Code Assessment
of the AMM + LT Hardening & YBLendingOracle Smart Contracts

PUBLIC

Date July 24, 2026

Produced for

By

Contents

1 Executive Summary

Dear Yield Basis team,

Thank you for trusting us to help you with this security audit. Our executive summary provides an overview of subjects covered in our audit of the latest reviewed contracts of AMM + LT Hardening & YBLendingOracle according to **Scope** to support you in forming an opinion on their security risks.

This is a focused delta review of hardening changes to `AMM.vy`, `LT.vy`, and the new `YBLendingOracle.vy`. The oracle receives most of our attention. It prices `LT` shares for external lending integrators, and its behaviour at the edge of the AMM's tradable region is where correctness and availability matter most.

Functional correctness is satisfactory. Discovered medium-severity issues were resolved. Access control is high. Privileged operations follow existing project conventions and are gated by the Factory admin. Arithmetic precision is good. `AMM` and `YBLendingOracle` share a system-wide `LEVERAGE = 2` invariant enforced by the Factory. However, `YBLendingOracle` users need to be aware and careful about rounding direction of the price computations.

Findings related to gas efficiency were resolved.

Findings marked as acknowledged are acceptable subject to the operational assumptions and conditions documented alongside each. If those conditions cease to hold, the findings need to be revisited.

In summary, we find that the codebase provides a satisfactory level of security.

It is important to note that security audits are time-boxed and cannot uncover all vulnerabilities. They complement but don't replace other vital measures to secure a project.

The following sections will give an overview of the system, our methodology, and the issues uncovered.

Sincerely yours,
ChainSecurity

2 Assessment Overview

In this section, we briefly describe the overall structure and scope of the engagement, including the code commit which is referenced throughout this report.

2.1 Scope

The assessment was performed on the source code files inside the Yield Basis repository based on the documentation files. The table below indicates the code versions relevant to this report and when they were received.

Version	Date	Commit Hash	Note
1	15 April 2026	fb3c7466567822ba3fe9fe587e3ea15afe8a1502	Initial review
2	27 June 2026	8dc5388376bfab6fc54f4f2d5f31d07f204bc71d	Fix review
3	20 July 2026	00f0019d582a25dd4e87ef64e5d549b76ffdca10	Fix review

The smart contracts are written in Vyper version `0.4.3`.

This is a focused delta review covering specific changes to three components. The files in scope are:

```
contracts/
├── LT.vy
├── AMM.vy
└── utils/
    ├── YBLendingOracle.vy
    └── YBPriceProxy.vy
```

2.1.1 Excluded from scope

All other contracts in the repository, including `contracts/utils/liboracle.vy`, are out of scope for this delta review.

In **Version 2**, `YBLendingOracle.vy` no longer depends on `contracts/utils/liboracle.vy`. Its helpers (`_A_at_last_timestamp`, `_scaled_A_raw_from_A`, `_portfolio_value`) are now imported from the `twocrypto-ng` submodule at `contracts/twocrypto_lp_oracle`. The imported helpers themselves are out of scope.

3 System Overview

This assessment is a focused delta review of AMM + LT Hardening & YBLendingOracle. For a comprehensive description of the Yield Basis system architecture, including core formulas, all contracts, deposit/withdraw flows, staking, and admin fees, we refer the reader to the prior ChainSecurity audit report.

The scope of this review covers three targeted changes to the Yield Basis codebase in version **Version 1**, as defined in the Assessment Overview. The changes are summarized below.

3.1 LT: `emergency_withdraw` Without Kill

`LT.emergency_withdraw()` is an alternative withdrawal path that works regardless of whether the AMM is killed. Unlike `withdraw()`, which uses `remove_liquidity_fixed_out()` to target the exact debt amount in stablecoins, `emergency_withdraw()` uses balanced `remove_liquidity()`. If balanced removal yields insufficient stablecoins to cover the user's debt share, the caller must supply the shortfall.

When the AMM is not killed, only the position owner may call `emergency_withdraw()` for their own shares. The function performs full value accounting via `_calculate_values()`, updates admin fees, total liquidity, staker balances, and checkpoints the gauge. When killed, `_calculate_values()` is skipped and `totalSupply` is used directly. Privileged actors (factory admin, emergency admin) may withdraw on behalf of other users, but the withdrawn assets must be sent to the user's account (not to the privileged actor).

Both `withdraw()` and `emergency_withdraw()` (when not killed) depend on `value_oracle()`. `emergency_withdraw()` now supports failing `value_oracle()` calls.

3.2 AMM: Relaxed Safety Checks for Improving Trades

`AMM.get_x0()` enforces safety limits that are checked on every trade. In the updated version, `AMM.exchange()` relaxes these checks for trades that move the system toward the target ratio of `debt / collateral_value = 0.5`. Trades moving away from the target still undergo full safety checks.

