CHAINSECURITY

PUBLIC

Code Assessment of the LP Oracle Smart Contracts

October 16, 2026

Produced for

Curve



Contents

1	Executive Summary	3
2	Assessment Overview	4
3	System Overview	6
4	Trust Model	9
5	System Considerations	11
6	Terminology	12
7	Findings	13
8	Limitations and use of report	23

1 Executive Summary

Dear Curve team,

Thank you for trusting us to help Curve with this security audit. This executive summary gives an overview
of the subjects covered in our audit of the latest reviewed contracts of LP Oracle according to **Scope**.

Curve implements LP oracle contracts for 2-coin Curve pools. The codebase is organized around a shared StableSwap solver using Newton iteration with a bisection fallback, with thin wrappers for Twocrypto-with-stableswap and StableSwap-NG pools that compute portfolio values and LP token prices for external integrators.

We focused primarily on pricing correctness, manipulation resistance, and the assumptions the oracle contracts inherit from their underlying pools. The primary concerns are edge cases when using the oracle to price LP tokens as collateral in potentially large lending markets.

In `curve-std` **Version 5**, the Curve team improved the solver's initial guess, removed redundant calculations, and expanded the supported amplification range to cover StableSwap-NG amplification ramps. These changes correct the reported gas regressions and amplification-related quote failures. Our review of these changes identified no additional correctness or security findings.

The contracts are stateless and carry no privileged roles, keeping the administrative attack surface minimal.

Security regarding pricing correctness and manipulation resistance remains improvable. The Curve team accepted the risks described in **Oracle Not Suitable for Low-Liquidity Pools**, **EMA Price Cap Propagates to the LP Oracles, Mispricing the LP Token After a Token Crash**, and **Collateral Value Jumps**. The solver improvements do not remove these limitations, which arise from the underlying pools and their oracle and rebalancing mechanisms. The reviewed implementation retains the underlying EMA cap limitation, and integrating protocols must account for the resulting valuation risk. The pricing approach is mathematically well-founded, but the oracle output should not be treated as a universally conservative LP valuation without appropriate controls. Lending markets must size their LTV, liquidation thresholds, and supply caps accordingly.

Integrators must also account for possible atomic increases in LP valuation, inconsistent historical amplification on idle pools, and quote failures when StableSwap-NG oracle inputs leave the supported price range. Appropriate pool selection, monitoring, and conservative lending parameters remain necessary. Liquidity checks must consider the persistence of a biased EMA after liquidity is added.

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, the issues uncovered, and how they have been addressed. We are happy to receive questions and feedback to improve our service.

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 LP Oracle repositories based on the documentation files. The table below indicates the code versions relevant to this report and when they were received.

Curve Std

Version	Date	Commit Hash	Note
1	12 March 2026	09ad21756cd573cd6ac7afb32fb299fef32429cc	Version 1
2	21 May 2026	949ae6c0a2c3dfb9e6808aa6d2ca9b737a5afd59	Version 2
3	17 August 2026	048cb23d0ed4c48768815ea5c46d4a676da8de35	Version 3
4	17 September 2026	60dbf5029d2e586f24f232b87e06df729eff407b	Version 4
5	30 September 2026	eedf8c022af0904ff474ca00c9a7d02c4c86eaf4	Version 5
6	8 October 2026	68a13eb2ee80bd704fc259a3aed0a6ce53d59b72	Version 6

Curve Twocrypto-ng

Version	Date	Commit Hash	Note
1	7 May 2026	4fefc2ccc63ab1b3dc3b084962d792dbd5c2fea3	Version 1
2	27 May 2026	ef4ed70295473badc2953875172cb744ec43c102	Version 2
3	17 August 2026	ec54c8148207a12bbb0d8291173d03007a9a5259	Version 3
4	8 October 2026	6e36b876e64b99fafff4c4e1c73c8bafb6fea0f2	Version 4

Curve Stableswap-ng

Version	Date	Commit Hash	Note
1	18 April 2026	6bee7bcee15a9bc577f747e61aab837b8529a0dc	Version 1
2	27 May 2026	9cd4e19d585b77bf79feca0a3eca9a356229ab6e	Version 2
3	17 August 2026	911b5b45e4edafda96a74ec5f464a673c380456c	Version 3
4	8 October 2026	3332cd44656ec64b3c048885e1dd71955254b262	Version 4

The latest version listed for each repository (Version 6 for `Curve Std`, Version 4 for `Curve Twocrypto-ng` and `Curve Stableswap-ng`) corresponds to the merge of the respective feature branches into `main`, provided to us after the review of the preceding versions had concluded.

For repository `Curve Std`:

```
curve_std/stableswap/lp_oracle_2.vy
```

For repository `Curve Twocrypto-ng`:

```
contracts/main/LPOracle.vy
```

For repository `Curve Stableswap-ng`:

```
contracts/main/StableSwapNGLPOracle.vy
```

2.1.1 Excluded from scope

All other contracts not mentioned in scope.

3 System Overview

This system overview describes the initially received version (**Version 1**) of the contracts as defined in the Assessment Overview.

Curve provides a set of LP oracle contracts for Curve pools. The oracle computes a manipulation-resistant LP token price and portfolio value for 2-coin pools using the StableSwap invariant.

