CHAINSECURITY

PUBLIC

Code Assessment
of the YBLendingOracleLL Smart Contracts

November 03, 2026

Produced for

yieldbasis



Contents

1	Executive Summary	3
2	Assessment Overview	5
3	System Overview	7
4	Trust Model	10
5	System Considerations	12
6	Terminology	14
7	Findings	15
8	Limitations and use of report	24

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 YBLendingOracleLL according to **Scope** to support you in forming an opinion on their security risks.

This is a focused review of the new `YBLendingOracleLL.vy` and `YBLendingOracleLLFactory.vy` contracts, and of `YBLendingOracleLLUSD.vy`, which was added in **Version 2**. `YBLendingOracleLL` prices LT shares for external lending integrators with an exponential moving average (EMA) that smooths the underlying fundamental value to resist same-block manipulation. Yield Basis addressed the reported and self-identified issues in **Version 2**, **Version 3**, **Version 4** and **Version 5**, and also changed how the oracle prices the AMM position when its safe-debt limits stop arbitrage and when the AMM cannot be traded. In **Version 4**, `YBLendingOracle` and `YBLendingOracleLL` additionally price the cryptopool at its current `A` when an `A` ramp has changed it since the pool's last trade, see **The Oracles Price the Pool at a Stale a and D During an a Ramp**. In **Version 5**, both oracles also price a killed AMM at what `LT.emergency_withdraw()` pays, see **Price Ignores Killed Markets**. We reviewed these changes as part of this report.

Security regarding access control is high. Privileged operations are gated by the YB Factory admin (DAO), and all other entry points are permissionless and cannot harm the system. Arithmetic precision is good, but users should be aware of the rounding direction of the price computations.

Functional correctness is satisfactory overall. All issues we found have been resolved or acknowledged.

Integrators should be aware of the following:

- Price queries revert for markets without outstanding LT tokens.
- While the AMM cannot be traded, the price is the live balance value of the position and is not smoothed by the EMA, see **The Oracle's Own EMA Protects Only Half of the Reported Price**.
- The price is not continuous at the boundary of the untradable region: it steps up when the AMM becomes untradable and back down when it becomes tradable again, by about ten percent in the Yield Basis team's tests. Positions near the liquidation threshold can become liquidatable at that moment, and borrowers can overborrow at the higher price. Lending integrators need to take this into account, see **ybLT Price Excludes the Pending Arbitrage Loss and Steps at the Untradable Boundary**.
- While a market is killed, the price is what `LT.emergency_withdraw()` pays and is not smoothed by the EMA. It is not continuous at a kill or an unkill, see **Price Ignores Killed Markets**.
- The EMA is only advanced by `price_w()` calls. Checkpoint regularly, also while a market is untradable, otherwise the price reads well above the reference oracle after recovery.

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 that is referenced throughout this report.

2.1 Scope

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

Version	Date	Commit Hash	Note
1	9 September 2026	7c8c076cfbfa3672a6a0fd10f1cbfaa86476e66a	Initial review
2	16 October 2026	14bf040e9bc55ec54a7d4f624f37e16917b88609	Fixes
3	23 October 2026	a31ccd5f94716b0c3514c22395ee8c7c610a09c7	Untradable AMM fix
4	29 October 2026	db73d5200832a484c6968f35dccfab71ec2477c2	Live A and gas optimisation
5	30 October 2026	51132f56e17319ce7eedecd53733f8f6f79b6245	Killed AMM fix

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

The files in scope are:

```
contracts/
└── utils/
    ├── YBLendingOracle.vy
    ├── YBLendingOracleLL.vy
    ├── YBLendingOracleLLFactory.vy
    └── YBLendingOracleLLUSD.vy
```

`YBLendingOracleLLUSD.vy` was added in **Version 2** as part of the fix for Per-Market USD and Asset Oracles Redundantly Repeat the Same Computation and is in scope from **Version 2** onwards.

For `YBLendingOracle.vy`, only the fixes made to both `YBLendingOracle.vy` and `YBLendingOracleLL.vy` are in scope, such as the fix for Price Ignores the AMM's Safe-Debt Limits.

2.1.1 Excluded from scope

All other contracts in the repository are out of scope. This includes the changes made between **Version 1** and **Version 2** to files outside of the scope, such as `LTMigrator.vy`, the deployment scripts and the tests. The changes made between **Version 2** and **Version 3** are limited, within the scope, to `YBLendingOracleLL.vy`: the fundamental is now also computed while the AMM cannot be traded, see The Oracle's Own EMA Protects Only Half of the Reported Price. All other files changed in that range, such as `LTMigratorGated.vy`, `LTDepositZap.vy`, the deployment scripts and the tests, are out of scope. The changes made between **Version 3** and **Version 5** are limited, within the scope, to `YBLendingOracle.vy` and `YBLendingOracleLL.vy`: both read the cryptopool at its current `A`, see The Oracles Price the Pool at a Stale a and D During an a Ramp, and price a killed AMM, see Price Ignores Killed Markets. All other files changed in that range, such as the deployment scripts, the documentation and the tests, are out of scope. The rest of `YBLendingOracle.vy` is out of scope, including the **Version 2** change that removes the `LT.updated_balances()` call from

`staked_price_in_asset()` and `staked_price_in_usd()`. `YBLendingOracle.vy` was covered in a prior report.

`YBLendingOracleLL` imports `LPOracle` from the `twocrypto_lp_oracle` submodule (`contracts/twocrypto_lp_oracle`) for the interpolated `A` parameter and the StableSwap invariant solve, and `snekmate.utils.math` for `_wad_exp()` and `_mul_div()`. Both are external dependencies and out of scope.

3 System Overview

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

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.

`YBLendingOracleLLFactory` and `YBLendingOracleLL` together provide an EMA-smoothed, per-market ybLT price oracle suitable for use by a Curve LlamaLend lending market. Each market gets two clones: one priced in USD, one priced in the underlying asset.