3.3 New Contract: `YBLendingOracle`

`YBLendingOracle` is a view-only oracle that reports the per-token value of an `LT` position. It provides two functions:

- `price_in_usd(lt, use_balances)` returns the per-LT-token value denominated in USD.
- `price_in_asset(lt, use_balances)` returns the per-LT-token value denominated in the underlying asset.

The oracle computes the Curve LP oracle price by solving for portfolio value on the StableSwap invariant at the pool's `price_oracle` using the interpolated `A` parameter (via `liboracle.vy`). The LP oracle price is combined with `AMM.get_x0()` to derive the per-token value, adjusted for leverage and admin fees.

The oracle calls `amm.get_state()` via `raw_call` with `revert_on_failure=False`. If `get_x0()` reverts (negative discriminant), the oracle falls back to balance-based valuation: `collateral * lp_price - debt`. The `use_balances` parameter forces this fallback explicitly.

`liboracle.vy` uses bisection to find what the Curve pool balances would be at a given target price, then computes the total portfolio value from those balances.

3.4 New Contract: `YBPriceProxy`

Added in **Version 3**, `YBPriceProxy` is a per-market `price()` forwarder for `YBLendingOracle`. `YBLendingOracle` is a singleton that prices any market by `LT` address. Many price-feed consumers need a fixed, argument-less `price()` view scoped to one market and one denomination. `YBPriceProxy` can be used for that.

`YBLendingOracle.create_oracles(market_id)` deploys two EIP-1167 minimal-proxy clones of the `YBPriceProxy` implementation via `create_minimal_proxy_to`, one for USD and one for the underlying asset. `create_oracles()` is permissionless. If oracles already exist for that market, it returns the existing addresses.

A clone stores only `(oracle, lt, in_usd)` and holds no pricing logic. `price()` `staticcall`s the bound `YBLendingOracle` singleton's `price_in_usd(lt, False)` or `price_in_asset(lt, False)`, always with `use_balances=False`. A consumer reading a proxy's `price()` cannot select the balance-based fallback. That path can only be reached by calling `YBLendingOracle.price_in_asset()` or `price_in_usd()` directly.

3.5 Deployed Instances (Mainnet)

Both contracts are already live on Ethereum mainnet, wired to a real market:

- `YBLendingOracle` singleton: `0x58241c11abd0bdb1448ef9f38f8aa7fda21a3a44`
- `YBPriceProxy` implementation: `0x7b66d1d70645d22a015a12438d42b2aefc255d28`
- `wbtc.oracles.yieldbasis.eth` resolves to `0x881511e11b3fc4179A841B0202b4D0c3f3F11B2E`, an EIP-1167 clone bound to Factory market #7, whose `asset_token` is WBTC (`0x2260FAC5E5542a773Aa44fBCfeDf7C193bc2C599`), with `in_usd = False`
- The sibling USD-denominated clone for the same market is `0x0e0baa1b3c3ca2f9555a97b68098b56e36f7a41a`

Both contracts are verified on Etherscan as exact bytecode matches. Their source is identical to the reviewed files, except for docstring and license comments: `YBPriceProxy`'s deployed `@license` line differs from the local repo, and `YBLendingOracle`'s deployed copy omits its top-of-file docstring entirely. No functional code differs.

4 Trust Model

The following roles are present in the system:

- Factory Admin (`Factory.admin`)
  - Creates new markets via `add_market()`, deploying AMM/LT/oracle/staker from blueprints
  - Can kill and unkill `LT` contracts via `set_killed()`
  - Can set `AMM` exchange fee and borrowing rate
  - Can change stablecoin allocation given to `AMM` for borrowing
  - Can donate fees with a high discount rate
  - Can change the price aggregator (`set_agg()`), affecting all existing markets
  - Can change blueprint implementations (`set_implementations()`), affecting future deployments
  - Can set the fee receiver address
  - Can set `min_admin_fee` up to 100%
  - Can set the flash lender implementation
  - Can initialize `gauge_controller` if not set during deployment
  - `LT.admin` is set to the Factory during `add_market()`. The `_check_admin()` function accepts both the Factory address and `Factory.admin()`, so Factory Admin has direct access to all LT admin functions.
  - Trust Level: Fully trusted
- Emergency Admin (`Factory.emergency_admin`)
  - Can kill and unkill `LT` contracts via `set_killed()`
  - In killed state, can force `emergency_withdraw()` on behalf of other users from both LT and LiquidityGauge (funds are sent to the token owner)
  - Cannot call any other admin functions (rate, fee, allocations, implementations)
  - Trust Level: Partially trusted
- Fee Receiver (`Factory.fee_receiver`)
  - Receives all admin fees as minted LT tokens when `withdraw_admin_fees()` is called
  - Passive recipient, set by Factory Admin
  - Trust Level: Fully trusted
- Staker (LiquidityGauge)
  - Deployed from blueprint during `add_market()`, set once via `LT.set_staker()`
  - Holds staked LT tokens with special rebase and token reduction mechanics
  - Can call `checkpoint_staker_rebase()` on LT
  - Trust Level: Fully trusted (protocol contract)
- Gauge Controller Owner
  - Can add and kill gauges, directing YB token emissions
  - Trust Level: Fully trusted

