PUBLIC

# Code Assessment

## of the Yield Basis Core Smart Contracts

July 7th, 2025

Produced for

by

CHAINSECURITY

# Contents

# 1 Executive Summary

Dear Yield Basis,

Thank you for trusting us to help Yield Basis with this security audit. Our executive summary provides an overview of subjects covered in our audit of the latest reviewed contracts of Yield Basis Core according to Scope to support you in forming an opinion on their security risks.

Yield Basis implements an Automated Market Maker (AMM) solution that protects liquidity providers from impermanent loss by maintaining constant leverage against Curve Cryptopool LP shares.

The most critical subjects covered in our audit are functional correctness, incentive compatibility, and access control. Functional correctness is high. Incentive compatibility is good but requires attention in certain scenarios, see Rational user will stake tokens during losses recovery. Access control is satisfactory with appropriate privileged role management.

The general subjects covered are event handling, documentation quality, and testing. Documentation is good. Whitepaper is a great help. Test suite would benefit from stateful testing that might help to uncover odd and undesired rounding issues.

In summary, we find that the codebase provides a good 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, the issues uncovered, and how they have been addressed. We are happy to receive questions and feedback to improve our service.

Sincerely yours,

ChainSecurity

## 1.1 Overview of the Findings

Below we provide a brief numerical overview of the findings and how they have been addressed.

| | |
|---|---|
| Critical-Severity Findings | 0 |
| High-Severity Findings | 0 |
| Medium-Severity Findings | 1 |
| • Code Corrected | 1 |
| Low-Severity Findings | 10 |
| • Code Corrected | 10 |

# 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 following code files inside the Yield Basis Core repository:

- `contracts/AMM.vy`
- `contracts/CryptopoolLPOracle.vy`
- `contracts/Factory.vy`
- `contracts/LT.vy`
- `contracts/VirtualPool.vy`

A Whitepaper document was provided as a source of documentation. The table below indicates the code versions relevant to this report and when they were received.

| V | Date | Commit Hash | Note |
|---|---|---|---|
| 1 | 30th April 2025 | de498361f7971b8b01c432093e4ba36c21a4b37d | Initial Version |
| 2 | 16th June 2025 | f6fcf71f94ef19015917469d3f2c7fc1b949ee76 | Version with fixes |
| 3 | 19th June 2025 | fc3ce208ce0813aa57bca06c87703dca8ae5a65e | Version with fixes |
| 4 | 21th June 2025 | 27eb439cb0cd86e4bc3709eca3d7b7536f8b51e5 | Version with fixes |
| 5 | 30th June 2025 | 74dcb46765081ef5170aa0cffcb8925f98cf84b6 | Minor changes |

For the vyper smart contracts, the compiler version `0.4.3` was chosen.

### 2.1.1 Excluded from scope

The following items were not a part of this assessment review:

- Any smart contracts, imports, 3rd party libraries and dependencies not mentioned in Scope section
- The "aggregated" CRVUSD/USD price oracle is not in-scope. The LT staker contract is out-of-scope.
- Cryptopool implementation is not in scope
- Protocol economic modeling and game theory analysis

## 2.2 System Overview

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

Furthermore, in the findings section, we have added a version icon to each of the findings to increase the readability of the report.

Yield Basis offers an Automated Market Maker (AMM) solution that protects liquidity providers from impermanent loss. This is done by borrowing stablecoins against Curve Cryptopool LP shares while keeping constant leverage (of 2 by default).

Following are the tokens that the system will interact with:

- Stablecoin - assumed to be CurveUSD.
- Asset token - Any other ERC20
- Curve Cryptopool LP token - A Stablecoin and Asset twocrypto LP token. This is assumed to be a new Curve StableSwap Twocrypto pool that supports donation functionality. Stablecoin must be token 0 and Asset must be token 1 in this Cryptopool.
- LT token - Leveraged token that keeps a constant leverage relative to Cryptopool LP share.