3.1 YBLendingOracleLLFactory

`YBLendingOracleLLFactory` deploys and administers the per-market `YBLendingOracleLL` clones. It is bound at deploy to the YB `Factory` (for resolving `market_id -> lt`), the `YBLendingOracleLL` implementation, a default EMA time, and a `dao` address.

`create_oracles(market_id)` is permissionless. It resolves the market's `LT` from `Factory.markets(market_id)` and reverts if the market does not exist or has no `lt`. It deploys two EIP-1167 minimal-proxy clones of the `YBLendingOracleLL` implementation, `initialize()`s one with `in_usd=True` and the other with `in_usd=False`, and stamps both with the current `default_ema_time`. The call is idempotent: if oracles already exist for the market, it returns the existing pair without redeploying.

`dao` is a plain storage address, set at deploy to the YB Factory admin but not read from the live `Factory` on each call, so it does not automatically track a later change of the `Factory`'s own admin. `dao` may:

- Retune the EMA time of a single clone (`set_ema_time()`) or both clones of a market (`set_ema_time_market()`).
- Change the EMA time stamped on future clones (`set_default_ema_time()`). Existing clones are unaffected.
- Transfer the `dao` role to a new address (`set_dao()`).

`MAX_EMA_TIME` (`10**9`, about 31.7 years) bounds `ema_time` on both `default_ema_time` and per-clone updates.

3.2 YBLendingOracleLL

Each clone prices one `LT` market in one denomination (`in_usd`), both fixed at `initialize()`. `initialize()` is one-time: it reverts if `lt_token` is already set.

The price is decomposed into a fundamental rate and a live shift:

- The fundamental is the `LT -> asset` conversion rate at `price == price_scale` (analogous to `LT.pricePerShare()`). When the AMM position is solvable, it reproduces the AMM's `get_x0()` and the LT's admin-fee and liquidity accounting (`_calculate_fresh_lv()`) to price the position, and does not depend on `price_oracle`. Because it absorbs same-block wash-trade pumping of the AMM's equity and of the cryptopool's `virtual_price`, it is the quantity that gets EMA-smoothed.
- The shift to `price == price_oracle` (`po_shift`) is, while the AMM position is solvable, a pure function of the pool's `A` parameter and the `price_oracle / price_scale` ratio, independent of `virtual_price` or AMM state, so it is applied live rather than smoothed; it is manipulation-resistant because `price_oracle` is itself the pool's own EMA. This no longer holds in the insolvency fallback described

below, where `po_shift` is instead built from live `virtual_price`, `collateral`, and `debt` with no smoothing at all.

`price()` is a view: it reads the EMA state already committed to storage and applies the live shift, without checkpointing anything. `price_w()` is state-changing: it advances the fundamental EMA using the previously recorded fundamental, records the current fundamental for the next call, checkpoints the aggregator (`agg.price_w()`), and returns the price computed from the newly advanced EMA. Consumers are expected to call `price_w()` regularly (e.g., on every borrow/liquidate) so the EMA tracks. Before the first `price_w()`, the EMA is unseeded (`fundamental_ema == 0`) and `price()`/`price_w()` fall back to the raw, unsmoothed fundamental.

If the AMM's position is imbalanced enough that `get_x0()` would revert (the ratio-9/16 insolvency boundary at `L=2`), `_get_x0()` returns `solvable=False` instead of reverting, and the fundamental and shift fall back to a balance-based valuation (`collateral * lp_price_ps * agg_price - debt`, floored at 0) using the LT's last-committed liquidity and supply values directly, bypassing `_calculate_fresh_lv()`.

Both `price()` and `price_w()` guard against reentrancy by probing the AMM's own `check_nonreentrant()` lock via a static `raw_call`.

`set_ema_time()` on a clone is callable only by the `factory` address recorded at `initialize()`, which is itself DAO-gated.

3.3 Relation to the Previous `YBLendingOracle` / `YBPriceProxy` Design

The repository also contains `YBLendingOracle.vy` and `YBPriceProxy.vy`: a singleton oracle keyed by `LT` address, with per-market `YBPriceProxy` clones forwarding to it. `YBLendingOracleLLFactory` and `YBLendingOracleLL` replace that split with a factory that deploys self-contained, per-market oracle clones directly, each holding its own EMA state rather than delegating to a singleton.

Unlike the current design, `YBLendingOracle` has no protection against same-block oracle manipulation. It uses the most recent fundamental value as is with no delay or EMA. Besides this, it uses an equivalent method to price ybLT tokens.

`YBLendingOracle.vy` and `YBPriceProxy.vy` are out of scope for this review; they were covered in a prior report.

3.4 Changes in Version 2

- `po_shift` now takes into account the divergence between the price at which the AMM trades its collateral and the collateral price implied by the pool portfolio value. See Price Ignores the AMM's Safe-Debt Limits for details. `YBLendingOracle` has also been updated to use the same formula without an EMA.
- The fundamental is not updated while the AMM is not solvable (untradable). The EMA therefore converges to the most recent price snapshot.
- `YBLendingOracleLL` no longer supports `in_usd`. Instead, its existing functions act as if `in_usd` is false, and the new `price_usd()` and `price_usd_w()` functions return the USD price. `YBLendingOracleLLUSD` was added and acts as a proxy that forwards calls to the `YBLendingOracleLL` instance to mimic the behavior of `in_usd == True`. See Per-Market USD and Asset Oracles Redundantly Repeat the Same Computation.

3.5 Changes in Version 3

- While the AMM is not solvable, the EMA is fed with the fallback balance-based price instead of the most recent solvable price snapshot.

3.6 Changes in Version 4

- `YBLendingOracleLL` and `YBLendingOracle` read the current `A` of the cryptopool (`pool.A()`) instead of the `A` at its last trade. If the end time of the `A` ramp is later than the last trade (`future_A_gamma_time > last_timestamp`), the `A` has changed since the pool cached `D`. In that case the oracles recalculate `D` from the pool's stored balances with the pool's `newton_D()` at the current `A`, as `Twocrypto._get_D()` does, and compute `virtual_price` as `1e18 * xcp(D) / totalSupply`, as `Twocrypto.tweak_price()` does. Otherwise they use the cached `D` and `virtual_price`. See The Oracles Price the Pool at a Stale a and D During an a Ramp.