- Users
  - Can deposit assets, withdraw assets, and perform exchanges
  - Can trade the underlying cryptopool, seek arbitrage, stake and unstake
  - Several functions are permissionless: `withdraw_admin_fees()`, `distribute_borrower_fees()` (default discount), and `allocate_stablecoins()` (up to existing limit) are callable by anyone
  - Trust Level: Untrusted

4.1 External Trusted Components

- **Curve cryptopools**: Trusted for price data (`price_scale()`, `virtual_price()`) and liquidity operations (`add_liquidity()`, `remove_liquidity_fixed_out()`). LT grants unlimited token approvals to its cryptopool.
- **Price aggregator (`agg`)**: Returns crvUSD/USD price. Validated at deployment to return between 0.9 and 1.1. Affects all value calculations across AMM, LT, and YBLendingOracle. Can be changed by Factory Admin for all existing markets.
- **CryptopoolLPOracle**: Computes LP token price using `price_scale` (state price, not EMA). Called by AMM for trade validation and by LT for value calculations. No access control.
- **Flash lender (`Factory.flash`)**: Used by VirtualPool for flash-loan-assisted swaps. Set by Factory Admin.

4.2 Upgradeability

Contracts are not upgradeable. They are deployed from blueprints via `create_from_blueprint` and are immutable once deployed. However, Factory Admin can change blueprint implementations for future deployments and swap the aggregator and flash lender globally, affecting existing markets.

5 System Considerations

We leverage this section to highlight findings that are not necessarily issues. The mentioned topics serve to clarify or support the report, but do not require a modification inside the project. Instead, they should raise awareness in order to improve the overall understanding for users and developers.

SC1 AMM Zone Transitions and Relaxed Safety Checks

System Consideration | Version 1

The `AMM.exchange()` function divides the system state space into five zones based on the `debt / collateral_value` (d/cv) ratio:

```
A          B                  C                D                E
|          |                  |                |                |
0        1/16               1/2             8.5/16            9/16
       MIN_SAFE          EQUILIBRIUM       MAX_SAFE         CRITICAL
```

The range `B ≤ d/cv ≤ D` defines the allowed post-state for operations with strict safety checks:

- `exchange()` uses strict checks when the trade does not improve the ratio toward equilibrium
- `_deposit()` always uses strict checks
- `value_change(..., is_deposit=True)` uses strict checks for deposit previews

Zone E (`d/cv ≥ 9/16`) causes the quadratic discriminant in `get_x0()` to become negative, resulting in a revert.

New `AMM` relaxed safety check logic in `exchange()` works as follows:

- Below equilibrium `d/cv < 1/2`, relax checks when `d/cv` increased (moved toward equilibrium)
- Above equilibrium `cv/d ≥ 1/2`, relax checks when `d/cv` decreased (moved toward equilibrium)

When strict mode applies, `get_x0()` enforces same [B, D] bounds as before. When relaxed mode applies, as long as state goes in the direction of equilibrium — no checks are performed.

The `_withdraw()` function performs no safety checks on the AMM side.

Zone Transition Analysis

Crossing from zone [B, D] into zone [D, E] (beyond `MAX_SAFE = 8.5/16`) triggers strict checks for any trade that does not improve the ratio. Once in [D, E], the system can only execute trades that decrease `d/cv` (move back toward equilibrium). Before that, the system required one big trade that would immediately bring the system back to [B, D]. Now — smaller trades are allowed to stay within [D, E] zone as long as ratio improves.