3.1 Architecture Overview

The system consists of three main components:

1. **`lp_oracle_2`**: Core mathematical library implementing the StableSwap invariant inverse problem solver.
2. **`LPOracle`**: Wrapper adapting the solver for 2-coin Twocrypto (FXSwap-style, Twocrypto with Stableswap invariant) pools.
3. **`StableSwapNGLPOracle`**: Wrapper adapting the solver for 2-coin StableSwap-NG stable pools, including pools with yield-bearing assets.

3.1.1 Core Library: `lp_oracle_2`

The library solves an inverse problem: given an amplification parameter `A` and a target marginal price `p`, it finds the pool composition `(x, y)` on the StableSwap invariant where the marginal price equals `p`, then returns the portfolio value $V = x + p \cdot y$ in units of coin 0, with the invariant normalized to `D = WAD = 1e18`.

The solver proceeds as follows:

1. **`_y_from_bisection`**: Locates the pool composition `y` satisfying `p(y) = p_target` via binary search on the monotone branch $y \in (0, D/2]$, converging to within `p_target / 10^6` (0.01 basis points) relative tolerance or a 1-wei bracket.
2. **`_x_from_y`**: Computes `x(y)` from the closed-form quadratic root of the StableSwap invariant.
3. **`_portfolio_value`**: Returns $x + p \cdot y$ / WAD.

For `p < 1`, the library exploits the symmetry of the invariant by solving on the reciprocal branch and mapping the result back.

Input domain constraints enforced by the library: $A_{\text{raw}} \in [1, \text{MAX_A_RAW}]$, p / 0.~~N~~o upper bound is placed on `p`.

3.1.2 Twocrypto-with-Stableswap Wrapper: `LPOracle`

`LPOracle` reads `price_oracle()`, `price_scale()`, `A()`, `D()`, and `totalSupply()` from a Twocrypto-with-stableswap pool and translates them into the canonical form expected by `lp_oracle_2`.

Two adaptations are required:

- **Amplification scaling**: The pool stores `A_pool = A_true * N_COINS^(N_COINS-1) * POOL_A_PRECISION`. The wrapper rescales this to `A_raw = A_true * A_PRECISION`.
- **A at last timestamp**: Because the Twocrypto pool's price oracle may be computed at an earlier timestamp while A is ramping, the wrapper replicates the pool's linear A-ramp formula evaluated at `last_timestamp` to keep the invariant calculation consistent with the oracle snapshot.

`_lp_price()` computes `portfolio_value * D / total_supply`. It uses `pool.D()` - the stored invariant from the last pool interaction - not a freshly recomputed `D` from live balances, making it resistant to same-block donation attacks.

The public interface exposes `portfolio_value(_pool, _i)` and `lp_price(_pool, _i)`, where `_i` $\in \{0, 1\}$ selects the numeraire coin. Both functions return WAD-scaled values in units of the selected coin.

An optional `sanity_check(_pool)` validates that `A`, `virtual_price`, `price_scale`, and `price_oracle` are positive. It is a standalone entry point and is not called by the pricing functions.

3.1.3 StableSwap-NG Wrapper: `StableSwapNGLPOracle`

`StableSwapNGLPOracle` reads `price_oracle(0)`, `A_precise()`, and `get_virtual_price()` from a StableSwap-NG pool.

Amplification scaling differs from the Twocrypto wrapper: the NG pool stores `A_ng = A_true * N_COINS^(N_COINS-1) * 100`, which the wrapper converts to `A_raw = A_ng * A_PRECISION / (N_COINS^(N_COINS-1) * 100)`.

`_lp_price()` computes `portfolio_value * get_virtual_price() / PRECISION`. Unlike the Twocrypto wrapper, `get_virtual_price()` recomputes `D` from live pool balances. For pools containing rebasing tokens (`asset_types` type 2), those balances are read via live `balanceOf` calls, making the result sensitive to same-block token donations.

The public interface mirrors that of `LPOracle`. For pools containing yield-bearing assets (wstETH, sDAI, ERC-4626 tokens), the oracle returns values denominated in the **base asset** of the selected coin (e.g. stETH for wstETH, DAI for sDAI), not in the token itself. Callers that need a token-denominated price must apply the pool's `stored_rates`.

An optional `sanity_check(_pool)` validates `virtual_price`, `price_oracle(0)`, the `A` bounds, and the 2-coin constraint. The 2-coin check is implemented by calling `coins(2)` and asserting the call reverts.

3.1.4 Supported Pool Types

The oracle is designed for plain 2-coin Curve pools:

- **Twocrypto-with-Stableswap / FXSwap** (volatile, 2-coin) via `LPOracle`.
- **StableSwap-NG** (stable, 2-coin) via `StableSwapNGLPOracle`

The following pool types are outside the supported set:

- Pools with more than 2 coins.
- Pools containing rebasing tokens (asset type 2), for which `get_virtual_price()` reads live `balanceOf` and is sensitive to donations.
- Older implementations of Stableswap pools (not Stableswap-NG).
- Twocrypto pools with the CryptoSwap invariant.