### 2.2.1 Core formulas

As per the whitepaper, the following formulas define the system bonding curve for swap invariant $I$ at oracle price $p_o$.

$$(x_0(p_o) - d)y = I(p_o)$$

The $x_0$ is a defining AMM state equation that for a given amount of collateral $y$, debt $d$ and target leverage ratio $L$ is:

$$x_0(p_o) = \frac{p_o y + \sqrt{p_o^2 y^2 - 4 p_o y d (\frac{L}{2L-1})^2}}{2 (\frac{L}{2L-1})^2}$$

Value of Leveraged AMM pool $V$, measured in stablecoins, is defined as:

$$V = \frac{x_0}{2L - 1}$$

### 2.2.2 System contracts overview

The following contracts are used as part of the implementation:

**LT**

This is an ERC20 token that represents a leveraged position. It manages:

- User deposits and withdrawals
- Borrowing fees (stablecoin) donation to Cryptopool.
- Revenue split between staked and unstaked LT tokens.
- Admin fees distribution by LT minting
- Stablecoin allocation management.

It is provided an *aggregated* USD oracle which reports the price of CurveUSD in real-world USD. (It is expected to aggregate oracles from stablecoin pools, but this is out-of-scope)

**AMM**

An AMM is coupled one-to-one with an instance of LT. It manages the following aspects on behalf of the LT:

- Account for collateral (Cryptopool LP tokens) and debt (stablecoins) values
- Manages trades between stablecoin and collateral via `exchange()` function
- Tracks interest rate and collects borrowing fees

It uses an instance of **CryptopoolLPOracle** to query the current $p_o$ value.

**Factory**

Factory serves as a control and deployment center for the entire Yield Basis system. It:

- Deploys new `AMM`, `LT` and `CryptopoolLPOracle` contracts for existing Cryptopools.
- Optionally deploys `VirtualPool` and `Staker` contracts for the system features.
- Manages administrative permissions
- Controls fee settings
- Stores and updates implementations of various contracts that are used for new deployments.
- In Version 5 `gauge_controller` address is stored in factory.

**VirtualPool**

`VirtualPool` enables efficient trading using fee-less flash CurveUSD flash loans. It also helps users skip Curve Cryptopool LP deposit/withdrawal and only deal with stablecoins and assets.

**CryptopoolLPOracle**

This is a contract that returns current USD prices for Curve Cryptopool LP tokens. It uses the `lp_price()` method of the pool, and the same CurveUSD aggregated price oracle as **LT**.

### 2.2.3 System Usage scenarios

**Deployment**

A privileged factory admin can call the `Factory.add_market()` function for an existing Curve Cryptopool contract. This action will deploy new `LT`, `AMM` and `CryptopoolLPOracle` contracts using implementation blueprints stored in the `Factory` contract. Since the LT is deployed first, its `set_amm()` method is used to bind it to the AMM. Optionally, `VirtualPool` and `Staker` contracts can be deployed. `add_market()` also sets up newly deployed contracts and allocates stablecoins via the `LT.allocate()` call. Allocated tokens are transferred from `LT.admin` (default is factory) to the `AMM` contract.

**Deposit**

The following sequence happens during this process:

1. User can call `LT.deposit()` to deposit asset tokens and a specified amount of stablecoins to borrow from `AMM`.
2. Deposited asset and borrowed stable tokens are then deposited into Cryptopool in exchange for LP tokens. The receiver of LP tokens is the `AMM` contract.
3. `AMM._deposit()` is then called to account for collateral (LP shares) and debt changes. Limits on debt/collateral ratio are enforced.
4. Based on Core Formulas, the value of the user's deposit is estimated and LT tokens are minted. For the first deposit, 1 `LT` token = 1 $V$ unit of value (in stablecoins). For subsequent deposits, shares are calculated proportionally based on value change.
5. Slippage protection and max AMM debt capacity are also validated.

**Withdraw**

The following sequence happens during this process:

1. User can call `LT.withdraw()` to redeem a specified amount of LT tokens, receiving asset tokens and paying back borrowed stablecoins.
2. LT contract calculates the current AMM value and what fraction of total assets and debts the user needs to be given.
3. LT calls `AMM._withdraw()` with the calculated fraction, which proportionally reduces collateral and debt in the AMM and returns the amounts to be withdrawn.

