the mismatched handler. The handler's `previewPayOut()` and `payOut()` methods operate entirely on B: they check B liquidity and transfer B to the user if available. Because USDC and USDT both use 6 decimals and trade near $1, the wrong-asset payout is not limited to rounding dust.
The handler's `getYAssetInfo()` and all other view functions report only B balances, so the stranded A tokens remain invisible to normal accounting and removal safety checks. The configuration error turns an operational mistake into a live cross-asset state instead of failing at registration time.

Impact

Low. A stale or incorrect asset-to-handler wiring can cause deposits of asset A to accumulate as untracked balances on the handler address. If the mismatched handler holds suitable direct or reserve B liquidity, users can receive asset B when redeeming through the asset-A route. The wrong-asset payout is material for the in-scope USDC and USDT pairs given their identical

decimals and expected price parity. The handler can be drained of B using requests nominally made for A, while the received A balances are ignored by all reporting views. Although top-level stranded tokens are recoverable through the handler's admin `emergencyWithdraw()` path and the mapping can be repaired, the issue remains localized to the misconfigured asset until detected and corrected.

Recommendation

In `ProTokenSettings.setYAsset()`, add a validation check that requires `IYAssetOperationsHandler(_settings.yOperationsHandler).getYAsset() == _yAsset` before storing the mapping. This ensures the handler's managed asset matches the registered yAsset at configuration time.

```
   function setYAsset(
       address _yAsset,
       ProTokenSettingsTypes.YAssetSettings memory _settings
   ) external override onlyAdmin {
       if (_yAsset == address(0)) revert ZeroAddress();
       if (_settings.yOperationsHandler == address(0)) revert ZeroAddress();
+      if (IYAssetOperationsHandler(_settings.yOperationsHandler).getYAsset() !=
    _yAsset)
+          revert HandlerAssetMismatch();
       yAssets.add(_yAsset);
       // ... rest of function
   }
```

For defense in depth, `setYProtocolHandlers()` should also verify that each downstream yield handler reports the same managed asset. Additionally, `distributeYAsset()` could perform a sanity check that the handler actually received the expected token amount before proceeding with allocation.

Developer Response

Fixed in commit 8b9e747: - Added defense in depth; we now compare the yAsset in the settings struct with the supplied yAsset address

2.7.8 Supported `usdCap = 0` strategist round trips can underfund the proUSD+ withdrawal reserve

Technical Details

`StrategyVault` manages two segregated proUSD pools: a strategist-drawable backing pool (`depositProUSD` against `depositBase`) and a withdrawal reserve (`withdrawProUSD` against `withdrawBase`). When a strategist borrows through `borrow()`, the vault reduces `depositBase` by the requested USD amount and burns the corresponding proUSD calculated at `lastPrice`. The burned proUSD is forwarded to `ProTokenOperations.strategicUnmint()`, which returns yAsset tokens. On repayment, the strategist returns yAsset through `repay()`, which forwards it to `ProTokenOperations.strategicMint()` to mint proUSD back. The vault then credits the withdrawal reserve with the minted proUSD and derives a new base obligation by multiplying `proUSDMinted * lastPrice`.

The accounting becomes asymmetric when a strategist-reachable yAsset is configured with `usdCap = 0` and proUSD trades above parity. In `_calculateUnmintAmount()` and `_calculateMintingAmount()`, when `usdCap` is zero, both apply a legacy 1:1 token cap. At a proUSD price of $1.10, borrowing 110 base burns 100 proUSD through `borrow()`, but `strategicUnmint()` returns only 100 USDC because the 1:1 cap limits the yAsset payout to the proUSD token count. The strategist then repays exactly those 100 USDC. `strategicMint()` mints approximately 90.9 proUSD back (100 USDC at $1.10 per proUSD), and `repay()` credits the withdrawal reserve with only 100 base (90.9 proUSD × $1.10). Meanwhile, the original user position still carries a 110 base withdrawal claim. When the user completes withdrawal via `ProTokenPlusOperations.executeCompleteWithdraw()`, `_convertFromBase()` converts 110 base back to 100 proUSD at the current price, but the reserve holds only 90.9 proUSD. `StrategyVault.take()` reverts with `WithdrawReserveUnderfunded`, blocking the withdrawal until an external recapitalization.
The drift occurs because `borrow()` and `repay()` use `lastPrice` to reconcile base and proUSD amounts, while `strategicUnmint()` and `strategicMint()` apply independent live conversion logic in `ProTokenOperations`. The 1:1 cap in the `usdCap = 0` mode is not invertible: unmint caps the yAsset payout to match the proUSD token count, and mint later caps the proUSD minted to match the yAsset token count, so a round trip at proUSD > $1 irreversibly loses reserve.