3.2 Changes in Version 4

The binary search is replaced by a Newton iteration with a bisection fallback.

The solver keeps a bracket $[lo, hi]$ containing the root, initially $[1, D/2]$. Each iteration evaluates $p(y)$ at the current candidate and narrows the bracket on the side given by the sign of $p(y) - p_{\text{target}}$. The exit conditions and the tolerance are unchanged.

The next candidate is the Newton step $y_{k+1} = y_k + (p(y_k) - p_{\text{target}})/|p'(y_k)|$. A candidate outside the bracket is replaced by the bracket midpoint.

The rate of change of the marginal price, $p'(y)$, is computed from a closed-form expression in x, y and p. It is evaluated only when $p(y)$ at the candidate is below $500 \cdot \text{WAD}$. Otherwise the midpoint is used.

The first candidate is $y_0 = 1/(4\sqrt{A_{\text{eff}}(p-1)})$, clamped into the bracket.

At most 16 Newton steps are taken. The solver then bisects for up to 60 further iterations.

3.3 Changes in Version 5

The first candidate now interpolates the constant-product and high-amplification limits. In normalized units, it is $y_0 = 1/\sqrt{4p + 16A_{\text{eff}}(p-1)}$. This improves the starting point at low amplification and gives $y_0 = 1/2$ at parity.

`MAX_A` is increased from `100000` to `500000`, raising `MAX_A_RAW` to `5000000000`. After the StableSwap-NG wrapper's amplification scaling, this covers the full amplification range permitted by the pool's `ramp_A()`.

The solver reuses `x` between the price and derivative evaluations and returns `(x, y)` to avoid recomputing `x` after convergence. The derivative calculation also reuses intermediate products. These changes reduce gas consumption.

4 Trust Model

The oracle contracts are stateless, non-upgradeable view contracts with no owner, no privileged roles, and no stored state beyond immutable constants. There are no administrative functions. The trust surface is entirely external: the pool the oracle is pointed at, and the integrator that deploys and calls it.

4.1 Pool

The oracle places the following assumptions on the pool it is given. None of these are enforced dynamically by the pricing functions and must be called by the integrator if needed.

Checked by `sanity_check()`:

- `A`, `virtual_price` / `price_oracle` / `price_scale` are positive and within the bounds the oracle expects.
- The pool has exactly 2 coins.

Assumed unconditionally by the pricing functions:

- The pool's `price_oracle()` (Twocrypto) or `price_oracle(0)` (StableSwap-NG) is an exponential moving average that lags sufficiently behind spot to be manipulation-resistant within a single block. No staleness check is performed. The oracle inherits the pool's own oracle security model.
- The LP Oracle relies on the underlying pools being efficiently arbitraged: Hence, it is assumed that no updates to the pool mean that the price is still valid.
- For rate-oracle and ERC4626 StableSwap pools (asset types 1 and 3), `get_virtual_price()` queries the external rate contract on every call and scales `D`, and therefore `lp_price`, by the returned value with no smoothing. There is no EMA or circuit breaker on the stored rate. A downward event (slashing, vault loss) causes an immediate markdown. The oracle therefore inherits the full trust assumption of the external rate source.
- The StableSwap pool does not contain rebasing tokens (asset type 2). If it does, `get_virtual_price()` reads live `balanceOf` and the LP price is sensitive to same-block token donations, see LP Oracle Can Be Atomically Manipulated Upwards.
- The pool's amplification parameter `A` is within the range `[POOL_A_PRECISION, MAX_A * POOL_A_PRECISION]`.
- The Stableswap version from commit `6bee7bcee15a9bc577f747e61aab837b8529a0dc` is used and with `POOL_A_PRECISION = 100`.
- The price returned by the pool's `price_oracle()` is within a sane range, defined as $0.01 < p < 100$.

4.2 Integrator

The safety of the oracle is partially delegated to the user:

- `sanity_check()` must be called by the user at deployment time to validate the pool before it is used for pricing. The value-returning functions `portfolio_value()` and `lp_price()` never call it. A user that skips `sanity_check()` and points the oracle at an unsupported pool might receive a revert or an incorrect price with no indication of the failure.
- The user must not assume the oracle's output is conservatively biased in a fixed direction. The rounding direction of `lp_oracle_2._portfolio_value()` is not constant: the result can be over-estimated or

under-estimated. The magnitudes are negligible for pricing when the price is in the sane range, but the sign is not guaranteed.

- The user must select the correct wrapper for the pool type. `LPOracle` and `StableSwapNGLPOracle` are not interchangeable. They expect different pool interfaces and apply different amplification scaling.
- The state and the parameters of the underlying pools and the systems integrating with the oracle must be monitored constantly to well-behave in their respective combinations!

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 `stored_rates` Jumps

System Consideration | Version 1

`lp_price` is the product of an EMA-smoothed portfolio value and `get_virtual_price`, which recomputes pool balances from live `stored_rates` on every call. Any change in the external rate, such as yield accrual, a rate-oracle update, or a direct token transfer to an ERC4626 vault, moves `lp_price` without requiring a pool interaction. Since `get_virtual_price` directly depends on `stored_rates`, only pools with trusted sources of `stored_rates` should be considered.