3.7 Changes in Version 5

- `YBLendingOracleLL` and `YBLendingOracle` read `AMM.is_killed()`. A killed AMM is treated like an untradable one: the price is the balance value of the position, which is what `LT.emergency_withdraw()` pays, and it is not smoothed by the EMA. A killed LT pays a pro-rata share of the whole AMM, so the admin fee buffer is not deducted, and the staked price uses the stored balance of the staker. `YBLendingOracleLL` keeps recording the fundamental, so when the AMM is unkilled the price returns to the value after arbitrage. See Price Ignores Killed Markets.

3.8 Deployed Instances (Mainnet)

Addresses of a valid deployment on Ethereum mainnet were provided by Yield Basis. Under the assumption that `0xf939E0A03FB07F59A73314E73794Be0E57ac1b4E` is the canonical `crvUSD` deployment, then

- `0x370a449FeBb9411c95bf897021377fe0B7D100c0`'s runtime bytecode matches the Yield Basis Factory
- `0xcE43c6668945FEfa079Da1c38F7CDab5A525Ec7C`'s runtime bytecode matches `YBLendingOracle`
- `0xaC866A97FB35fdE170853d3dA58Da7195F97f7f8`'s runtime bytecode matches `YBLendingOracleLLFactory`

The bytecode of contracts that these contracts immutably rely upon was also checked. Parameters, including addresses, that can be set by privileged actors were not checked.

Additionally, we verified that in block 26449532 (Thu, 19 Nov 2026 13:49:35 UTC), the ENS names `oracles.yieldbasis.eth` and `oracles-ll.yieldbasis.eth` resolve to the last two contracts.

4 Trust Model

4.1 Roles

DAO (`YBLendingOracleLLFactory.dao`)

- Trust level: Fully trusted.
- Permissions: Retune the EMA time of any existing clone (`set_ema_time()` , `set_ema_time_market()`), change the EMA time stamped on future clones (`set_default_ema_time()`), and transfer the `dao` role (`set_dao()`). Cannot change which `LT` a clone prices, its `in_usd` denomination, or the deployed `FACTORY` / `LL_IMPL` addresses, all of which are set once at deploy or `initialize()` . In **Version 2** , `set_ema_time()` and `set_ema_time_market()` are replaced by `set_ema_time(market_id, ema_time)` , which retunes the single EMA shared by a market's USD and asset prices; `in_usd` no longer exists, and the `USD_BLUEPRINT` address is also set once at deploy.
- Assumptions: Set at deploy to the YB Factory admin's address, but stored independently and not synchronized with the live `Factory` admin afterwards. A later change of the `Factory` 's own admin has no effect here unless `set_dao()` is also called.
- Worst-case impact: Set `ema_time` to an extreme value (up to `MAX_EMA_TIME` , about 31.7 years) on any market's oracle, degrading its manipulation resistance (too short) or its responsiveness to real price moves (too long). Cannot directly move funds or redirect a clone to a different `LT` .

Sequencer / block builder (entity controlling transaction ordering)

- Trust level: Fully trusted on networks where a single entity controls consecutive blocks.
- Permissions: No privileged on-chain role. Controls transaction ordering and block placement.
- Assumptions: See Assumptions on the Underlying Sequencing Mechanism.
- Worst-case impact: Reliably manipulate the EMA oracle's input at no competitive cost, degrading its manipulation resistance for every consumer on that network. Cannot directly move funds or change oracle parameters.

Anyone (permissionless callers)

- Trust level: Untrusted.
- Permissions: Call `create_oracles(market_id)` to deploy the USD/asset oracle pair for any existing YB Factory market. The call is idempotent, so repeated or racing calls are harmless. Call `price_w()` on any deployed clone to checkpoint its EMA. In **Version 2** , `price_w()` / `price_usd_w()` on the `YBLendingOracleLL` clone and `price_w()` on its `YBLendingOracleLLUSD` forwarder all checkpoint the same EMA.
- Worst-case impact: None directly. `price_w()` only advances the EMA using already-committed and live-read state, and `create_oracles()` cannot be used to overwrite an existing pair.

4.2 External Trusted Components

- **YB** **`Factory`** : Supplies `market_count()` and `markets(market_id).lt` , used by `create_oracles()` to resolve and validate the market. Out of scope for this review.
- **`LT`** : Supplies `liquidity()` , `totalSupply()` , `staker()` , `balanceOf()` , `min_admin_fee()` , `amm()` , `CRYPTOPOOL()` , `agg()` . Its liquidity and admin-fee accounting is reproduced in `_calculate_fresh_lv()` . The oracle's correctness depends on this reproduction staying aligned with `LT` 's own implementation. Out of scope for this review.