4. The LP tokens (collateral) are removed from the Cryptopool using `Cryptopool.remove_liquidity_fixed_out()`, so the exact amount of stablecoin to cover the user's debt part is taken.
5. The LT contract burns the redeemed shares from the user's balance.
6. The stablecoin debt is transferred back to the AMM, effectively repaying the borrowed amount.
7. The asset tokens are transferred to the specified receiver address.
8. Slippage protection is enforced via the `min_assets` parameter to prevent excessive value loss during withdrawal.

In emergency situations when the AMM is killed, users can call `LT.emergency_withdraw()` which follows a similar but simplified process using `Cryptopool.remove_liquidity()` instead. If the balanced withdrawal yields not enough stables to cover the user's debt, stables will be transferred from the user.

**Staking**

When LT tokens are transferred to or from the staker contract, `liquidity.staked` and `liquidity.ideal_staked` are recalculated.

Note that the Staker contract itself is out of scope.

**Stablecoin Allocation**

The stablecoins lent to depositors are expected to be uncirculated CurveUSD provided through the factory. By calling `allocate_stablecoins()`, the administrator can adjust the total amount of stablecoins available for lending, also referred to as *debt ceiling*. This also transfers the difference to or from the factory. We assume that governance allocates enough stablecoins for the system not to run out.

Additionally, anyone can call the `allocate_stablecoins()` method without passing a new debt ceiling value. This means that the value will remain the one previously set by the administrator, but the method will still transfer excess stablecoins back to the factory and vice versa.

**Profits and losses split in LT**

There are 3 parties in Yield Basis who profit (or take losses) from system performance: staked tokens, unstaked tokens and admin. All tokens on the balance of the `staker` contract are considered to be staked and the rest are unstaked. Staked tokens forfeit the profits and in general do not earn any yields. In the LT contract, the value in stablecoins is computed as in Core formulas and converted to value in asset tokens using a price oracle. Two storage variables track this converted value of the pool: `total` and `admin`. Each token owns a portion in `liquidity.total`. When `liquidity.total` grows, e.g., due to growth of `Cryptopool.lp_price()` or AMM swap fees, The `current_value - liquidity.total - liquidity.admin` value change is distributed between 3 parties:

1. Admin fee ratio is calculated based on what % of the tokens are staked. [nothing staked, everything staked] -> [10%, 100%].
2. Staked tokens don't get any yield; however, if loss has ever happened, the value of staked tokens is brought back to `ideal_staked`. The `ideal_staked` storage variable tracks the value of staked tokens if no loss ever happened. If value change is negative, staked tokens socialize the loss.
3. Unstaked tokens get the rest of the value change.

Note that value change can be negative in case of losses.

`LT.withdraw_admin_fees()` zeroes `liquidity.admin` and mints the tokens for this value.

**Admin Parameters**

Privileged actors also control the rate charged for borrowing stablecoins and the trading fee for the AMM. The rate can be at most 100% APR and the fee 10% of the output tokens.

### 2.2.4 Assumptions

Yield basis has certain assumptions that are important to highlight:

1. Value of AMM for `exchange()` is measured in USD, while stablecoins received or sent from trader always accounted 1:1 to USD. This is similar to `crvUSD` liquidation/de-liquidation mechanism
2. Leverage of 2 assumes close to 50% LTV for asset. Some risky assets might need lower LTV.
3. `Factory.flash` is assumed to be 0 fee Flash loan contract.

### 2.2.5 Admin fee changes in Version 4

In the new version of the LT pool the admin fee is computed the same way as before, but when it is applied has changed.

A new parameter was added: `MIN_STAKED_FOR_FEES`. This parameter is a threshold below which admin fees are calculated as before. When more than `MIN_STAKED_FOR_FEES` tokens are staked the fee:

1. Not changed if pool suffered losses until all losses are repaid.
2. Once the losses are repaid admin fee applies as usual.
3. Admin fee is not applied on value loss.

Delta between `liquidity.staked` and `liquidity.ideal_staked` variables is used to determine the pool losses.