SC2 No Staleness Check in YBLendingOracle

System Consideration | Version 1

`YBLendingOracle._price()` does not check `pool.last_timestamp()` against any staleness threshold. Not all inputs are equally stale. Two self-update on every read:

- `pool.price_oracle()` applies EMA decay toward `last_prices` based on elapsed time

- `amm.get_debt()` accrues interest via `rate_mul` on every read

The remaining inputs are storage reads that only update during specific interactions:

Updated by Curve pool interactions (swaps, add/remove liquidity, which trigger `tweak_price()`):

- `pool.price_scale()`, `pool.virtual_price()`

Updated by LT state-changing operations (deposit, withdraw, transfer involving staker):

- `lt.liquidity()`, `lt.totalSupply()`

Updated by AMM exchange/deposit/withdraw:

- `amm.collateral_amount`

AMM trades are not fully independent from the Curve pool: `AMM.exchange()` calls `LT.distribute_borrower_fees()`, which calls `CRYPTOPOOL.add_liquidity()` with accrued interest fees. This triggers `tweak_price()`, refreshing `price_scale`, `virtual_price`, and `last_timestamp`. However, Curve pool swaps do not update AMM or LT state. Neither AMM exchanges nor Curve pool swaps update `lt.liquidity()` or `lt.totalSupply()`. The staleness of these LT stored values causes systematic price deviation, see YBLendingOracle: Stale Accounting Causes Systematic Price Deviation.

When neither the Curve pool nor the AMM has been active, `lp_price_oracle = portfolio_value * D / supply` still partially updates because `portfolio_value` depends on `price_oracle` (live EMA). `lp_price_ps = 2 * vprice * sqrt(price_scale)` is entirely frozen. The ratio `lp_price_oracle / lp_price_ps` used in the normal branch therefore drifts over time, driven solely by the EMA component.

The `_A_at_last_timestamp()` call is not a staleness guard. It evaluates the pool's A parameter at `last_timestamp` rather than `block.timestamp` so that `liboracle._portfolio_value(A, ...)` remains consistent with the cached `D` computed at that same A value. Without this, an ongoing A ramp (e.g., governance changing A from 100 to 200 over 3 days) would cause a mismatch if the pool goes inactive mid-ramp: `D` was computed with `A = 150` but the oracle would evaluate `_portfolio_value` with `A = 180`.

Protocols integrating `YBLendingOracle` must implement their own staleness checks (e.g., comparing `pool.last_timestamp()` against a maximum age) before using the returned price for collateral valuation or liquidation decisions.

SC3 System-Wide crvUSD = USD Equivalence Assumption

System Consideration **Version 1**

The system treats crvUSD 1:1 to USD throughout its accounting. This originates in `AMM.get_x0()`, where collateral is valued in USD, but debt remains in raw crvUSD units. `coll_value` is USD-denominated, while `debt` is in crvUSD. Every downstream consumer inherits this assumption:

- `AMM.value_oracle()`
- `AMM.exchange()`
- `LT._calculate_values()`
- `YBLendingOracle._price()` the normal case uses `x0` from `AMM.get_state()` (same `get_x0`)

`YBLendingOracle` fallback branch explicitly computes `collateral * lp_price_crvUSD * agg_price - debt_crvUSD`. This is equivalent to `get_x0()` 1:1 crvUSD to USD treatment.

Because all components share the same simplification, consistency remains. There is no internal arbitrage between the AMM, LT share pricing, and the oracle. The deviation only exists relative to true USD valuations outside the system.

If crvUSD trades at price `d` in USD, the error is `debt * (1 - d)` everywhere.

When crvUSD < $1, the system subtracts more than the true USD debt, undervaluing net positions. This direction is conservative for lending (triggers liquidations earlier).

When crvUSD > $1, it subtracts less, overvaluing them.

SC4 YBLendingOracle Integrator Considerations

System Consideration | Version 1

`YBLendingOracle` is a read-only pricing view intended for lending markets that list `LT` shares as collateral. Several behaviours matter to a consumer.

Zero returns are actionable

`price_in_asset` and `price_in_usd` return `0` in two stressed states rather than reverting.

1. The success branch when `factor = isqrt(10**36 * lp_price_oracle // lp_price_ps) * (2*L) // (2*L - 1) <= 10**18`
2. Fallback branch (when `AMM.get_state()` reverts)

Consumers must treat `price == 0` correctly, and not as "no data" or "skip".

Rounding directions

Every `//` and `isqrt` in the two computations rounds toward zero, so both `factor` and `coll_value` sit at or below their mathematical value. The two boundaries are therefore marginally outward: the oracle can return `0` for a small set of inputs whose true value is a few wei above the boundary. This is the safer direction for a lending market.

6 Terminology

For the purpose of this assessment, we adopt the following terminology. To classify the severity of our findings, we determine the likelihood and impact (according to the CVSS risk rating methodology).

- **Likelihood** represents the likelihood of a finding to be triggered or exploited in practice
- **Impact** specifies the technical and business-related consequences of a finding
- **Severity** is derived based on the likelihood and the impact

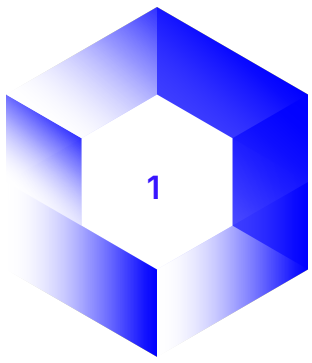
We categorize the findings into four distinct categories, depending on their severity. These severities are derived from the likelihood and the impact using the following table, following a standard risk assessment procedure.