- **`LevAMM` (`LT.amm()`)**: Supplies `collateral_amount()`, `get_debt()`, and the read-only reentrancy probe `check_nonreentrant()`. `_get_x0()` reproduces `AMM.get_x0()` bit-for-bit. A future change to `AMM.get_x0()` would need a matching change here. In **Version 2**, `_sqrt_arb_endpoint()` additionally hardcodes the AMM's `MIN_SAFE_DEBT` / `MAX_SAFE_DEBT` limits (at `L=2`) and the rule `AMM.exchange()` applies to them, so a change to either in the AMM would also need a matching change here. In **Version 3**, when `get_x0()` cannot be solved, `_get_x0()` additionally returns its double root at the solvability boundary, `coll_value / (2 * LEV_RATIO)`, which the oracle records as the fundamental while the AMM cannot be traded, so it relies on `AMM.get_x0()` being continuous at that boundary. In **Version 5**, `is_killed()` is read as well. `LT.set_killed()`, callable by the LT admin, the YB Factory admin and the emergency admin, therefore changes the reported price. Out of scope for this review.
- **Cryptopool (`IFXSwap`)**: Supplies `price_oracle()`, `price_scale()`, `virtual_price()`, and the `A`-ramp timing accessors (`last_timestamp()`, `initial_A_gamma()`, `initial_A_gamma_time()`, `future_A_gamma()`, `future_A_gamma_time()`) consumed via `LPOracle._A_at_last_timestamp()` to compute `pv_norm`. In **Version 4**, the oracle instead reads `A()` and, during an unobserved `A` ramp, additionally `gamma()`, `precisions()`, `balances()`, `totalSupply()` and the pool's `MATH` contract (`newton_D()`) to recalculate `D` at the current `A`. This reproduces `Twocrypto._get_D()` and `Twocrypto._xcp()`, so a change to either in the pool would need a matching change here. The live `po_shift` calculation relies on `price_oracle()` being the pool's own EMA for its manipulation resistance. No access control on these views. Out of scope for this review.
- **Price aggregator (`agg`)**: Supplies `price()` / `price_w()` (crvUSD/USD), feeding every USD-denominated calculation including the `asset_price` used by `in_usd` clones (in **Version 2**, by `price_usd()` / `price_usd_w()`). Out of scope for this review.
- **`LPOracle` (`contracts/twocrypto_lp_oracle`)**: Supplies `_A_at_last_timestamp()`, `_scaled_A_raw_from_A()`, and `_get_x_y()`, used to compute `pv_norm`, the price-ratio-only portfolio value multiplier. Imported submodule, out of scope for this review.
- **`snekmate.utils.math`**: Supplies `_wad_exp()` (EMA decay factor) and `_mul_div()` (signed full-precision multiply-divide). External library, out of scope for this review.

4.3 Upgradeability

Each clone (`YBLendingOracleLL`) is an EIP-1167 minimal proxy pointing at the immutable `LL_IMPL` implementation set at `YBLendingOracleLLFactory` deploy. Neither the factory's `FACTORY` / `LL_IMPL` bindings nor a clone's `lt_token` / `in_usd` / `factory` (`lt_token` / `factory` in **Version 2**) bindings (set once via `initialize()`) can be changed after creation. `ema_time` is the only mutable per-clone parameter, and only through the DAO-gated factory. The factory itself is not upgradeable: a redesign requires deploying a new factory and implementation and pointing consumers at newly created clones.

In **Version 2**, a clone no longer has an `in_usd` binding, and the per-market `YBLendingOracleLLUSD` forwarder is deployed from the immutable `USD_BLUEPRINT` blueprint with the market's `YBLendingOracleLL` clone as its immutable `ORACLE`, and is not upgradeable either.

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 Assumptions on the Underlying Sequencing Mechanism

System Consideration **Version 1**

In order to protect against wash trading, the EMA is only advanced using a measurement recorded in a previous, different block; the guard checks elapsed time, not who produced the intervening block. Because of this, the protection does not strictly require the entity that controls block ordering to change: any two ordinary, uncoordinated permissionless transactions landing in two different blocks can reproduce the same manipulate-then-checkpoint pattern, on Ethereum Mainnet as much as on any L2. A sequencer or block builder who also controls consecutive blocks makes the attack reliable and cost-free rather than just possible, since it no longer has to compete with ordinary transaction ordering to land the checkpoint transaction itself. Many L2 networks have a single sequencer that rotates infrequently if at all, which gives that entity exactly this guarantee. Hence, on such networks the sequencer must be fully trusted for the EMA oracle to be considered safe there.

SC2 crvUSD = USD Equivalence Assumption

System Consideration **Version 1**

The oracle treats crvUSD as 1:1 with USD when accounting for internal debt. `get_x0()` values collateral, taking into account the reported price of crvUSD, but converts crvUSD debt to USD at 1:1. The fallback branch performs a similar treatment.

This is consistent with YB core, so there is no internal arbitrage between the AMM, LT share pricing, and the oracles. The deviation only exists relative to true USD valuations outside the system.

If crvUSD trades at price `d` in USD, the fallback branch's plain subtraction makes the error exactly `debt * (1 - d)`. In the primary path, `debt` enters `get_x0()`'s solvency formula under a square root, so the propagated error there is a nonlinear function of collateral value, debt, and `d` rather than this same flat difference, though the direction is unchanged.

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.

SC3 The Oracle's Own EMA Protects Only Half of the Reported Price

System Consideration **Version 1**

In `YBLendingOracleLL`, `fundamental` is protected by a dedicated exponential moving average whose time constant, `ema_time`, is stored by the oracle itself and can be retuned by the DAO specifically to resist manipulation. The other half of the price, `po_shift`, carries no equivalent protection of its own:

```
s: uint256 = 2 * L * isqrt(pv_norm * PRECISION)
if s > (2 * L - 1) * PRECISION:
    po_shift = (s - (2 * L - 1) * PRECISION) * price_scale // price_oracle
```

It reads the cryptopool's `price_oracle` and `price_scale` live on every call, so its only protection against manipulation is whatever lag the cryptopool itself happens to apply to those two values. `price_oracle` absorbs a trade's price with roughly a one-block minimum delay and then keeps absorbing more of it on every following block, at a rate set by the pool's own moving-average time constant. `price_scale` moves far slower, gated by accumulated pool profit and capped to a small step per block. The two don't have to move in the same direction either: `price_scale`'s own slow repeg can move it up, while a trade pushes `price_oracle` down. Since `po_shift` uses that same `price_oracle` / `price_scale` ratio twice, once inside `pv_norm` and again as the explicit multiplier, the two uses can compound. The practical effect is that a price reported by `YBLendingOracleLL` can change more than either of `price_oracle` or `price_scale` in the same time frame.