## 2.3 Trust Model

The following roles are present in the system:

- Factory Admin
  - LT staker address
  - `LT.admin`, but acts as default admin for any Factory-deployed set of contracts itself.
  - Can kill LT contract, thus only allowing `emergency_withdraw()` calls.
  - Can set AMM exchange fee and borrowing rate
  - Since Version 5: Can initialize `gauge_controller` if it was not set during deployment using `set_gauge_controller()`
  - Can change stablecoin allocation given to AMM for borrowing
  - Can donate fees with a high discount rate
  - Trust Level: Fully trusted
- Emergency admin
  - Can kill LT contract, thus only allowing `emergency_withdraw()` calls.
  - Trust Level: partially trusted
- LT Admin
  - Can kill LT contract, thus only allowing `emergency_withdraw()` calls.
  - Can set AMM exchange fee and borrowing rate
  - Can change stablecoin allocation given to AMM for borrowing
  - Can donate fees with a high discount rate
  - Trust Level: Fully trusted

- Users
  - Can deposit assets, withdraw assets, and perform exchanges.
  - They may also trade the underlying cryptopool, look for arbitrage opportunities within the system, stake, unstake, and so on. We only assume that they are financially self-interested.
  - Trust Level: Untrusted

# 3 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.

# 4 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.

# 5 Open Findings

In this section, we describe any open findings. Findings that have been resolved have been moved to the Resolved Findings section. The findings are split into these different categories:

- Design: Architectural shortcomings and design inefficiencies
- Correctness: Mismatches between specification and implementation

Below we provide a numerical overview of the identified findings, split up by their severity.

| | |
|---|---|
| Critical-Severity Findings | 0 |
| High-Severity Findings | 0 |
| Medium-Severity Findings | 0 |
| Low-Severity Findings | 0 |

# 6 Resolved Findings

Here, we list findings that have been resolved during the course of the engagement. Their categories are explained in the Open Findings section.

Below we provide a numerical overview of the identified findings, split up by their severity.

| | |
|---|---|
| Critical-Severity Findings | 0 |
| High-Severity Findings | 0 |
| Medium-Severity Findings | 1 |

- Staker Balance Not Updated Code Corrected

| | |
|---|---|
| Low-Severity Findings | 10 |

- Asset Token Reentrancy Code Corrected
- First Deposit Can Be Less Than MIN_SHARE_REMAINDER Code Corrected
- Staker Should Not Be Fee_Receiver Code Corrected
- Twocrypto Pool Changes Code Corrected
- AMM.fee Is Not Validated During Deployment Code Corrected
- LT.preview_deposit() Diverges From deposit() Code Corrected
- LT.pricePerShare() Reverts on an Empty Pool Code Corrected
- LT.set_admin() Does Not Check ABI of a Contract Code Corrected
- LT Negative Admin Fees Can Be Erased Code Corrected
- VirtualPool Problems Code Corrected

| | |
|---|---|
| Informational Findings | 7 |

- Factory Stablecoin Assumptions Code Corrected
- Full Staked Supply Prevents Recovery Specification Changed
- Gas Optimizations Code Corrected
- Misleading Function Name Code Corrected
- Missing Event on Staker Balance Update Code Corrected
- Missing Nonreentrant Code Corrected
- AMM.get_x0() Comment Inaccuracy Code Corrected

## 6.1 Staker Balance Not Updated

Correctness Medium Version 1 Code Corrected

*CS-YBCORE-001*

Every state modifying function should call `_calculate_values()` and update the total supply and the balance of the staker to reflect a potential token reduction. In `emergency_withdraw()`, the `balanceOf[staker]` is not updated.

**Code corrected:**

An update was introduced.

## 6.2 Asset Token Reentrancy

Correctness | Low | Version 1 | Code Corrected

*CS-YBCORE-002*

In `LT.emergency_withdraw()`, the `ASSET_TOKEN.transfer()` call is performed before the state of the contract is updated, i.e., burning the user's shares and updating `self.liquidity`. This is not an issue with common choices for `ASSET_TOKEN` but can lead to reentrancy attacks draining the contract if the token contract has transfer hooks, such as those in ERC-777.

---

**Code corrected:**

The order of operations was changed to comply with the Check-Effects-Interactions pattern.