Likelihood	**Impact**		
	High	Medium	Low
High	**Critical**	**High**	**Medium**
Medium	**High**	**Medium**	**Low**
Low	**Medium**	**Low**	**Low**

As seen in the table above, findings that have both a high likelihood and a high impact are classified as critical. Intuitively, such findings are likely to be triggered and cause significant disruption. Overall, the severity correlates with the associated risk. However, every finding's risk should always be closely checked, regardless of severity.

7 Findings

In this section, we describe the findings identified during the course of the engagement. The findings are categorized by severity as explained in the Terminology section. Below we provide an overview of the total number of findings per severity, as well as the number of open issues.



Open Issues

Total number of findings

Critical | High | Medium | Low | Informational

7.1 Overview

The following table lists all identified findings along with their severity and current status. A finding is considered resolved once it has been fully corrected or the specification has been changed accordingly. Non-informational findings with any other status, including partial fixes, acknowledgements, and accepted risks, remain open.

Medium	#001	Low Gas Attack on `get_state()` Call	Code Corrected	✓
Medium	#002	YBLendingOracle Reverts on Price Oracle Divergence	Code Corrected	✓
Low	#003	Hardcoded Equilibrium Threshold Assumes Leverage of 2	Acknowledged	⊖
Low	#004	Oracle Fallback Reverts on Insolvency Instead of Returning Zero	Code Corrected	✓
Low	#005	YBLendingOracle: Stale Accounting Causes Systematic Price Deviation	Code Corrected	✓
Low	#009	Gas-Ratio OOG Guard Insufficient at Deeper Call Depths	Code Corrected	✓
Info	#006	Low Gas Attack on `value_oracle()` Call	Acknowledged	⊖
Info	#007	Missing Reentrancy Check in YBLendingOracle	Code Corrected	✓
Info	#008	Unnecessary `get_state()` Call in YBLendingOracle	Code Corrected	✓

7.2 Descriptions

#001 Low Gas Attack on `get_state()` Call

Security	Medium	Version 1	Code Corrected

If `get_state()` reverts, `YBLendingOracle._price()` falls back to estimating the value of the LT token using the raw balances. This is done under the assumption that the subtraction in `get_x0()` reverted and caused the error. This code path can also be reached if `get_state()` runs out of gas, however, and there's no reliable way to distinguish the two cases.

The consumer of `YBLendingOracle` might be potentially vulnerable to this problem. However, without the exact code of such a price consumer, risk cannot be estimated.

Code Corrected

In **Version 2**, `YBLendingOracle._price()` captures `gas_before = msg.gas` immediately before the `raw_call` to `get_state()`. When the call fails, the function checks `assert msg.gas > gas_before // 16, "GAS"`. An imbalance revert should leave most of `gas_before` in the calling frame, so the guard passes and the balance fallback proceeds. A 63/64-rule OOG consumes the full forwarded gas and leaves only `gas_before / 64`, so the guard fails and the whole call reverts. 1/16 is a tighter ratio of `msg.gas` to `gas_before`, so it remains valid under future global gas repricing.

The guard is inside the `if not use_balances:` block introduced for Unnecessary `get_state()` Call in YBLendingOracle, so a caller explicitly requesting the balance path is unaffected.

#002 YBLendingOracle Reverts on Price Oracle Divergence

Correctness	Medium	Version 1	Code Corrected

In `YBLendingOracle.vy`, the `_price()` function computes an LP oracle value using the expression:

```
isqrt(10**36 * lp_price_oracle // lp_price_ps) * (2 * L) // (2 * L - 1) - 10**18
```

With `L = 2`, this simplifies to `isqrt(...) * 4 // 3 - 10**18`. The subtraction underflows via Vyper's built-in overflow protection when `lp_price_oracle / lp_price_ps < (3/4)^2 = 9/16`.

This condition is distinct from the AMM's own D-computation revert in `get_x0()`. During a price crash, `virtual_price` can crash. `price_oracle` can converge to it over some time, while `price_scale` stays high. Consequently, `get_state()` can succeed (no `AMM.get_x0()` revert) while the `YBLendingOracle`'s isqrt expression underflows.

When triggered, `price_in_asset()` and `price_in_usd()` revert entirely. This can happen during a market crash — precisely the scenario where oracle availability is most critical for liquidations and collateral checks. The `use_balances=True` fallback bypasses the vulnerable code path but is not the default.