Impact

Low. A normal strategist borrow and repay cycle can leave the withdrawal reserve holding insufficient proUSD to satisfy matured user withdrawals. When a user calls `ProTokenPlus.completeWithdraw()` after unbonding, the transaction reverts, freezing the withdrawal until the vault is manually recapitalized or the strategist over-repays. Repeated borrow/repay rounds can accumulate a structural deficit, requiring ongoing off-protocol subsidies to restore solvency. The issue does not steal funds but blocks legitimate user exits and undermines the vault's promise of timely withdrawals.

Recommendation

Record an explicit strategist debt obligation when `borrow()` is called and require `repay()` to settle that debt directly before crediting the withdrawal reserve. The vault should maintain a mapping of outstanding borrow amounts in base terms and validate that each repay operation restores at least the recorded base obligation, rather than deriving the reserve credit from whatever `strategicMint()` happens to mint. As an immediate guardrail, disallow `usdCap = 0` for any yAsset reachable through `StrategyVault` borrow/repay paths, or modify the strategist mint and unmint conversions to be symmetric by applying the same cap direction on both legs.

Developer Response

Fixed in commit d360232: - Removed the USD else branch in which the cap was 0

2.7.9 Ordered fail-hard withdrawals can strand later healthy liquidity

Technical Details

The `YAssetOperationsHandler.payOut()` function attempts to satisfy redemptions by first using unallocated reserve assets held directly on the contract, then unwinding yield handlers in strict configuration order. When reserve assets alone cannot cover the requested amount, the function iterates through the `protocolHandlers` array and calls each handler's `withdrawYieldAsset()` for its expected portion. If any handler reverts during withdrawal, the entire payout operation reverts immediately, and handlers later in the array are never attempted.
This behavior creates a liveness issue when multiple handlers are configured for the same yAsset and an earlier handler becomes temporarily non-withdrawable while still reporting a positive balance. Both `AaveV3YieldHandler` and `MorphoYieldHandler` can exhibit this pattern: `AaveV3YieldHandler.getBalance()` returns `IAToken(aTokenAddress).balanceOf(address(this))`, and `MorphoYieldHandler.getBalance()` returns `expectedSupplyAssets()`. Both values can remain positive even when the underlying lending protocol cannot honor a withdrawal request due to high utilization or reserve illiquidity.
The preview function `previewPayOut()` simply sums `IERC20(yAsset).balanceOf(address(this))` and each handler's `getBalance()` return value. It does not account for whether handlers can actually withdraw on demand. When `ProTokenOperations._executeUnmint()` and `strategicUnmint()` evaluate `if (yOps.previewPayOut(amount))` to decide between instant payout and async unmint queueing, they may receive `true` based on aggregate balance even when the first handler cannot withdraw. The subsequent `yOps.payOut()` call then reverts before reaching later healthy handlers, and the transaction rolls back entirely.
The fail-hard ordering means that a single non-withdrawable handler at the front of the list can block access to all liquidity held by later handlers, even if those later sources alone would be sufficient to cover the full redemption amount.

Impact

Low. When an earlier handler in the `protocolHandlers` array cannot withdraw but still reports a positive balance, all redemptions that depend on later handler liquidity are blocked. User unmint finalizations fail despite sufficient aggregate router liquidity, and strategist redemptions via `strategicUnmint()` also revert. The router remains solvent in aggregate, but handler ordering becomes a single point of failure for redemption liveness. Users cannot access their funds through the instant path, and the protocol cannot rebalance vault liquidity or replenish withdrawal reserves until external conditions change or a privileged operator manually reorders handlers or pulls liquidity into reserve.

Recommendation

Make `payOut()` tolerant of individual handler failures. Instead of reverting when one handler cannot withdraw, attempt each handler conservatively and continue to the next handler when one reverts or returns less than requested. Only revert after all handlers have been exhausted and the remaining amount is still uncovered. In `ProTokenOperations`, treat a failed instant payout as a safe fallback to the async unmint queue rather than allowing the revert to propagate. This allows the router to degrade gracefully when one liquidity source is temporarily unavailable, while still using the instant path whenever sufficient healthy liquidity exists.
Additionally, prefer ordering handlers by proven instant liquidity rather than static configuration alone, or separate instant-liquidity sources from yield-only sources to reduce the risk of a single venue blocking the rest.