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

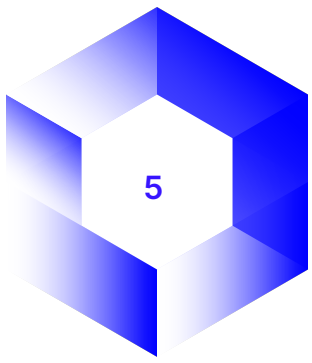
We categorize the findings into four distinct categories, depending on their severity. These severities are derived from the likelihood and the impact using the following table, following a standard risk assessment procedure.

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	Collateral Value Jumps	Risk Accepted	!
Medium	#002	EMA Price Cap Propagates to the LP Oracles, Mispricing the LP Token After a Token Crash	Risk Accepted	!
Medium	#003	Oracle Not Suitable for Low-Liquidity Pools	Risk Accepted	!
Low	#004	`_y_from_bisection` Bracket-Exit Fall-Through Violates the Documented 1e-6 Price Tolerance	Code Corrected	✓
Low	#005	`LPOracle._A_at_last_timestamp` Returns Wrong A After Multiple Ramp Epochs on an Idle Pool	Risk Accepted	!
Low	#006	LP Oracle Can Be Atomically Manipulated Upwards	Risk Accepted	!
Low	#007	SanityCheck for Two-Coins Can Be Bypassed with Low-Gas Call	Code Corrected	✓
Info	#008	Dead Duplicated `_scaled_price` Helper in LPOracle	Acknowledged	−
Info	#009	Incorrect and Incomplete Comments	Code Corrected	✓
Info	#010	LP Oracle Sanity Check A-Bound Is a No-Op	Code Corrected	✓
Info	#011	No Fallback When p Leaves the `[MIN_P, MAX_P]` Safeguard Band	Code Corrected / Acknowledged	✓
Info	#012	Newton Solver Increases Gas at Low Amplification Despite Reducing the Iteration Count	Code Corrected	✓
Info	#013	Pool Amplification Ramps Can Disable Oracle Quotes	Code Corrected	✓
Info	#014	Repeated Calculations Increase Oracle Gas Usage	Code Corrected	✓

7.2 Descriptions

#001 Collateral Value Jumps

Design | Medium | Version 1 | Risk Accepted

If a lending protocol uses this LP oracle as collateral value, it may assume the LP token is priced smoothly and safely. However, rebalance events in Twocrypto (with the StableSwap invariant) can cause jumps in collateral value. A borrower who was healthy before the rebalance may become undercollateralized after it.

The practical impact depends on the drop in LP token value caused by the adjustment step of `price_scale`, and the consuming protocol's risk parameters. The adjustment step size is a configuration parameter of Twocrypto pools that must be chosen carefully to limit the LP price impact of the rebalance. In a lending integration, the relevant question is whether an oracle movement can exceed the borrower's collateral buffer, i.e. the distance between the loan-to-value ratio and the liquidation threshold. A pool with conservative LTV parameters can tolerate larger transient LP-price movements, while a high-LTV market may become sensitive to comparatively small oracle changes.

> **Risk Accepted**
> The Curve team acknowledged the issue as by-design behavior of the rebalance mechanism and accepted the residual risk to integrating protocols. Sizing of the `price_scale` adjustment step and the choice of LTV parameters in any lending integration remain the responsibility of the integrator. This is a central design decision and must be communicated and documented prominently.

#002 EMA Price Cap Propagates to the LP Oracles, Mispricing the LP Token After a Token Crash

Design | Medium | Version 1 | Risk Accepted

`Twocrypto.tweak_price` clamps the spot price fed into the EMA before each update to:

```
# Twocrypto.tweak_price / TwocryptoView._price_oracle
min(max(last_prices, unsafe_div(price_scale, 2)), 2 * price_scale)
```

and `StableSwapNG.upkeep_oracles` applies only an upper cap:

```
# CurveStableSwapNG.upkeep_oracles
min(spot_price[i], 2 * 10**18)
```

When the true market price is outside the clamp band, `price_oracle` converges toward the clamped bound rather than the true price. The effect is reduced only when subsequent pool interactions allow `price_scale` to move and the clamp window to shift. Until then, lending protocols or other integrations that accept LP tokens as collateral should not treat `lp_price()` as a conservative valuation.

In detail, each StableSwap-NG spot sample fed into the EMA is capped at `2 * 10**18`. For Twocrypto, each spot sample is clamped to `[price_scale / 2, 2 * price_scale]`, and the wrapper feeds `p_oracle * PRECISION // p_scale` into the LP oracle. This is not a fixed ceiling because `price_scale` can be rebalanced toward `price_oracle` on qualifying pool interactions, so the bound moves in the right direction. The rebalancing is only conditional and throttled (vector-distance threshold, `adjustment_step`, profitability checks), so `price_scale`, and hence the cap, tracks a fast move only slowly.

`StableSwapNGLPOracle._portfolio_value` and `LPOracle._portfolio_value` pass this value into `lp_oracle_2._portfolio_value(A_raw, p)` as the marginal price `p`. The cap therefore spills over into the

LP oracle. For StableSwap-NG the oracle cannot represent any pool state whose true marginal price is above `2 * 10**18`. For Twocrypto the oracle price lags far behind a true price that has moved much faster than `price_scale` can rebalance.