## 6.3 First Deposit Can Be Less Than `MIN_SHARE_REMAINDER`

Design | Low | Version 1 | Code Corrected

*CS-YBCORE-003*

When withdrawing from `LT`, users cannot leave dust shares in the pool.

```
assert supply >= MIN_SHARE_REMAINDER + shares or supply == shares, "Remainder too small"
```

However, `deposit()` allows tiny deposits that bypass this limitation and might cause numerical problems.

For example:

1. Initial deposit of 1 wei of value mints 1 wei of share.
    - `LiquidityValuesOut(admin=0, total=1, ideal_staked=0, staked=0, staked_tokens=0, supply_tokens=1)`
2. Underlying Cryptopool experiences `virtual_price` drop:
    - `LiquidityValuesOut(admin=-1, total=1, ideal_staked=0, staked=0, staked_tokens=0, supply_tokens=1)`

Due to rounding up of admin fees (even if `value_change` is negative), a pool that effectively has 0 value keeps 1 as a value, but all loss goes to `self.liquidity.admin`.

Requiring at least `MIN_SHARE_REMAINDER` for the first deposit would protect the pool from such odd states.

---

**Code corrected:**

A check was introduced. At least `MIN_SHARE_REMAINDER` needs to be minted as a first deposit.

## 6.4 Staker Should Not Be Fee_Receiver

Design | Low | Version 1 | Code Corrected

*CS-YBCORE-004*

`LT.withdraw_admin_fees()` modifies balance of staker based on reduction only after `self._mint()` call has occurred. This will result in a wrong balance amount for staker contract.

---

**Code corrected:**

A check was added to `withdraw_admin_fees()`. In addition, a constraint was added to `set_staker()` to prevent such a situation.

## 6.5 Twocrypto Pool Changes

Correctness | Low | Version 1 | Code Corrected

*CS-YBCORE-005*

1. The newest version of Twocrypto with donation has the `donate()` function removed and a flag to `add_liquidity()` added instead.
2. `calc_remove_liquidity()` is not exposed on the pool version with `donate()`. `Twocrypto` has `calc_withdraw_one_coin()` but not this one.

---

**Code corrected:**

`YieldBasis` `LT` now is compatible with the latest `Twocrypto` version.

## 6.6 `AMM.fee` Is Not Validated During Deployment

Correctness | Low | Version 1 | Code Corrected

*CS-YBCORE-006*

When `Factory.add_market()` deploys the `AMM`, no validation is done. On the other hand `AMM.set_fee()` performs `MAX_FEE` check. As a result, an `AMM` with broken `exchange()` function due to > 100% fee can be deployed.

---

**Code corrected:**

A check was introduced to `AMM.__init__()` to match the `set_fee()` constraint.

## 6.7 `LT.preview_deposit()` Diverges From `deposit()`

Design | Low | Version 1 | Code Corrected

*CS-YBCORE-007*

1. `max_debt` checks are skipped in `preview_deposit()`. The checks are enforced in `AMM._deposit()`, which is not called in `preview_deposit()`.

2. In the case when `supply != 0` but `liquidity.total == 0`, `deposit()` treats this as an initial deposit, while `preview_deposit()` will fail.

---

**Code corrected:**

1. `preview_deposit()` will check and revert if `max_debt` is exceeded.
2. Fixed to be compatible with `preview_deposit()`

## 6.8 `LT.pricePerShare()` Reverts on an Empty Pool

Design | Low | Version 1 | Code Corrected

*CS-YBCORE-008*

Due to division by zero, `pricePerShare()` will revert. This is not consistent with a real price of 1 for the first deposit and might break UI integrations.

---

**Code corrected:**

In Version 2, `unsafe_div` is used. It will return 0 instead of reverting.

In Version 4, the function returns $10^{18}$ as a special case when the denominator is zero. This is less surprising for integrations.

## 6.9 `LT.set_admin()` Does Not Check ABI of a Contract

Design | Low | Version 1 | Code Corrected

*CS-YBCORE-009*

If `LT.admin` is a smart contract, it must implement certain methods:

1. `min_admin_fee()` function that returns `uint256`
2. `admin()` function that returns `address`
3. `emergency_admin()` function that returns `address`
4. `fee_receiver()` function that returns `address`

The `set_admin()` setter does not check that the new admin implements these methods. As a result, it is possible to brick `LT` functionality due to a bad update of the admin.