Developer Response

Fixed in commit 0a07fe0: - `payOut()` is now wrapped in `try/catch` to tolerate per-handler failures

2.7.10 Queued unmint claims can be leapfrogged by later instant redeemers

Technical Details

When an approved unmint cannot be paid out instantly, `ProTokenOperations._executeUnmint()` burns the user's proUSD and routes the request to `ProTokenUnmintHandler.createUnmintRequest()`. The handler records only accounting metadata: the user's address, the yAsset, and the amount owed. No yAsset is moved, reserved, or locked at that moment. The backing remains in `YAssetOperationsHandler` and its configured yield handlers, where it continues to appear as freely available liquidity.
Later, when new liquidity arrives for the same yAsset—whether from new mints, yield accrual, or withdrawals from external protocols—the system does not subtract outstanding queued obligations when evaluating instant payout eligibility.
`YAssetOperationsHandler.previewPayOut()` sums only the handler's direct balance and each yield handler's balance. `YAssetOperationsHandler.payOut()` follows the same logic: it pulls from the handler's balance and then iterates through yield handlers until the requested amount is satisfied. Neither function accounts for queued liabilities.
As a result, a later user who finalizes an approved unmint for the same yAsset can receive an instant payout using liquidity that arrived after an earlier user was queued. The earlier queued user must wait until an operator manually calls `ProTokenUnmintHandler.processNextUnmintBatch()` and transfers the required yAsset from their own wallet via `safeTransferFrom`. Because queued obligations are never netted out of available liquidity calculations, each replenishment can be consumed by new instant redeemers before operators fund the older batch.
This creates a priority inversion: once a user's proUSD is burned and their claim is queued, later redeemers can exit first by consuming newly available liquidity, while the earlier user holds an unsecured accounting claim with no reserved backing.

Impact

Low. Users whose proUSD has already been burned and whose unmint requests have been queued lose priority over subsequently arriving liquidity. Later users who finalize approved unmints for the same yAsset can receive instant payouts ahead of earlier queued claimants. The queued user's redemption is delayed until operators manually fund the batch from an external source, even when liquidity for that yAsset repeatedly flows back into the handler. This violates FIFO-style redemption expectations and subordinates already-burned claims to later requests, extending wait times and weakening the claim's economic seniority.

Recommendation

Track queued liabilities on-chain and exclude them from instant-liquidity checks. Maintain a reserved or netted balance per yAsset that `previewPayOut()` and `payOut()` subtract from available liquidity before allowing any later instant payout. Alternatively, route all redemptions for a yAsset through the queue whenever any unpaid queued liability exists, preserving FIFO

priority until the backlog is cleared. Either approach ensures that once a user's proUSD is burned into a queued claim, later redemptions cannot consume liquidity that should first satisfy that older obligation.

Developer Response

Fixed in commits 0a07fe0 and 63ca735: - Introduced queued liabilities on-chain. Withdrawal assets are now reserved. - Instant redemptions now require the backlog to be zero

2.7.11 ProTokenPlus ignores the configured proUSD price source and can freeze under supported oracle-mode pricing

Technical Details

The core mint/unmint system offers two proUSD pricing modes through `ProTokenSettings.ProTokenPriceSettings`. When `oraclePriceSource` is zero, `ProTokenOperations._getUsdRepresentationProToken()` reads the token's own `getUSDPrice()`. When `oraclePriceSource` is configured with a valid oracle adaptor, the same function switches to oracle-based pricing and ignores the direct token price.
ProTokenPlus and StrategyVault do not honor this configuration.
`ProTokenPlusOperations._convertToBase()` and `_convertFromBase()` always call `IProToken(proUSD).getUSDPrice()` directly when computing base-denominated values for tier minimums, deposit caps, position `worthBase`, and withdrawal payouts.
`StrategyVault._tryGetPrice()` does the same when settling growth and sizing borrow/repay operations against the three ledgers (`depositBase`, `withdrawBase`, `depositProUSD`, `withdrawProUSD`, `growthProUSD`).
`ProToken.setUSDPrice()` explicitly permits setting the direct price to zero when oracle-mode pricing is in use. In that state, `getUSDPrice()` reverts with `USDPriceDisabled()`. Core mint/unmint continue to operate through the configured oracle path, but ProTokenPlus/StrategyVault paths fail:

- `ProTokenPlus.createDepositRequest()` calls `_convertToBase()` to enforce the tier's USD minimum and reverts when the conversion cannot read a direct price.
- `ProTokenPlus.completeWithdraw()` calls `_convertFromBase()` to compute the payout and reverts for the same reason.
- `StrategyVault` functions such as `regive()`, `borrow()`, `repay()`, and `rotate()` all depend on `lastPrice` being non-zero, which is only updated if `_tryGetPrice()` can read a direct token price.