This is normally harmless because trading keeps the EMA near the traded pool price. It becomes a problem on a catastrophic crash of token0. As token0 collapses, traders dump it into the pool until the pool is almost entirely token0 and the true marginal price of coin1 in coin0 blows up far past 2. From that point the pool is one-sided and illiquid. The oracle price is stuck at `p ~= 2 * 10**18` while the true price is, e.g., `100 * 10**18`.

The mispricing is largest in the coin1 numeraire (`_i = 1`), where the wrapper divides by `p_oracle`. Feeding the capped `p = 2e18` instead of the true `p = 100e18` overstates the coin1 LP value by the following measured factors:

`A`	**coin1 overpricing at `p_true = 100e18`, cap `2e18`**
10	~20x
100	~30x
1000	~40x
100000	~49x

In the coin0 numeraire (`_i = 0`) the measured error over the same range is at most ~2x. The large error is confined to coin1.

This is an inherent limitation of the capped Curve EMA design that the LP oracles inherit. It cannot be addressed in the oracle alone. After a depeg larger than the cap the reported `lp_price` does not reflect the collapsed pool value, with a coin1 error of order `p_true / p_cap`.

> **Risk Accepted**
> The Curve team accepted the risk and documented the cap behavior in the NatSpec of `lp_price()`, explicitly warning integrators that the LP token price returned by the oracle is capped at the underlying `price_oracle`'s cap (`2.0` for StableSwap-NG, `[0.5, 2.0]` relative to `price_scale` for Twocrypto).

#003 Oracle Not Suitable for Low-Liquidity Pools

Design | Medium | Version 1 | Risk Accepted

`StableSwapNGLPOracle` and `LPOracle` should not be used as a price feed for shallow liquidity pools. The EMA underpinning `price_oracle(0)` can be biased at low cost in a thin pool, and a corrupted EMA persists after liquidity is added, making depth at the time of the oracle read an unreliable safety indicator.

EMA manipulation cost scales with pool depth. In a thin pool the capital required to move the spot price by a given amount is low, making a multi-block accumulation of distorted price possibly cheap relative to the oracle's influence over collateral pricing. If the portfolio value is used to e.g. estimate the collateral value of a loan, there is an inherent connection between the size of the loan and the associated risk and cost to manipulate the market respectively the oracle.

A manipulated EMA is not reset by liquidity additions. When large liquidity is deposited after the EMA has been biased, the new corrected spot price is blended with the stored `last_ema_value`, which still carries the manipulated state. The EMA recovers toward the correct price only over a window proportional to `ma_exp_time`. During this recovery period the pool appears deep, but the oracle is still biased. An integrator

that checks depth at read time has no reliable way to distinguish a legitimately deep pool from one that was recently inflated after an EMA manipulation.

> **Risk Accepted**
>
> The Curve team accepted the risk. Restricting the LP oracle to pools with sufficient liquidity, and accounting for the persistence of EMA bias after liquidity is added, is left to the integrating protocol. It must be ensured that this behavior is known and monitored.

#004 `_y_from_bisection` Bracket-Exit Fall-Through Violates the Documented 1e-6 Price Tolerance

Correctness **Low** **Version 1** **Code Corrected**

`_y_from_bisection` documents and is expected to return a `y` whose marginal price is within `1/PRICE_TOL_REL = 1e-6` (relative) of `p_target`. It has two exit paths:

```
if unsafe_sub(pm, p) <= tol_abs:   # price-tolerance exit (intended)
    return mid
...
if unsafe_sub(hi, lo) <= 1:        # bracket fall-through
    return hi
```

The bracket fall-through returns `hi` unconditionally, with no price check. When `p_target` is very large, `p(y)` is steep at the solution: `|dp/dy|` over a single wei of `y` greatly exceeds `p_target / 10**6`. The bracket `[lo, hi]` then collapses to width `<= 1` before the price-tolerance check can ever fire, and the function returns `hi`, whose marginal price can be arbitrarily far from `p_target`. The documented `1e-6` relative bound is not honored in this regime.

This propagates downstream. `_get_x_y` returns the curve point `(x, y)` at this wrong `y`. `_portfolio_value` then computes `x + p_target * y // WAD`, combining a curve point that does *not* have marginal price `p_target` with the `p_target` multiplier. The reported portfolio value can diverge from the true `V(p_target)` by far more than `1e-6`.

The danger is largely theoretical: the prices required to trigger the fall-through are extreme (many orders of magnitude away from `WAD`). Such prices are not expected on a well-funded pool, but could arise on a newly initialized or very low-liquidity pool whose spot price is far from parity.