Since the `is_contract` check in vyper checks for `EXTCODESIZE`, with EIP-7702 any EOA can also wrongly bypass this check. If the smart contract to which the account delegates does not define these methods, `LT` will also brick.

---

**Code corrected:**

In `set_admin()` function the required functions are called once to verify ABI compliance.

## 6.10 LT Negative Admin Fees Can Be Erased

Design | Low | Version 1 | Code Corrected

*CS-YBCORE-010*

If the pool experiences a value loss, admin fees might become negative. Until the pool recovers and `self.liquidity.admin > 0`, the admin will not be able to `withdraw_admin_fees()`. However, if a user withdraws `shares` and admin fees are negative, a `shares/supply` portion of the negative admin fees is erased. During deposits, negative admin fees are not scaled up. Thus, anyone can call `deposit()` and `withdraw()` in a loop until the admin fees are lowered.

---

**Code corrected:**

With a new design, the admin does not pay for value loss if `MIN_STAKED_FOR_FEES == 1e16` tokens are staked. In addition, if enough is staked, the admin fees are only charged when all losses are paid off. These two factors limit how big `self.liquidity.admin` might become.

## 6.11 VirtualPool Problems

Design | Low | Version 1 | Code Corrected

*CS-YBCORE-011*

1. `onFlashLoan()` does not check that `msg.sender` is equal to `FACTORY.flash()`. A direct call to this function is possible, while `VirtualPool.exchange()` only permits `FACTORY.flash()`.
2. `VirtualPool.exchange()` allocates a huge `data: Bytes[10**5]` array that does not need to be that big. Vyper allows implicit casting `Byte[10] -> Bytes[100]`.
3. `onFlashLoan()` does not use the `fee` parameter. This assumes 0 flash loan fees. A more generic solution does not require much extra gas or code.

---

**Code corrected:**

Points 1 and 2 were fixed. Point 3 is left unfixed. Non-zero fee flash loans are assumed to be unsupported.

## 6.12 Factory Stablecoin Assumptions

Informational | Version 1 | Code Corrected

*CS-YBCORE-023*

`Factory` does not assert `STABLECOIN.transfer()` and `transferFrom()` are successful. Also it does not use `default_return_value=True`. `AMM` and `LT` don't have such strong requirements and can handle weird ERC20, not only CurveUSD.

---

**Code corrected:**

An assert was added to all `STABLECOIN` calls in `Factory`.

## 6.13 Full Staked Supply Prevents Recovery

Informational | Version 1 | Specification Changed

*CS-YBCORE-012*

In `LT`, if the value of staked tokens has ever dropped, `ideal_staked` will be greater than `staked`. Usually when the `LT` pool value grows, admin gets a fee, unstaked gets the rest of the value change, and `staked` can only recover back to `ideal_staked` value. However, if everything is staked, the admin fee is 100%. Thus, no recovery will happen.

---

**Specification changed:**

In Version 4, the admin fee is no longer taken when the `ideal_staked` is greater than `staked`. Thus, no admin fees will be taken until full recovery. The exception is when too few tokens are staked (<`MIN_STAKED_FOR_FEES`).

## 6.14 Gas Optimizations

Informational | Version 1 | Code Corrected

*CS-YBCORE-013*

We include a non-exhaustive list of potential gas optimizations.

1. `AMM.check_nonreentrant()` could be `@view`.
2. In `AMM.exchange()` on lines 300 and 312, `self.collateral_amount` has already been read into the `collateral` variable, so a storage load can be avoided.
3. `AMM._deposit()` computes `value_before` that is practically unused. This value is a mix of total and admin values that cannot be used directly.
4. `LT.preview_emergency_withdraw()` computes: `max(lv.admin, 0)` in a branch where it is not negative.
5. `LT.emergency_withdraw()` computes: `max(lv.admin, 0)` in a branch where it is not negative.
6. `LT.withdraw()` updates `self.liquidity.total` immediately after `_calculate_values()` and later adjusts it again.

---

**Code corrected:**

All points were addressed

## 6.15 Misleading Function Name

Informational | Version 1 | Code Corrected

*CS-YBCORE-014*

The `AMM.admin_fees()` view function computes the interest collected on lent stablecoins. These fees do not go to the admin, but are donated into the cryptopool. In contrast, `LT.withdraw_admin_fees()` refers to the true admin fee.