A secondary concern exists when the direct token price is live but drifts from the configured core oracle. In that case, ProTokenPlus and StrategyVault book base-denominated accounting (`depositBase`, `withdrawBase`, `totalDepositsBase`) using one price while core mint/unmint use another. This decouples the vault's internal USD ledger from the actual proUSD redemption economics, potentially overstating or understating deposit caps, position values, and reserve obligations.

Impact

Low. Once oracle-mode pricing is enabled and the direct token price is disabled or left stale, ordinary users cannot finalize ProTokenPlus deposits or complete matured withdrawals. Deposit

finalization reverts in `_convertToBase()` when enforcing the tier's minimum, and withdrawal completion reverts in `_convertFromBase()` when computing the payout amount. The ProTokenPlus product is frozen until an admin either re-enables the direct price or upgrades the code to respect the configured price source.
Vault management functions that depend on `lastPrice` also fail when `_tryGetPrice()` returns zero. `regive()`, `borrow()`, `repay()`, and `rotate()` all revert with `PriceUnavailable()` if the direct token price cannot be read, blocking reserve refills and strategist debt management.
If the direct price is live but diverges from the core oracle, ProTokenPlus and StrategyVault use a different proUSD valuation than the core layer. This mispricing propagates into deposit cap enforcement, position `worthBase` records, fixed reward calculations, and the three vault ledgers that govern reserve obligations and strategist debt. While concrete loss from this drift was not demonstrated end-to-end, the mismatch breaks the assumption that all system components operate on a single proUSD price.

Recommendation

Centralize proUSD price resolution behind one shared helper used by `ProTokenOperations`, `ProTokenPlusOperations`, and `StrategyVault`. That resolver should read `ProTokenSettings.proTokenPriceSettings`, prefer the configured oracle when present, and fall back to `IProToken.getUSDPrice()` only when no oracle source is configured. Replace all direct `IProToken(proUSD).getUSDPrice()` calls in `ProTokenPlusOperations._convertToBase()`, `_convertFromBase()`, and `StrategyVault._tryGetPrice()` with the new shared resolver.
As a temporary safeguard while any ProTokenPlus or StrategyVault code path still depends on direct token pricing, forbid `setUSDPrice(0)` by reverting when `_price == 0` and an active Plus or vault deployment exists. Once the shared resolver is deployed and all modules reference it, that safeguard can be removed.

Developer Response

- 0a07fe0
- proUSD now has only a single price source

2.7.12 Deposit requests can deadlock escrowed proUSD on invalid or stale pre-merge inputs

Technical Details

`executeCreateDepositRequest()` immediately escrows the caller's proUSD into the contract and stores the provided `unlockedPositionsToMerge` array inside the new `DepositRequest` struct. This array is arbitrary user input validated only at finalization time.
When `executeFinalizeDepositRequest()` runs, it first decrements `totalPendingDeposits` and marks the request `EXECUTED`, then unconditionally calls `_mergeUnlockedIfProvided(request.user, request.unlockedPositionsToMerge)` before processing either deposit approval or refund based on the authority-signed `proofKind`. If the merge array contains at least two IDs, `_executeUnlockedMerge()` validates that every listed position still exists, is owned by the requester, has status `ACTIVE`, is effectively unlocked, is not duplicated, and shares the same tier. Any violation reverts the transaction.

Position IDs stored in the request at creation time can become invalid or deactivated before finalization through legitimate user actions such as calling `executeUnlockedMerge()` on the same IDs, relocking them via `executeRelock()`, or withdrawing them. Once `_deactivatePosition()` marks a position inactive, its ID cannot be reactivated, and the stored merge array remains unchanged inside the pending request. Because finalization atomically applies the merge before either depositing or refunding, a revert during merge rolls back the status update, leaving the request stuck in `PENDING` with no on-chain path to skip the merge or cancel the request.

Impact

Low. A deposit request with invalid or later-stale merge IDs cannot be finalized through normal user flows, locking the escrowed proUSD inside the contract indefinitely. Because `_mergeUnlockedIfProvided()` is called before both the approval and refund branches, even an authority-signed `PROOF_OF_RETURN` cannot release the funds. The request owner loses access to the deposited amount unless governance deploys a contract upgrade or adds a specialized recovery path. This is localized to the requesting user and does not compromise shared protocol solvency or enable value theft by third parties.

Recommendation

Decouple refund finalization from optional pre-merge logic. For `PROOF_OF_RETURN`, skip `_mergeUnlockedIfProvided()` and transfer the escrowed amount directly to the requester. For `PROOF_OF_APPROVE`, either validate and lock the merge array at request creation, or treat merge as best-effort and proceed with deposit creation if merge inputs are stale. Alternatively, add an on-chain cancel function that allows the requester to reclaim escrowed proUSD from a pending deposit request without depending on mutable position state.