In **Version 2**, `po_shift` additionally uses the arbitrage endpoint from `_sqrt_arb_endpoint()`, which depends on the AMM's current state whenever that state is outside the AMM's safe-debt band. Moreover, when the AMM cannot be traded at all (`get_x0()` cannot be solved), the reported price is the balance value of the position, computed only from values read in the current call (the cryptopool's `virtual_price`, the AMM's `collateral` and `debt`, and the LT's liquidity and supply):

```
cv_po: uint256 = collateral * lp_price_ps // PRECISION * agg_price // PRECISION * pv_norm
    // PRECISION
if cv_po > debt:
    lv: LiquidityValues = staticcall lt.liquidity()
    balance_price = ((cv_po - debt) * lv.total // (convert(max(lv.admin, 0), uint256) +
        lv.total)
                    * PRECISION // staticcall lt.totalSupply()) * PRECISION // asset_price
```

In that state, the EMA protects none of the reported price. While the AMM is untradable, `AMM.exchange()`, `LT.deposit()` and `LT.withdraw()` revert and `LT.emergency_withdraw()` removes collateral and debt proportionally, so these inputs mainly change through the cryptopool, for example when trading fees raise its `virtual_price`.

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

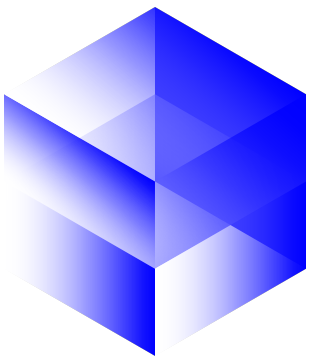
We categorize the findings into four distinct categories, depending on their severity. These severities are derived from the likelihood and the impact using the following table, following a standard risk assessment procedure.

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.



No 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	Price Queries Revert on Markets with No LT Tokens Outstanding	Specification Changed ✓
Medium	#004	The EMA Ignores the Position While the AMM Is Untradable	Code Corrected ✓
Low	#002	Per-Market USD and Asset Oracles Redundantly Repeat the Same Computation	Code Corrected ✓
Low	#005	Price Ignores Killed Markets	Code Corrected ✓
Low	#006	Price Ignores the AMM's Safe-Debt Limits	Code Corrected ✓
Low	#007	The Oracles Price the Pool at a Stale a and D During an a Ramp	Code Corrected ✓
Info	#003	Gas Optimizations	Acknowledged ⊖
Info	#008	The Pool's Price Oracle Cannot Fall Below Half of Price_scale	Acknowledged ⊖
Info	#009	ybLT Price Excludes the Pending Arbitrage Loss and Steps at the Untradable Boundary	Acknowledged ⊖

7.2 Descriptions

#001 Price Queries Revert on Markets with No LT Tokens Outstanding

Correctness | Medium | Version 1 | Specification Changed

In `YBLendingOracleLL`, `price()` and `price_w()` divide by the LT token's total supply unconditionally, on either pricing branch. Every call ends in the same final step:

```
fundamental: uint256 = (value_ps * lv_total // (convert(max(lv_admin, 0), uint256) +
    lv_total)
                        * PRECISION // lt_supply) * PRECISION // p_o_ps
```

A market with zero LT tokens outstanding, such as a newly created market before its first deposit, has `lt_supply` equal to zero here, so this division reverts however it is reached. When the AMM position is solvable, `lt_supply` comes from an internal liquidity-split step that already fails earlier, on its own supply division:

```
supply: int256 = convert(staticcall lt.totalSupply(), int256)
f_a: int256 = convert(
    10**18 - (10**18 - staticcall lt.min_admin_fee()) * isqrt(convert(10**36 - staked *
        10**36 // supply, uint256)) // 10**18,
    int256)
```

This path is reached because the leverage-solvency check classifies an all-zero collateral/debt position as solvable rather than as the no-value case it effectively is:

```
coll_value: uint256 = p_oracle * collateral // PRECISION
d_sub: uint256 = 4 * coll_value * LEV_RATIO // PRECISION * debt
if coll_value * coll_value < d_sub:
    return (False, 0)
```

With `collateral == 0` and `debt == 0`, both `coll_value` and `d_sub` are `0`, so the check takes the solvable branch. Reclassifying this state as insolvent would not fix the problem on its own: the insolvent fallback path fetches `lt_supply` directly from `lt.totalSupply()` and feeds it into the same final division shown above, so it reverts there too whenever supply is zero. `price()` and `price_w()` are then completely unusable for that market until a nonzero supply exists, on either path.

Specification Changed

Yield Basis clarified that the oracle is only meant to be queried for markets with a nonzero LT supply, and that lending markets using it are created after the market has been seeded.

In **Version 2**, the code is unchanged, so `price()` and `price_w()` still revert while the LT's total supply is zero. Since the reverting calls do not change any state, an oracle deployed before the first deposit starts pricing normally once the LT has a nonzero supply. Note that the supply can also return to zero later if all shares are withdrawn, in which case price queries revert again.

#004 The EMA Ignores the Position While the AMM Is Untradable

Correctness | Medium | Version 2 | Code Corrected

In **Version 2**, `YBLendingOracleLL.price_w()` records a fundamental, and with it advances the EMA input `fundamental_last`, only while the AMM is tradable:

```
if snap.tradable:
    if ema == 0:
        ema = snap.fundamental                # seed the EMA at the first checkpoint
    self.fundamental_last = snap.fundamental  # record current value for the NEXT advance
# Untradable: nothing to record, the EMA keeps converging to the last tradable fundamental
```

While the AMM cannot be traded (`get_x0()` has no solution), the returned price is the balance value of the position and does not use the EMA. The EMA, however, keeps converging to the last fundamental recorded before the AMM became untradable.

That fundamental does not describe the position when the AMM becomes tradable again. While untradable, the EMA will use the last stale fundamental and will converge to it. Once it is tradable again, arbitrage resumes and the value of the position drops. Until the EMA has converged to the real fundamental via checkpoints after recovery, the reported price will be overvalued for a multiple of `ema_time`.

Since the oracle is meant for lending markets, overvaluing ybLT allows borrowers to take out loans that are not covered by their collateral, which can leave the market with bad debt.