Code Corrected

In **Version 2**, the `get_state()` success branch of `YBLendingOracle._price()` no longer subtracts `10**18` from a value that may be smaller. The factor `(2*L)/(2*L-1) * sqrt(lp_price_oracle / lp_price_ps)` is computed first, and `yb_oracle` is only assigned when the factor exceeds `10**18`. Otherwise `yb_oracle` stays at its initial `0`.

The complementary fix for the balance fallback branch is covered by Oracle Fallback Reverts on Insolvency Instead of Returning Zero.

#003 Hardcoded Equilibrium Threshold Assumes Leverage of 2

Correctness	Low	Version 1	Acknowledged

In `AMM.exchange()`, the relaxed safety check logic uses a hardcoded threshold of `2 * 10**18` to determine whether the system is above or below equilibrium. This value assumes `LEVERAGE = 2 * 10**18`, but the contract accepts arbitrary leverage values greater than `10**18` at deployment.

```
if coll_vs_debt_after > 2 * 10**18:
    if coll_vs_debt_before > coll_vs_debt_after:
        # We improved -> relax the check
        check_state = False
else:
    if coll_vs_debt_before < coll_vs_debt_after:
        # We improved -> relax the check
        check_state = False
```

The equilibrium ratio is `collateral_value / debt = LEVERAGE / (LEVERAGE - 10**18)`, not always 2. For leverage values other than 2, the hardcoded threshold causes the relaxed check logic to incorrectly classify movements to equilibrium between good and bad. This may cause trades to be blocked unnecessarily when they should be relaxed, or vice versa.

Acknowledged

In **Version 2**, no expression is generalized but the assumption is now documented as a comment in `AMM.vy`. A second comment block at the threshold itself flags that the `2 * 10**18` value is the equilibrium collateral/debt ratio `LEVERAGE / (LEVERAGE - 10**18)` and equals 2 only because of the system-wide leverage assumption.

Yield Basis confirms that deploying an AMM with a different leverage is not reachable through the Factory (the only supported deployment path) and that any future change of leverage would require generalizing both `AMM.exchange()` and `YBLendingOracle`, so the local generalization is deliberately deferred.

#004 Oracle Fallback Reverts on Insolvency Instead of Returning Zero

Correctness	Low	Version 1	Code Corrected

In `YBLendingOracle._price()`, the fallback branch computes:

```
yb_oracle = collateral * lp_price_oracle // 10**18 * agg_price // 10**18 - debt
```

When the oracle-computed collateral value is less than `debt`, the subtraction underflows and the call reverts.

The fallback is entered when `AMM.get_state()` reverts, which happens when the quadratic discriminant in `get_x0()` underflows at `debt > 9/16 * coll_value_spot`. The fallback uses `lp_price_oracle`. Because the EMA lags behind spot during a crash, the fallback's effective collateral value is higher than the spot-based value and handles the common case of temporary distress correctly.

However, if the position becomes genuinely insolvent at oracle prices (collateral value at EMA price < debt), the fallback also reverts. Both the primary path and the fallback become unavailable simultaneously. The oracle should return zero for an insolvent position. Integrators (e.g., lending protocol liquidation logic) may depend on continuous price availability.

Code Corrected

In **Version 2**, `YBLendingOracle._price()` no longer subtracts `debt` from `coll_value` unconditionally. `coll_value` is computed first, the subtraction is only performed when `coll_value > debt`, and `yb_oracle` stays at its initial `0` otherwise. An insolvent position now reports `0` rather than reverting, leaving the liquidation path of a lending integrator usable.

The matching success-branch fix is covered by YBLendingOracle Reverts on Price Oracle Divergence.

#005 YBLendingOracle: Stale Accounting Causes Systematic Price Deviation

Correctness	Low	Version 1	Code Corrected

`YBLendingOracle._price()` reads stored `lt.liquidity()` and `lt.totalSupply()` to compute per-token price. These stored values are only updated during LT state changing functions (deposit, withdraw, or transfer involving a staker). Between checkpoints, the effective token supply and admin fee balance drift from the stored values due to `_calculate_values()` logic that the oracle does not replicate. Two effects happen:

1. Admin fee undercount (pushes reported LT price up)
2. Missing supply burn (pushes price down)

As a result, divergence occurs, whose direction depends on the staking ratio. At low staking ratios the oracle slightly overestimates. Above a low staking threshold the supply burn effect dominates and the oracle underestimates, with the error scaling with both staking ratio and accumulated trading fees.

Numerical example

After 20 round-trip trades of $20k each$ (5.6k in fees), the oracle diverges from `pricePerShare()` as follows:

Staking %	Net divergence	Direction
0%	0.054%	OVER
50%	0.147%	UNDER
90%	0.989%	UNDER
99%	4.151%	UNDER

Divergence grows roughly linearly with cumulative fee income. In an extreme scenario (99% staked, 50 round-trips of $20k), the stored supply exceeds the effective supply by 10.65%, causing a 9.57% oracle underestimation.