Developer Response

- a9334ea
- The merging of unlocked positions was moved to the `PROOF_OF_APPROVE` branch

2.7.13 Optimistic liquidity preview can block approved unmints

Technical Details

The protocol's unmint flow branches on `YAssetOperationsHandler.previewPayOut()` to decide whether to pay out instantly or queue the request in `ProTokenUnmintHandler`. The preview method returns `true` when the sum of idle reserve plus handler-reported balances meets or exceeds the requested amount. For Aave and Morpho handlers, `getBalance()` returns an accounting value—current aToken balance or `expectedSupplyAssets`—rather than the amount of underlying that can be withdrawn immediately.
When an external lending market becomes highly utilized, a handler's accounting balance can exceed the underlying liquidity available for withdrawal. In that state, `previewPayOut()` returns `true` because it sums the accounting balances, so `_executeUnmint()` chooses the instant branch and calls `yOps.payOut(...)`. The `payOut()` method attempts to withdraw the shortfall from handlers, but the handler's `withdrawYieldAsset()` call fails when the underlying

pool cannot provide the requested amount. The Aave handler at `AaveV3YieldHandler.withdrawYieldAsset()` reverts with `WithdrawFailed` if `IPool.withdraw()` returns less than requested, and the Morpho handler at `MorphoYieldHandler.withdrawYieldAsset()` behaves identically.
Because `_executeUnmint()` has no fallback logic to queue the request after a failed payout, the entire finalization transaction reverts. The unmint request remains in `PENDING` status, the user's proUSD stays escrowed in `ProTokenOperations`, and no batch is created. As long as the handler's accounting balance stays above the requested amount while withdrawable liquidity stays below it, every retry fails the same way. The same pattern also affects `strategicUnmint()`.

Impact

Low. Users holding approved unmint requests can become unable to redeem even asynchronously when handlers use accounting balances that exceed immediately withdrawable underlying. The queue fallback does not activate under the exact condition it is meant to handle—insufficient instant liquidity—so large unmint requests can remain stuck pending until privileged intervention or external liquidity returns. This breaks the protocol's documented async-unmint fallback and can materially delay exits for affected users during market stress.

Recommendation

Do not use `previewPayOut()` as the authoritative gate between instant payout and queueing. Wrap `yOps.payOut(...)` inside a `try/catch` block within `_executeUnmint()` and `strategicUnmint()`. On any failure, create the unmint request in `ProTokenUnmintHandler` with the full amount and emit the corresponding queued event. This allows the fallback path to activate when handlers cannot deliver the requested liquidity, regardless of their reported accounting balances.

```
  if (yOps.previewPayOut(yAssetToReceiveAmount)) {
-     yOps.payOut(_recipient, yAssetToReceiveAmount);
-     emit ProTokenUnmintInstant(...);
+     try yOps.payOut(_recipient, yAssetToReceiveAmount) {
+         emit ProTokenUnmintInstant(...);
+     } catch {
+         uint256 handlerRequestId = IProTokenUnmintHandler(
+             proTokenInfo.proTokenUnmintHandler
+         ).createUnmintRequest(_recipient, _yAsset, yAssetToReceiveAmount);
+         emit ProTokenUnmintQueued(..., handlerRequestId);
+     }
  } else {
      uint256 handlerRequestId = IProTokenUnmintHandler(
          proTokenInfo.proTokenUnmintHandler
      ).createUnmintRequest(_recipient, _yAsset, yAssetToReceiveAmount);
      emit ProTokenUnmintQueued(..., handlerRequestId);
  }
```

Apply the same pattern to `strategicUnmint()`. Longer term, expose a handler-specific "withdrawable now" metric rather than relying on accounting balances from `getBalance()`.