The realistic range of `A_raw` is roughly `5000` (pool minimum, `A_eff = 0.5`) to `1e9` (`A_eff = 1e5`). The fall-through begins once `p_target` exceeds a threshold that depends on `A_raw`. Below that threshold the bisection converges normally:

`A_raw` (`A_eff`)	bracket exit begins at `p_target` ~	price error at threshold
`5000` (`0.5`, pool min)	`1.3e37`	~ 1e-6
`10000` (`1`)	`3.3e36`	~ 1e-6
`1e6` (`100`)	`3.1e35`	~ 1e-6
`1e9` (`1e5`, max)	`2.6e33`	~ 1e-6

Past the threshold the error grows rapidly. Measured behavior at a fixed `p_target = 1e50` (`WAD = 1e18`), calling `_get_x_y(A_raw, p_target)` then `_portfolio_value(A_raw, p_target)` with realistic `A_raw`. In

both cases the bracket collapses to `y = 2` wei (one wei above `lo = 1`) and the fall-through returns it before the price-tolerance check fires:

- `A_raw = 10000` (`A_eff = 1`): returned point `(x, y) = (1.77e26, 2)`. Its marginal price `p_hat` is $\approx 4.4 \times 10^{43}$, about 2.3×10^{6} times below the requested `1e50`. `_portfolio_value` returns `200000176776695671636881497958653` ($\approx 2.0 \times 10^{32}$) while the true `V(p_target)` is $\approx 3.5 \times 10^{28}$, overstated by $\approx 5.7 \times 10^{3}$ times.
- `A_raw = 1e9` (`A_eff = 1e5`, max): returned point `(x, y) = (5.59e23, 2)`. `p_hat` is $\approx 1.4 \times 10^{41}$, about 7.2×10^{8} times below `1e50`. `_portfolio_value` returns `200000000559017494373921029782009` ($\approx 2.0 \times 10^{32}$) against a true `V` of $\approx 7.5 \times 10^{26}$, overstated by $\approx 2.7 \times 10^{5}$ times.

Code Corrected

The Curve team added explicit input bounds on `p` in `_get_x_y`, restricting it to `MIN_P = 1e16 <= p <= MAX_P = 1e20` (i.e. `0.01 <= p <= 100`). The bracket-fall-through regime only begins around `p ~ 1e33` and above, which is many orders of magnitude beyond the new allowed input range. The price-tolerance fall-through will therefore no longer be reached in normal operation, especially given the underlying `price_oracle` is itself capped at `2.0`.

#005 `LPOracle._A_at_last_timestamp` Returns Wrong `A` After Multiple Ramp Epochs on an Idle Pool

Correctness | **Low** | **Version 1** | **Risk Accepted**

`LPOracle._A_at_last_timestamp()` is intended to return the A that was effective at `pool.last_timestamp()`, so that `pool.D()`, solved under that A, is paired with a consistent amplification value. The function breaks when more than one ramp epoch has elapsed since the last pool interaction.

The third branch, `t <= initial_t: return initial_A`, returns the `initial_A` of the latest ramp. This is correct only if A was unchanged between `last_timestamp` and the latest ramp start. It is wrong whenever a completed ramp, or a `stop_ramp` followed by a new ramp, occurred in that window while the pool was idle, because `initial_A` then reflects a later A value, not the one under which `D` was computed.

A concrete sequence: swap at `T0` with `A = A1` fixes `last_timestamp = T0` and `D = D0`. Admin ramps `A1 -> A2`, lets it complete, then starts a new ramp `A2 -> A3`. At the oracle call, `t = T0 <= initial_t` of the new ramp, so the function returns `A2`. But `D0` was solved under `A1`. The oracle evaluates `_portfolio_value` with the inconsistent pair `(A2, D0)`, producing an LP price wrong by approximately the pool imbalance times `|A2 - A1|`.

`ramp_A_gamma` and `stop_ramp_A_gamma` never write `last_timestamp` or `self.D`, so any number of admin ramp operations on an idle pool leave the cached `(D, last_timestamp)` frozen while the ramp state advances. The historical value `A(last_timestamp)` is not recoverable from on-chain state once a ramp epoch has completed after that timestamp. The pool stores only the latest ramp's two endpoints.

Risk Accepted

The Curve team accepted the risk, and documented the limitation in `_A_at_last_timestamp()` with an inline WARNING comment.

#006 LP Oracle Can Be Atomically Manipulated Upwards

Design | Low | Version 1 | Risk Accepted

Twocrypto's `LPOracle` or StableSwap's `StableSwapNGLPOracle` can be manipulated to return an inflated LP token value through token donations to the pool or through fee accumulation caused by trading.

The LP token is computed as follows for `StableSwapNGLPOracle`:

```
def _lp_price(pool: IStableSwapNG, i: uint256=0) -> uint256:
    return unsafe_div(
        self._portfolio_value(pool, i) * staticcall pool.get_virtual_price(),
        PRECISION
    )
```

`get_virtual_price()` can be instantly increased by trades to imbalanced positions and back to equilibrium, such that a high fee is accrued. If the pool contains rebasing tokens (`pool_contains_rebasing_tokens = True`, set when `asset_types` contains type `2`), donations can be made by directly transferring tokens to the pool, as `_balances()` reads the live `balanceOf` instead of a cached value.

For `LPOracle` the LP token value is scaled by `pool.D()`:

```
@internal
@view
def _lp_price(pool: IFXSwap, i: uint256=0) -> uint256:
    D: uint256 = staticcall pool.D()
    total_supply: uint256 = staticcall pool.totalSupply()
    return self._portfolio_value(pool, i) * D // total_supply
```

`pool.D()` can be likewise increased by accrual of fees through big imbalanced trades.