A checkpoint (any deposit, withdrawal, or staker transfer) eliminates the divergence.

For lending protocols using `YBLendingOracle` for collateral pricing, the dominant underestimation direction is conservative. Borrowers can borrow less than the true collateral value, reducing undercollateralization risk but causing capital efficiency loss. The slight overestimation at 0% staking (~0.05%) is in the unsafe direction but negligible in magnitude. Using `LT` as borrowable asset is dangerous and should be done with a very conservative LTV setting.

Code Corrected

In **Version 2**, `YBLendingOracle._price()` no longer reads stale `lt.liquidity()` and `lt.totalSupply()` in the normal path. A new internal helper `_calculate_fresh_lv()` replicates

> `LT._calculate_values()` to compute the up-to-date `total`, `admin`, and effective supply from the current AMM value, then uses those values for per-token scaling. The cached `lt.liquidity()` and `lt.totalSupply()` are still used in the imbalanced fallback branch (`use_balances` / `get_state()` failure), where `LT` state is frozen.

#009 Gas-Ratio OOG Guard Insufficient at Deeper Call Depths

Security	Low	Version 2	Code Corrected

Low Gas Attack on `get_state()` Call (**Version 2**) added a guard so that `YBLendingOracle._price()` can tell a genuine "AMM too imbalanced" revert from `AMM.get_state()` apart from an attacker-forced out-of-gas (OOG):

```
gas_before: uint256 = msg.gas
success, response = raw_call(
    amm.address, method_id("get_state()"),
    max_outsize=96, revert_on_failure=False, is_static_call=True)
if not success:
    assert msg.gas > gas_before // 16, "GAS"
```

The 1/16 threshold is derived from the EIP-150 63/64 rule for a *single* call boundary: a forced OOG at the immediate callee retains at most `gas_before / 64`, easily under `gas_before / 16`. This holds for one call hop.

However, `get_state()` calls `PRICE_ORACLE_CONTRACT.price()`, which itself calls `AGG.price()`. Each additional call forwards 63/64 of its gas, and keeps 1/64 reserve regardless of what happens deeper in the stack. If an attacker engineers the OOG to land at the deepest frame of a multi-hop chain, the gas left over at the `YBLendingOracle` level is the sum of every intervening frame's untouched reserve, not a single `gas_before / 64`.

Worst-case leftover gas after an OOG 5 calls deep is bounded by:

$$\frac{1}{64} + \frac{63}{64^2} + \frac{63^2}{64^3} + \frac{63^3}{64^4} + \frac{63^4}{64^5} \approx 0.0757 \cdot gas\_before > \underbrace{gas\_before / 16}_{(0.0625 \cdot gas\_before)}$$

i.e. past 5 nested hops, a forced OOG can leave more than `gas_before / 16` gas behind, which the guard misclassifies as a genuine revert rather than an OOG.

At the specific depth `YBLendingOracle.vy` reaches today (`get_state()` → `PRICE_ORACLE_CONTRACT.price()` → `AGG.price()`), the guard still holds. However, a deeper call chain or a future gas-repricing EIP could defeat this guard.

If the OOG is misclassified, an attacker can force `_price()` down the balance-based fallback branch on demand, receiving the fallback's differently computed price even when the AMM is not actually imbalanced.

> **Code Corrected**
>
> In **Version 3**, `YBLendingOracle.vy` drops the `raw_call` to `get_state()` and the gas-ratio guard entirely. `AMM.get_x0` is reproduced in-contract as a `YBLendingOracle._get_x0(p_oracle, collateral, debt) -> (solvable, x0)`, identical to `AMM.get_x0(..., safe_limits=False)`. This `_get_x0()` returns `solvable=False` instead of reverting on the discriminant underflow.

#006 Low Gas Attack on `value_oracle()` Call

Informational	Version 1	Acknowledged

In `emergency_withdraw()`, the failure of a call to `value_oracle()` is handled by falling back to proportional withdrawal, as if the LT had been killed by the admin. If this occurs, only 1/64 of the gas will be available for the fallback path. In the current state of the code, an exploit could not be found, but future evolutions might cause the issue to manifest.

Acknowledged

In **Version 2**, no code change is applied. Yield Basis reports a measured margin of `cost(value_oracle) > 63 * cost(rest-of-emergency_withdraw)` of `36.5k` gas versus a required `9.9M` gas, which means forcing the proportional path through gas starvation would require `value_oracle()` to become roughly `270x` more expensive than it is. The remainder of `emergency_withdraw` (the `amm._withdraw` call, the `CRYPTOPOOL.remove_liquidity` call, several token transfers, and `_burn`) is dominated by state-access costs that the upcoming Glamsterdam repricing (EIP-7904) leaves untouched while making `value_oracle()` itself relatively cheaper, so the margin is expected to widen rather than shrink.