Developer Response

Fixed in commit 1246649: - `payOut()` is no longer called optimistically

2.7.14 Pending old-proToken unmint requests become unfinalizable after proToken changes

Technical Details

`createUnmintRequest()` transfers the currently configured proToken into `ProTokenOperations`, but the request only stores `user`, `receiver`, `yAsset`, `amount`, and `minAmountOut`. It does not store the proToken address that was actually escrowed:

```
ProTokenSettingsTypes.GetProTokenInfoResponse memory proTokenInfo =
    IProTokenSettings(proTokenSettings).getProTokenInfo();

IERC20(proTokenInfo.proToken).safeTransferFrom(msg.sender, address(this), _amount);

unmintRequests[requestID] = ProTokenOperationsTypes.Request({
    user: msg.sender,
    status: ProTokenOperationsTypes.Status.PENDING,
    receiver: _receiver,
    yAsset: _yAsset,
    amount: _amount,
    minAmountOut: _minAmountOut
});
```

During finalization, both the approve path and the return path read `getProTokenInfo()` again and use the current `proTokenInfo.proToken`:

```
IERC20(proTokenInfo.proToken).safeTransfer(request.user, request.amount);
```

```
IProToken(proTokenInfo.proToken).burn(address(this), _amount);
```

If `ProTokenSettings.setProToken()` is called while unmint requests are pending, old requests no longer settle against the token they escrowed. If `ProTokenOperations` has no balance of the newly configured proToken, finalization reverts and the old proToken escrow remains stuck. If the contract does hold new proToken from later activity, the old request can instead settle against that unrelated new-token balance.
A focused PoC demonstrated the second case: an old request created with the original proToken received a `PROOF_OF_RETURN` after the settings update, and the refund transferred the newly configured proToken while the original proToken escrow stayed locked in the contract.

Impact

Low. This requires an admin-controlled proToken settings change while unmint requests are pending, but a normal migration or configuration update can make old-token unmint requests unfinalizable. In the presence of later new-token liquidity, those stale requests can also settle against unrelated balances while leaving the originally deposited proToken stuck.

Recommendation

Bind each unmint request to the proToken address that was transferred in at request creation. Finalization should refund or burn that stored token, or `setProToken()` should be blocked while pending unmint requests exist.

Developer Response

Acknowledged. These functions are only meant for initialization. The ProToken address is very unlikely to be changed after deployment.

2.8 Gas Savings Findings

2.8.1 Unnecessary `timestampSeconds` variable can be removed

Technical Details

In `OracleChainlinkPushAdaptor.getOraclePriceForAsset`, `updatedAt` is returned by `latestRoundData()` and is already denominated in seconds. The code copies it into `timestampSeconds` and then only uses the copy:

```
uint256 timestampSeconds = updatedAt;

if (timestampSeconds > block.timestamp) {
    revert FutureOracleTimestamp();
}

if (block.timestamp - timestampSeconds > stalenessThreshold) {
    revert StaleOracleData();
}
```

The intermediate variable is unnecessary because `updatedAt` can be used directly.

Impact

Gas. Removing the redundant local variable slightly reduces bytecode and avoids an unnecessary stack assignment.

Recommendation

Use `updatedAt` directly in the future timestamp and staleness checks:

```
if (updatedAt > block.timestamp) {
    revert FutureOracleTimestamp();
}

if (block.timestamp - updatedAt > stalenessThreshold) {
    revert StaleOracleData();
}
```

Developer Response

Fixed in commit 53e28b0a: - Removed the unnecessary `timestampSeconds` variable

2.9 Informational Findings

2.9.1 Privileged roles can withdraw yAsset backing directly to themselves

Technical Details

`YAssetOperationsHandler.withdrawalYieldAssets()` allows `admin`, `operator`, and `externalBusiness` to withdraw managed yAsset from either the idle handler balance or a configured yield protocol handler:

```
if (
    msg.sender != IProTokenSettings(proTokenSettings).getExternalBusiness() &&
    msg.sender != IProTokenSettings(proTokenSettings).getOperator() &&
    msg.sender != IProTokenSettings(proTokenSettings).getAdmin()
) {
    revert Unauthorized();
}
```

When `_handler` is non-zero, the function withdraws from the configured Aave/Morpho handler. When `_handler == address(0)`, it withdraws from the unallocated balance held directly by `YAssetOperationsHandler`. In both cases, the resulting yAsset is transferred directly to `msg.sender`:

```
IERC20(yAsset).safeTransfer(msg.sender, actualAmount);
```

`withdrawalYieldAssetsMultiple()` exposes the same behavior for multiple handlers and transfers the aggregate amount to `msg.sender` at the end of the loop.
This means these privileged roles can take direct custody of assets backing the system. The contract does not restrict withdrawals to a reserve address, does not enforce a timelock, and does not perform an on-chain solvency or queued-liability check before transferring funds to the caller.

Impact

Informational. This appears to be an intended trust model rather than a smart contract bug, but it is a strong centralization assumption. A compromised or malicious `admin`, `operator`, or `externalBusiness` key can withdraw yAsset backing directly from the protocol's idle balances or yield positions. Users and integrators should understand that these roles have direct custody power over protocol funds.