Code Corrected

In **Version 3**, `_get_x0()` returns the double root of the `get_x0()` quadratic, `coll_value / (2 * LEV_RATIO)`, when it has no solution, instead of `0`. This is the value of `get_x0()` at the solvability boundary, so `x0` is continuous across it. `_snapshot()` now computes the fundamental off this `x0` in all cases, and `price_w()` records it and advances the EMA also while the AMM is untradable. On recovery, the EMA has therefore converged to the post-arbitrage fundamental and not to the stale pre-untradable one.

The price itself is not continuous at the boundary, and the EMA is only advanced by `price_w()`. See System Overview.

#002 Per-Market USD and Asset Oracles Redundantly Repeat the Same Computation

Design | **Low** | **Version 1** | **Code Corrected**

In `YBLendingOracleLLFactory`, `create_oracles()` deploys two separate `YBLendingOracleLL` clones for each market, one denominated in USD and one in the underlying asset:

```
usd = create_minimal_proxy_to(LL_IMPL)
asset: address = create_minimal_proxy_to(LL_IMPL)
extcall LendingOracleLL(usd).initialize(lt, True, ema_time, self)
extcall LendingOracleLL(asset).initialize(lt, False, ema_time, self)
```

Each clone independently runs the full snapshot computation and keeps its own EMA state (`fundamental_ema`, `fundamental_last`, `ema_ts`), but `in_usd` is never read inside that computation:

```
def _snapshot(lt: LT, pool: IFXSwap, amm: LevAMM, agg_price: uint256) -> Snapshot:
    ...
    return Snapshot(fundamental=fundamental, po_shift=po_shift,
                    asset_price=price_oracle * agg_price // PRECISION)
```

`in_usd` only matters in the very last step, where it decides whether to multiply by `asset_price`:

```
def _price_from(fundamental: uint256, snap: Snapshot) -> uint256:
    price_asset: uint256 = fundamental * snap.po_shift // PRECISION
    if self.in_usd:
```

```
        return price_asset * snap.asset_price // PRECISION
    return price_asset
```

So the two clones for the same market redundantly repeat the identical round trip of external calls into the LT, AMM, cryptopool and aggregator, and each maintains a separate copy of what is the same fundamental EMA. Only that final one-line multiplication actually differs between them. Every `price_w()` checkpoint has to be paid for twice to keep both denominations fresh, and if only one of the pair is checkpointed in practice, the other's EMA silently drifts out of sync with it despite tracking the same underlying value.

Code Corrected

`YBLendingOracleLLFactory.create_oracles()` now deploys a single `YBLendingOracleLL` clone per market, which owns the market's only EMA and exposes `price()` / `price_w()` in the underlying asset and `price_usd()` / `price_usd_w()` in USD. The USD oracle is the new `YBLendingOracleLLUSD`, a stateless forwarder deployed from a blueprint, whose `price()` / `price_w()` forward to `price_usd()` / `price_usd_w()` of the market's `YBLendingOracleLL` clone. Checkpoints from either denomination therefore advance the same EMA, and a single `price_w()` call keeps both denominations fresh.

#005 Price Ignores Killed Markets

Correctness	Low	Version 4	Code Corrected

`LT.set_killed()` lets the LT admin, the YB Factory admin or the emergency admin kill a market. While a market is killed, `AMM.exchange()` reverts, so the AMM cannot be traded. The only way out is `LT.emergency_withdraw()`, which pays a pro-rata share of the AMM's collateral and debt.

`YBLendingOracleLL` does not read `AMM.is_killed()`. As long as `get_x0()` has a solution, it prices a killed market like a tradable one, with the value after arbitrage of the AMM towards the market LP price:

```
x0_ok, x0 = self._get_x0(p_o_amm, collateral, debt)
...
if x0_ok:
    ...
    sqrt_e: uint256 = self._sqrt_arb_endpoint(sqrt_r, x0, debt, p_o_amm * collateral //
        PRECISION)
```

This arbitrage cannot happen while the market is killed, so the reported price is not what the ybLT can be redeemed for, which is what `LT.emergency_withdraw()` pays. For a lending market, the collateral is then valued differently from what liquidations can recover.

Code Corrected

In **Version 5**, `YBLendingOracleLL` and `YBLendingOracle` read `AMM.is_killed()` and treat a killed AMM as untradable:

```
killed: bool = staticcall amm.is_killed()
tradable: bool = x0_ok and not killed
```

A killed market is priced with the balance value of the position, which is what `LT.emergency_withdraw()` pays. When the market is unkilled, the price returns to the value after arbitrage, so it is not continuous at a kill or an unkill.

#006 Price Ignores the AMM's Safe-Debt Limits

Correctness | Low | Version 1 | Code Corrected

`YBLendingOracleLL` calculates the ybLT price as `fundamental * po_shift`. The `fundamental` is the value of the AMM position at the cryptopool's `price_scale`. `po_shift` changes it to the value at the market price. For this, `YBLendingOracleLL` assumes that arbitrageurs trade the AMM all the way to the market LP price:

```
s: uint256 = 2 * L * isqrt(pv_norm * PRECISION)
if s > (2 * L - 1) * PRECISION:
    po_shift = (s - (2 * L - 1) * PRECISION) * price_scale // price_oracle
```

Terms used below:

- **AMM price oracle:** `AMM.PRICE_ORACLE_CONTRACT.price()`, the LP token price at `price_scale` multiplied by the crvUSD aggregator price.
- **Market LP price:** the LP token price calculated with the pool's `price_oracle`. `r = pv_norm` is the market LP price divided by the AMM price oracle.
- **AMM price:** the current price of the AMM. `u = sqrt(AMM price / AMM price oracle)` shows where the AMM is on its curve. At equilibrium `u = 1`.

`AMM.exchange()` does not allow arbitrage to go all the way. After every trade it requires `debt / coll_value` to stay between `MIN_SAFE_DEBT = 1/16` and `MAX_SAFE_DEBT = 17/32`. There is one exception: a trade that moves the AMM towards its equilibrium (`debt / coll_value = 1/2`) without crossing it.