A starved call additionally cannot return a manipulated value through this entry point. The path is reached from a state-changing function, so an OOG aborts the whole transaction rather than silently flipping a returned value. The `LT._checkpoint_gauge` precedent of asserting `msg.gas >= 200_000` was not reused because, at this margin, an absolute floor would either be permanently slack or mis-calibrate across a future gas repricing.

#007 Missing Reentrancy Check in YBLendingOracle

Informational	Version 1	Code Corrected

`YBLendingOracle._price()` reads state from the AMM, LT, and cryptopool contracts via `staticcall` without verifying that none of these contracts are mid-execution. The AMM exposes a `check_nonreentrant()` view function that reverts if the AMM's reentrancy lock is held, and LT itself calls this function before sensitive operations such as `allocate_stablecoins()` and `withdraw_admin_fees()`.

The oracle does not perform this check. If `_price()` were called during a callback within an AMM operation, it could read intermediate state - for example, partially updated collateral or debt values - producing an incorrect price.

In the current codebase, no code path invokes the oracle mid-reentrancy, assuming all tokens handled by the system (crvUSD, Curve LP tokens, and asset tokens such as WBTC) are plain ERC-20 without transfer hooks. So token transfers in `AMM.exchange()` and `LT.withdraw()` never yield execution control to the recipient. All external calls target trusted, hardcoded contracts (price oracles, Curve pools, gauge controllers), and market creation via `Factory.add_market()` is admin-gated, preventing introduction of callback-capable tokens. The absence of the check leaves the oracle unprotected against read-only reentrancy if future integrations or token allowlisting changes introduce a callback path.

Code Corrected

In **Version 2**, `YBLendingOracle._price()` checks the AMM's reentrancy lock before reading any AMM, LT, or pool state via `AMM.check_nonreentrant()`.

The pool is safe because `pool.price_oracle()` and `pool.price_scale()` are themselves `@nonreentrant @view` on the Twocrypto pool. Together these two checks cover the AMM-callback vector and the `cryptopool.remove_liquidity` callback vector.

#008 Unnecessary `get_state()` Call in YBLendingOracle

Informational | **Version 1** | **Code Corrected**

In `YBLendingOracle._price()`, `get_state()` is called via `raw_call` on every invocation regardless of the `use_balances` flag. When `use_balances` is true, the response is never used because the if condition requires both `success` and `not use_balances`. The `raw_call` to `get_state()` triggers `get_x0()` computation and multiple storage reads in the AMM, wasting gas when the caller already knows it wants the balance-based fallback path.

Code Corrected

In **Version 2**, the `raw_call` to `get_state()` is gated behind `if not use_balances:` (`YBLendingOracle.vy:192-200`). When the caller passes `use_balances=True`, the call is skipped entirely, `success` stays `False`, and the function falls through to the balance branch. Yield Basis reports a measured saving of `~43k` gas on the `use_balances=True` path (`~85k` versus `~128k` for the x0 path).

The gas-snipe guard introduced for Low Gas Attack on `get_state()` Call is inside the same `if not use_balances:` block, so it only runs when `get_state()` is actually attempted.

8 Limitations and use of report

Security assessments cannot uncover all existing vulnerabilities; even an assessment in which no vulnerabilities are found is not a guarantee of a secure system. However, code assessments enable the discovery of vulnerabilities that were overlooked during development and areas where additional security measures are necessary. In most cases, applications are either fully protected against a certain type of attack, or they are completely unprotected against it. Some of the issues may affect the entire application, while some lack protection only in certain areas. This is why we carry out a source code assessment aimed at determining all locations that need to be fixed. Within the customer-determined time frame, ChainSecurity has performed an assessment in order to discover as many vulnerabilities as possible.

The focus of our assessment was limited to the code parts defined in the engagement letter. We assessed whether the project follows the provided specifications. These assessments are based on the provided threat model and trust assumptions. We draw attention to the fact that due to inherent limitations in any software development process and software product, an inherent risk exists that even major failures or malfunctions can remain undetected. Further uncertainties exist in any software product or application used during the development, which itself cannot be free from any error or failures. These preconditions can have an impact on the system's code and/or functions and/or operation. We did not assess the underlying third-party infrastructure which adds further inherent risks as we rely on the correct execution of the included third-party technology stack itself. Report readers should also take into account that over the life cycle of any software, changes to the product itself or to the environment in which it is operated can have an impact leading to operational behaviors other than those initially determined in the business specification.