The inflation of virtual price or `D` is a donation to the existing LPs and a cost for a theoretical attacker. However, integrating the LP oracle with protocols that require a non-inflatable price can lead to losses.

Risk Accepted

The Curve team accepted the risk as by-design behavior. Integrating with protocols that require a non-inflatable LP price remains the integrator's responsibility.

#007 SanityCheck for Two-Coins Can Be Bypassed with Low-Gas Call

Correctness | Low | Version 1 | Code Corrected

`StableSwapNGLPOracle._sanity_check()` enforces the two-coin restriction indirectly: it performs a static `raw_call` to `coins(2)` with `revert_on_failure=False` and then asserts `not success`. The pool is accepted as a two-coin pool only because the call to `coins(2)` reverted.

Treating "the subcall did not succeed" as "the pool has exactly two coins" is unsound because the `raw_call` forwards all remaining gas and is not given a fixed gas stipend, a caller can constrain the gas supplied to `sanity_check()` so that the `coins(2)` subcall runs out of gas under the 63/64 rule and returns `success == False`, while the outer call still has enough gas to complete. A pool that actually exposes `coins(2)` then passes the two-coin assertion. The check therefore depends on the gas the caller chooses to provide rather than on the pool's coin count.

The failure mode is a **false negative on the multi-coin check**: an actual >2-coin pool can be made to look like a 2-coin pool by restricting the gas supplied to `sanity_check()`.

> **Code Corrected**
>
> The Curve team added as a mitigation a plain `staticcall pool.coins(1)` after the gas-bounded `raw_call`. If the gas-starved `raw_call` fails, the following `staticcall pool.coins(1)` also fails (because of lack of gas) and causes the whole context to revert.

#008 Dead Duplicated `_scaled_price` Helper in LPOracle

Informational **Version 1** **Acknowledged**

`LPOracle._scaled_price()` computes the oracle price in the pool's scaled coordinate system as `unsafe_div(p_oracle * PRECISION, p_scale)` from `pool.price_oracle()` and `pool.price_scale()`. It is never called.

`_portfolio_value()` independently re-fetches `price_oracle()` and `price_scale()` and inlines the same conversion as `p_oracle * PRECISION // p_scale`. The two encodings of the same computation are not derived from a shared helper. The dead helper also carries an unused pair of `price_oracle()` / `price_scale()` staticcalls in the bytecode.

> **Acknowledged**
>
> The Curve team chose to retain `_scaled_price()` as a reference helper for integrations.

#009 Incorrect and Incomplete Comments

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

In `lp_oracle_2` the constant is annotated `# 2^64 < 10^19`, but `2^64 = 18 446 744 073 709 551 616 > 10^19`. The inequality is reversed.

> **Code Corrected**
>
> The Curve team lowered the iteration cap from `64` to `60` and updated the inline comment to the correct `# 10^18 < 2^60 < 10^19`.

#010 LP Oracle Sanity Check `A`-Bound Is a No-Op

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

The `LPOracle` (Twocrypto/FXSwap) and `StableSwapNGLPOracle` contracts expose a standalone `_sanity_check()`. It is assumed to be intended as a one-time deployment check that confirms the pool interface and basic return values, not as per-call input sanitization for `portfolio_value()` / `lp_price()`.

Within that check, the amplification bound is effectively a no-op. `LPOracle._sanity_check()` enforces `pool.A() >= POOL_A_PRECISION`, but `pool.A()` returns `_A_gamma()[0]`, the amplification `A_true * N_COINS * A_MULTIPLIER`. The pool itself enforces `MIN_A = N_COINS * A_MULTIPLIER = 20_000`, so `pool.A()` is always `>= 20_000`. The condition can therefore never fail for any well-formed pool and provides no actual validation of the amplification.

Code Corrected

The Curve team corrected the amplification bound in `LPOracle._sanity_check()` to `pool.A() >= N_COINS**(N_COINS-1) * POOL_A_PRECISION`, so that the threshold scales correctly with the pool's `N_COINS * A_MULTIPLIER` encoding. While the check remains redundant for `N_COINS == 2`-specific pools, the added multiplier makes the bound semantically correct.

#011 No Fallback When `p` Leaves the `[MIN_P, MAX_P]` Safeguard Band

Informational **Version 2** **Code Corrected / Acknowledged**

`lp_oracle_2._get_x_y` enforces a hard bound on the price input:

```
MIN_P: constant(uint256) = 10 ** 16  # 0.01
MAX_P: constant(uint256) = 10 ** 20  # 100
...
assert MIN_P <= p and p <= MAX_P
```

`p` is sourced from `pool.price_oracle()`, which is itself EMA-clamped to the pool's cap band (`[0.5, 2.0] * price_scale` on Twocrypto, `2.0` on StableSwap-NG). Under expected operation `p` is comfortably inside `[0.01, 100]` and the assertion never fires. However, the bound is global and independent of the pool's configured `price_scale`. Thus, it is unlikely but not impossible.