Along the AMM curve (`x0` and the AMM price oracle fixed), `debt / coll_value = 3/2 * u - u**2`. So the safe-debt limits become limits on `u`:

- If the market LP price falls, arbitrage stops at `u = 3/4 + sqrt(2)/8` (about `0.927`).
- If it rises, arbitrage stops at `u = 3/4 + sqrt(2)/2` (about `1.457`).
- If the AMM is already past the limit in the direction of the trade, arbitrage does not move the AMM at all.

If arbitrage leaves the AMM at `u_e`, the value of the position at the market price, relative to its value at `price_scale`, is:

```
L * (pv_norm / u_e + u_e) - (2 * L - 1)
```

This value is smallest when `u_e = sqrt(pv_norm)`. This is the full arbitrage that `YBLendingOracleLL` assumes, and it gives `2 * L * sqrt(pv_norm) - (2 * L - 1)`. When the limits stop arbitrage earlier, the real value is higher. So `YBLendingOracleLL` reports a price below the real value of the position. The difference is zero when the limits do not stop arbitrage. It grows with the distance between the market LP price and the AMM price oracle. For large moves, `YBLendingOracleLL` can report zero while the position still has value.

`YBLendingOracleLL` is meant for lending markets. A price that is too low can cause borrowers to be liquidated earlier than necessary.

This issue was identified by Yield Basis after the initial review.

Code Corrected

`YBLendingOracleLL` now finds where arbitrage really leaves the AMM, in the new `_sqrt_arb_endpoint()`. The function limits `sqrt(pv_norm)` to the states that `AMM.exchange()` lets

arbitrage reach from the current state of the AMM. It uses the constants `SQRT_P_MAX_DEBT = 3/4 + sqrt(2)/8` and `SQRT_P_MIN_DEBT = 3/4 + sqrt(2)/2`. The shift is now:

```
s: uint256 = L * (sqrt_r * sqrt_r // sqrt_e + sqrt_e)
if s > (2 * L - 1) * PRECISION:
    po_shift = (s - (2 * L - 1) * PRECISION) * price_scale // price_oracle
```

We checked the constants against `AMM.MIN_SAFE_DEBT` and `AMM.MAX_SAFE_DEBT`, the `debt / coll_value` relation along the AMM curve, and the limit on `sqrt(pv_norm)`. The limit matches the states that `AMM.exchange()` allows, in all cases, including AMM states outside the safe band.

In addition, when the AMM cannot trade at all, the price is now the balance value of the position at the pool's `price_oracle`. This is what `LT.emergency_withdraw()` pays. The EMA does not smooth this part of the price (see The Oracle's Own EMA Protects Only Half of the Reported Price). The effect of both changes on the price at the boundary is described in ybLT Price Excludes the Pending Arbitrage Loss and Steps at the Untradable Boundary.

#007 The Oracles Price the Pool at a Stale a and D During an a Ramp

Correctness **Low** **Version 3** **Code Corrected**

In **Version 3**, `YBLendingOracle` and `YBLendingOracleLL` read `virtual_price()` and `D()` from the pool's storage and `A` at the pool's `last_timestamp`:

```
vprice: uint256 = staticcall pool.virtual_price()
A_raw: uint256 = LPOracle._scaled_A_raw_from_A(
    LPOracle._A_at_last_timestamp(LPOracle.IFXSwap(pool.address)))
```

The pool refreshes `D` and `virtual_price` only in trades. While `A` ramps and no trade happens, the oracles price the curve as of the last trade. The pool's next trade recalculates `D` at the current `A` (`Twocrypto._get_D()`). After `stop_ramp_A_gamma()`, the stopped `A` is paired with the old `D`.

The error is second order. It grows with the change of `A` since the last trade and with the distance between `price_oracle` and `price_scale`. In the repository's ramp tests, **Version 3** overvalued ybLT by up to 0.05% with `price_oracle` 0.37% from `price_scale`. With `price_oracle` 4.7% from `price_scale`, it overvalued ybLT by up to 3.8% after a 0.1x ramp left unobserved for two days. Any trade in the pool removes the error.

This issue was found by Yield Basis and reported to us after the initial review.

Code Corrected

In **Version 4**, both oracles read the current `A` with `pool.A()`. While a ramp has not been observed by a trade and the pool did not trade in the current block, they recalculate `D` from the stored balances with the pool's `newton_D()`. `virtual_price` is computed as `1e18 * xcp(D) / totalSupply`, as in `Twocrypto.tweak_price()`. Otherwise the cached values are used.

#003 Gas Optimizations

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

We make a non-exhaustive list of potential gas optimizations:

1. In `YBLendingOracleLLFactory`, `usd_oracle` and `asset_oracle` are `HashMap`s. Given that market IDs are assigned sequentially, they could instead be storage arrays with sizes that upper bound the market count, which avoids a `keccak256` invocation for each access.

2. In `YBLendingOracleLLFactory.create_oracles`, `assert market_id < staticcall FACTORY.market_count()` performs an external `staticcall` that is redundant with the following `assert lt != empty(address)` check: in the `Factory` contract, `markets` is backed by a fixed-size array (`Market[MAX_MARKETS]`), so a `market_id` at or beyond `market_count` (but below `MAX_MARKETS`) makes `staticcall FACTORY.markets(market_id)` return a zero-valued `Market`, which is already caught by the `lt != empty(address)` assert; a `market_id` at or beyond `MAX_MARKETS` reverts on the array's own bounds check. The `market_count()` check and staticcall could be dropped.

3. In `YBLendingOracleLLFactory.set_ema_time_market`, `self.asset_oracle[market_id]` is read from storage twice (once for the `extcall` to `set_ema_time`, once for the `SetEmaTime` event), each incurring a `keccak256` invocation to compute the `HashMap` slot. Caching it in a local variable, as is already done for `usd_oracle` in the same function, avoids the redundant computation.