Recommendation

Clearly document this trust assumption. The keys for `admin`, `operator`, and `externalBusiness` must be secured as high-value custody keys, ideally using multisigs, hardware-backed signing, strict operational controls, and monitoring. If the protocol wants to reduce this trust assumption later, route withdrawals to controlled reserve contracts and add timelocks, limits, or solvency checks around privileged collateral movement.

Developer Response

Acknowledged. This is the intended model. We will additionally document the roles.

2.9.2 New unmint requests can join a batch that is already processable

Technical Details

`ProTokenUnmintHandler.createUnmintRequest()` opens a new batch only when the current batch is processed, no batch exists, or the current timestamp is strictly greater than the batch duration boundary:

```
curBatch.createTimestamp + unmintBatchDuration < block.timestamp
```

However, `processNextUnmintBatch()` treats the batch as processable once the timestamp has reached that same boundary, because it only reverts while the boundary is still in the future:

```
if (unmintBatch.createTimestamp + unmintBatchDuration > block.timestamp)
    revert BatchStillProcessing();
```

At exactly `block.timestamp == createTimestamp + unmintBatchDuration`, the batch is both processable and still appendable. A user finalizing a queued unmint during that boundary timestamp can be added to a batch that has already matured instead of being placed in the next batch.
This also creates operational friction: if the operator prepared a balance or allowance based on the batch's previous `totalAmount`, the last-moment append increases the required transfer amount and can make `processNextUnmintBatch()` revert until funding is refreshed.

Impact

Informational. No assets are stolen, but the rollover boundary is inconsistent with the intended batch lifecycle. Users can skip nearly a full batch window at the exact boundary timestamp, and operators may need to refresh funding if a boundary-block append changes the amount required to process the batch.

Recommendation

Align the batch rollover condition with the processing condition. `createUnmintRequest()` should stop appending to the current batch once that batch is eligible to be processed. The direct fix is to change the strict `<` check to `<=`:

```
curBatch.createTimestamp + unmintBatchDuration <= block.timestamp
```

Developer Response

Fixed in commit 3e62f7c6: - Changed "<" to "<=" to include the boundary

2.9.3 Admin access-control pattern is duplicated across contracts

Technical Details

Most protocol contracts define their own `onlyAdmin` modifier, usually by reading the admin from `ProTokenSettings`:

```
modifier onlyAdmin() {
    IProTokenSettings settings = IProTokenSettings(proTokenSettings);
    if (msg.sender != settings.getAdmin()) revert NotAdmin();
    _;
}
```

The same pattern appears across core contracts, oracle adaptors, vault contracts, and yield handlers. Separately, `ProTokenSettings` hand-rolls two-step admin transfer with `proposeAdmin()` and `acceptAdmin()` instead of inheriting a standard ownership component. Because admin authorization and admin transfer are security-sensitive, duplicating the pattern across many contracts makes future changes easier to apply inconsistently. A new contract or upgrade can accidentally use a different admin source, miss the two-step transfer semantics, or introduce a subtly different revert/control path.

Impact

Informational.

Recommendation

Centralize the admin logic in a shared base contract that exposes the common `onlyAdmin` modifier and admin-source handling, then have the other contracts inherit from it. For the admin transfer flow itself, either use OpenZeppelin `Ownable2StepUpgradeable` / `Ownable2Step` where appropriate or mirror that behaviour in a single internal base instead of hand-rolling it per contract.

Developer Response

Acknowledged. This is an intended design choice.

2.9.4 `MorphoYieldHandler.withdrawYieldAsset(0)` validates withdrawal against the original zero

Technical Details

`IYieldProtocolHandler.withdrawYieldAsset` documents `amount == 0` as a full-balance withdrawal request. `MorphoYieldHandler.withdrawYieldAsset` follows that convention by replacing zero with `this.getBalance()`:

```
uint256 amountToWithdrawAssets = amount == 0
    ? this.getBalance()
    : amount;
```

However, after the Morpho withdrawal, the short-withdrawal check compares `assetsWithdrawn` against the original `amount`, not `amountToWithdrawAssets`:

```
if (assetsWithdrawn < amount) revert WithdrawFailed();
```

For `amount == 0`, this check can never catch an under-withdrawal.

Impact

Informational.