Once the assertion reverts, every external caller of `lp_price` reverts as well. There is no admin path, no fallback price, and no oracle-replacement hook on the LP oracle itself. Any lending market that consumed this oracle as collateral pricing is then unable to liquidate, repay, or update positions for the affected pool until the integrator deploys a new oracle and rewires its markets.

An emergency fallback or safeguard mechanism to replace the oracle or otherwise resolve the situation would be beneficial to mitigate a potential indefinite DOS.

Code Corrected

In **Version 3** the Curve team clamped the spot price entering the Twocrypto EMA from below as well as above:

```
min(max(last_prices, unsafe_div(price_scale, 2)), 2 * price_scale)
```

The same clamp was applied in `TwocryptoView._price_oracle`. As a result, `price_oracle / price_scale` stays within `[0.5, 2.0]`, so the `p` passed by `LPOracle` remains inside `[MIN_P, MAX_P]`.

Acknowledged

The corresponding lower clamp was not added in `CurveStableSwapNG.upkeep_oracles`, where the spot price entering the EMA remains capped only from above. The Curve team acknowledged the remaining issue and plans to update it in the future.

#012 Newton Solver Increases Gas at Low Amplification Despite Reducing the Iteration Count

Informational **Version 4** **Code Corrected**

Overall the empirical analysis shows that the new implementation is significantly more gas efficient than the old one. However, there are edge cases were this does not hold.

A **Version 4** iteration is more expensive than a **Version 3** one, measured at `5094` against `2299` gas, because it adds a second `isqrt` in `_x_from_y` and the `_p_prime_abs` evaluation. The iteration count must therefore fall below `45.1%` of the bisection count for the change to pay for itself. Where it falls only to around `48%`, gas increases:

`A_raw` (`A_eff`)	`p`	Version 3	Version 4	
`2040` (`0.204`)	`3437214130349999616`	20106	47390	`+135.7%`
`21` (`0.0021`)	`2700054615852964352`	26509	46962	`+77.2%`
`10000` (`1.0`)	`23293208816245077837`	45259	47786	`+5.6%`

Code Corrected

The Curve team updated the initial guess to interpolate the constant-product and high-amplification limits and removed redundant calculations in the solver.

#013 Pool Amplification Ramps Can Disable Oracle Quotes

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

`StableSwapNGLPOracle` supports a smaller amplification range than the underlying StableSwapNG pool. A pool that initially passes `sanity_check()` can subsequently enter an oracle-unsupported range through a pool-authorized amplification ramp. Oracle quotes then revert for as long as amplification remains outside the supported range.

For two coins, `_scaled_A_raw()` converts `A_precise()` to `50 * A_precise()`. The library requires `A_raw <= 1000000000`, so oracle pricing requires `A_precise() <= 20000000`, corresponding to pool `A() <= 200000` at integer amplification values. The pool's `ramp_A()` accepts target A values below `1000000`, subject to its timing and per-ramp change limits.

A pool at `A = 100000` passes the oracle's amplification sanity check. A ramp to `A = 300000` is a threefold change within the pool's tenfold limit and, when scheduled under the applicable timing restrictions, can move the pool beyond the oracle's supported range. Above `A_precise() = 20000000`, the library's amplification assertion causes `portfolio_value()` and `lp_price()` to revert. The pricing methods do not call `sanity_check()` or provide a fallback quote.

An integration that requires these quotes may consequently be unable to execute price-dependent operations until amplification returns to the supported range or the integration changes its pricing source. The failure is recoverable; it is not an irreversible or necessarily permanent denial of service.

Code Corrected

> The Curve team increased `lp_oracle_2.MAX_A` from `100000` to `500000`. After the wrapper's amplification scaling, the library now supports the full amplification range permitted by StableSwapNG's `ramp_A()`. Pool-authorized amplification ramps can therefore no longer exceed the library's supported range and disable oracle quotes.

#014 Repeated Calculations Increase Oracle Gas Usage

Informational **Version 4** **Code Corrected**

Every iteration of `_y_newton()` solves for `x` twice. `_p_from_y()` computes it to work out the price, then discards it:

```
# lp_oracle_2._p_from_y
x: uint256 = self._x_from_y(A_raw, y)
```

The Newton branch computes it again, from the same `A_raw` and `y`:

```
# lp_oracle_2._y_newton
pm: uint256 = self._p_from_y(A_raw, y)
...
if iteration < MAX_NEWTON_ITERS and pm < SAFE_P_MAX:
    x: uint256 = self._x_from_y(A_raw, y)
```

It happens once more on the way out: `_y_newton()` returns only `y`, so `_get_x_y()` recovers `x` with a further call, on both branches and at both exits.

`_p_prime_abs()` repeats work too. It stores `xx = x * x`, then computes `16 * A_raw * x * x * y` without using it, and `p * p * y * y` and `x * y * y` both contain `y * y`.

Code Corrected

The Curve team updated `_y_newton()` to reuse `x` for the price and derivative evaluations and return `(x, y)`, avoiding a further calculation in `_get_x_y()`. The derivative calculation also reuses `x * x` and `p * y`, removing the repeated products. Besides, `_p_from_y` was renamed to `_p_from_x_y`.

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.