Acknowledged

Item 3 no longer applies: `set_ema_time_market()` was removed, and the new `set_ema_time(market_id, ema_time)` reads `asset_oracle[market_id]` from storage only once. Items 1 and 2 remain unchanged in **Version 2** and **Version 3**.

#008 The Pool's Price Oracle Cannot Fall Below Half of Price_scale

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

`YBLendingOracle` and `YBLendingOracleLL` take the asset price from `pool.price_oracle()`. It enters `asset_price = price_oracle * agg_price` and the ratio `price_oracle / price_scale` that gives `pv_norm`.

The pool computes `price_oracle` as an EMA of its last trade price, clamped to the range from `price_scale / 2` to `2 * price_scale` (`TwocryptoView._price_oracle()` and `Twocrypto.tweak_price()`):

```
min(max(last_prices, unsafe_div(price_scale, 2)), price_scale * 2) * (10**18 - alpha) +
price_oracle * alpha
```

`price_scale` moves in small steps and only when the pool has accumulated enough profit, so it can lag the market. On mainnet, `price_oracle` was up to 24% away from `price_scale` on the live markets.

If the asset price falls to less than half of `price_scale`, `price_oracle` converges to `price_scale / 2` and not to the market price. The oracles then overvalue the asset and the ybLT. For example, with the market at `0.4 * price_scale`, the asset is priced 25% too high. A borrower can then borrow against collateral that is worth less than the oracle reports, which can leave the lending market with bad debt. A rise to more than twice `price_scale` is underestimated in the same way.

> **Acknowledged**
>
> Yield Basis acknowledged the issue and does not see how to solve it well without relying on an external price oracle.

#009 ybLT Price Excludes the Pending Arbitrage Loss and Steps at the Untradable Boundary

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

`YBLendingOracleLL` ("LL") prices ybLT with one of two formulas. `_snapshot()` selects the formula with `x0_ok`, which is true when `AMM.get_x0()` has a solution, that is, when `debt / coll_value` is at most `9/16` (at `L = 2`).

Three different price oracles are involved:

- **AMM price oracle:** `AMM.PRICE_ORACLE_CONTRACT.price()`, the LP token price at the cryptopool's `price_scale` multiplied by the crvUSD aggregator price. The AMM computes `x0` and `coll_value` with it.
- **Pool `price_oracle`:** the cryptopool's own EMA of the asset price in crvUSD. The **market LP price** is the LP token price evaluated at it. `r = pv_norm` is the market LP price divided by the AMM price oracle.
- **Reference oracle:** `YBLendingOracle`, which the client's tests compare LL against.

The **AMM price** is the AMM's marginal price, `(x0 - debt) / collateral`.

The two formulas of LL:

- **Tradable:** the value of the position assuming arbitrage of the AMM towards the market LP price, as far as `AMM.exchange()` allows (`_sqrt_arb_endpoint()`). The LL price is `fundamental * po_shift`. The LT's share accounting uses the same basis, `x0 / (2L - 1)`.
- **Untradable:** `AMM.exchange()` cannot run, so no arbitrage is possible, and the position can be valued by what `LT.emergency_withdraw()` pays. It takes a pro-rata share of the AMM's actual collateral and debt and does not read `x0`. Its payout is the collateral valued at the market LP price minus the debt, the **exit value**, and LL returns it as `balance_price`. This price has no EMA.

LL price versus exit value in the tradable region. `LT.emergency_withdraw()` pays the exit value in the tradable region as well, before any arbitrage. The LL price is lower than the exit value. The difference is the loss the AMM will suffer when arbitrageurs trade the AMM price to the market LP price. LL deducts this loss before the trades happen. The exit value does not. The loss is zero when the AMM price equals the market LP price and increases with the difference between the two. `price_scale` moves in small steps and only when the pool has accumulated enough profit, so it can stay far from `price_oracle`. The AMM price oracle follows `price_scale`, so the market LP price can be far from it for a long time. Arbitrage then stops at the safe-debt limits and the loss stays pending until `price_scale` moves. The loss is small when `price_scale` is close to `price_oracle` and grows as they diverge.

Step at the boundary. Above `debt / coll_value = 9/16` the AMM has no `x0` solution and LL switches formulas. At the boundary the AMM price is `9/16` of the AMM price oracle.

- Below the boundary the LL price is the post-arbitrage value.
- Above it the LL price is the exit value.

The two differ by the pending arbitrage loss. The LL price steps up when entering the untradable region and down when leaving it. In the client's tests the step is of the order of ten percent. There is no step if arbitrage cannot move the AMM at the boundary, or if the equity is zero.

Arbitrage cannot push the AMM past the safe-debt limit `debt / coll_value = 17/32`, which is short of the boundary. The AMM reaches `9/16` when `coll_value` falls without a trade:

- The crvUSD aggregator price falls. A healthy AMM (`debt / coll_value = 1/2`) reaches the boundary at about 0.89, since the AMM price oracle includes that price and debt is counted 1:1.
- `price_scale` moves down after the AMM has been arbitraged to its limit. This lowers the AMM price oracle and `coll_value`. We did not test this route.

For integrators

- On entering the untradable region the LL price rises. This is the less conservative direction, and the new price equals what `emergency_withdraw()` pays.
- On leaving it the LL price drops in one step although the AMM has become tradable again. Positions near the liquidation threshold can become liquidatable at that moment.
- After an abrupt recovery the EMA lags upward, so LL reads below the reference oracle at first and converges over several checkpoints. This is the conservative direction.
- If `price_w()` is not called during the untradable stay, the stale pre-untradable `fundamental` keeps almost full weight, and LL reads well above the reference oracle after recovery (tens of percent in the client's test, which marks this case as an expected failure). LL does not enforce checkpointing. A lending market that calls `price_w()` on borrow and liquidate avoids it.

> **Acknowledged**
>
> Yield Basis considers the untradable region unlikely, as the collateral is already heavily impaired there. The jump can probably be mitigated by setting the AMM fee to a very high value while the AMM is untradable and reducing it slowly after the recovery.

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.