Recommendation

Compare the returned amount against `amountToWithdrawAssets`:

```
if (assetsWithdrawn < amountToWithdrawAssets) revert WithdrawFailed();
```

Developer Response

Fixed in commit 32721c1: - The correct variable ”amountToWithdrawAssets” is now being used

2.9.5 `setAToken()` can desynchronize accounting while Aave funds are deposited

Technical Details

`AaveV3YieldHandler.setYieldAsset()` and `setAavePool()` both check the current aToken balance and revert if funds are still deposited before changing core configuration. `setAToken` does not perform the same protection and can replace the aToken override at any time:

```
function setAToken(address _aToken) external onlyAdmin {
    _updateAToken(_aToken);
}
```

Because `_getATokenAddress()` always returns the override when `aToken != address(0)`, changing the override while funds are deposited makes `getBalance` and the withdrawal pre-check read the new token instead of the actual Aave reserve aToken.

Impact

Informational.

Recommendation

Apply the same no-deposited-funds guard used by `setYieldAsset()` and `setAavePool()`, or validate that the new `_aToken` equals `IPool(aavePool).getReserveAToken(yieldAsset)` before accepting it.

Developer Response

Fixed in commit a298ba5: - Added the same balance check before changing the aToken

2.9.6 Initializer rejects zero `_aToken` even though the handler supports pool-based aToken lookup

Technical Details

The initializer documents `_aToken` as an optional override, and `_getATokenAddress` already supports `aToken == address(0)` by querying `IPool(aavePool).getReserveAToken(yieldAsset)`. However, initialization currently reverts when `_aToken` is zero:

```
1 if (_aToken == address(0)) revert InvalidAddr();
```

This makes the dynamic lookup path unavailable at deployment time, even though the admin can later call `setAToken(address(0))` to enable it.

Impact

Informational. This does not create a security issue, but it makes deployment behavior inconsistent with the documented optional override and the implemented fallback logic.

Recommendation

Allow `_aToken == address(0)` during initialization if `aavePool` and `yieldAsset` are set, or update the documentation to state that an explicit aToken is required at deployment.

Developer Response

Fixed in commit 24a1a8a: - Removed the `aToken == address(0)` check

2.9.7 `IStrategyVault.claimYield()` NatSpec mismatch

Technical Details

The NatSpec for `IStrategyVault.claimYield()` states:

> Sweeps strategist-generated yield ... to the fixed yieldRecipient. **Permissionless trigger**; funds only ever go to the admin-set recipient.

But the implementation in `StrategyVault.claimYield()` restricts the caller:

```
1 function claimYield(
2     uint256 amount
3 ) external override onlyAdminOrOperator returns (uint256 claimed) {
```

So the call is gated to admin/operator, not permissionless.

Impact

Informational.

Recommendation

Reconcile documentation and code.

Developer Response

Fixed in commit 0d9867a0: - Fixed the mismatch in the comment. Removed the ”Permissionless” part

2.9.8 Global staleness threshold may be too loose for some Chainlink feeds

Technical Details

`OracleChainlinkPushAdaptor` uses one global `stalenessThreshold` for every mapped Chainlink feed. The threshold is initialized to `86400` seconds and can later be changed only globally through `setStalenessThreshold`.
This means all assets configured through `setAssetToPushOracleMappings` must share the same freshness tolerance, even though Chainlink feeds can have different expected heartbeat/update intervals. If the threshold is set high enough to support slower feeds, faster-moving feeds can be accepted long after their expected freshness window. If it is set low enough for faster feeds, slower feeds can unnecessarily revert as stale.

Impact

Informational.

Recommendation

Track staleness thresholds per asset/feed instead of globally, or add an upper bound and document that one adaptor instance should only be used for feeds with compatible heartbeat requirements.

Developer Response

Fixed in commit 63a65fd: - The staleness threshold is now per asset

2.9.9 Unused oracle payload parameter can be removed

Technical Details

Several `getOraclePriceForAsset` implementations accept the second `bytes calldata` parameter but never use it. For example, `OracleChainlinkPushAdaptor`, `OracleRedStonePushAdaptor`, `OracleRedStoneAdaptor`, and `OracleAlgebraAdaptor` derive prices from configured oracle mappings or TWAP routes instead of decoding caller-provided payload data.
The shared
`IOracleAdaptor.getOraclePriceForAsset(address asset, bytes calldata data)` interface

includes this parameter because pull-style adaptors such as `OraclePromisPullAdaptor` use `data` to decode and verify signed prices. However, for push/TWAP adaptors, the parameter constitutes dead API surface and creates unnecessary calldata/documentation noise.

Impact

Informational.

Recommendation

Either split the oracle interfaces so only payload-based adaptors expose the `bytes calldata data` parameter, or explicitly document the parameter as unused for push/TWAP adaptors. If interface compatibility is not required, remove the unused parameter from the affected adaptor functions.

Developer Response

Fixed in commit f1e4faad: - Removed the unused `bytes data` parameter

yAudit

Promis Smart Contracts

Completed 2026-07-17