---

**Code corrected:**

The function was renamed to `accumulated_interest()`.

## 6.16 Missing Event on Staker Balance Update

Informational | Version 1 | Code Corrected

*CS-YBCORE-015*

When a token reduction is performed, for instance when depositing to the `LT` or staking `LT` tokens, the balance of the staker contract is updated as well as the total supply, but no `Transfer()`, `Mint()`, or `Burn()` event is emitted. This makes it impossible to track the staker balance and total supply off-chain by relying just on events.

---

**Code partially corrected:**

All places were fixed.

## 6.17 Missing Nonreentrant

Informational | Version 1 | Code Corrected

*CS-YBCORE-016*

Unlike neighboring functions, `preview_emergency_withdraw()` allows reentrancy. The function is a view function, so it would be read-only reentrancy. ChainSecurity is not aware of a scenario where this can be exploited.

---

**Code corrected:**

The modifier was added.

## 6.18 `AMM.get_x0()` Comment Inaccuracy

Informational | Version 1 | Code Corrected

*CS-YBCORE-020*

The Safe Limits in the comment suggest that `0 <= debt <= 9/16 coll_value`. The real `MIN_SAFE_DEBT` and `MAX_SAFE_DEBT` depend on the leverage and are `coll_value / 16` and `8.5 * coll_value / 16` for a leverage of 2.

---

**Code corrected:**

A precise comment was added.

# 7 Informational

We utilize this section to point out informational findings that are less severe than issues. These informational issues allow us to point out more theoretical findings. Their explanation hopefully improves the overall understanding of the project's security. Furthermore, we point out findings which are unrelated to security.

## 7.1 `v_loss` Precision Can Be Improved

**Informational** **Version 4**

*CS-YBCORE-024*

In `LT._calculate_values()` `v_loss` is computed as: `min(value_change, v_st_loss * supply // staked)` as 1e18 variable. Later is is used to compute `dv_use_36` - 1e36 based value. In the spirit of other computations `v_loss` can be computed as a 1e36 variable, increasing precision of the truncated division.

## 7.2 Read-only Reentrancy on Token Transfer

**Informational** **Version 1**

*CS-YBCORE-022*

`Twocrypto.add_liquidity()` flow can be summarised as:

1. `_transfer_in()` token 0. Increase `self.balances[0]`
2. `_transfer_in()` token 1. Increase `self.balances[1]`
3. mint LP shares

If token 1 is an ERC777 token, the `LT.preview_emergency_withdraw()` can be executed at step 2. Computed `CRYPTOPOOL.balances(0) / CRYPTOPOOL.totalSupply()` value will be wrong. This problem is an example and other functions in `LT` may be flawed. `CRYPTOPOOL` in general does not protect against invalid state reads. Any use of ERC777 token as an asset or integration of LT with such an asset is discouraged.

## 7.3 Stray Tokens Can Get Stuck in Contracts

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

*CS-YBCORE-017*

System contracts don't have any sweep functions to extract excess tokens from their balances.

E.g. `AMM` tracks Cryptopool LPs via `self.collateral_amount`. However, the actual token balance is never verified. If tokens are sent directly to the contract, the actual balance diverges from the tracked amount. There is no admin function to withdraw excess tokens that exceed `self.collateral_amount`.

## 7.4 Unnecessary Receiver Restriction

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

*CS-YBCORE-018*

The functions `LT.withdraw()` and `LT.emergency_withdraw()` assert that the `receiver` parameter is not equal to the staker contract. Unlike `deposit()` which can mint `LT` shares to the receiver which would pose a risk, these functions simply transfer the asset token to the receiver. Anyone can transfer the asset token using its own transfer function, meaning that the check is easy to bypass. Thankfully, nothing bad happens if the staker has an unexpected balance of asset token.

## 7.5 `AMM.fee` Choice Considerations

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

*CS-YBCORE-019*

When considering an amount for `fee` in `AMM`, the following aspects need to be taken into account:

1. **EMA lag compensation**: The underlying Cryptopool exponential moving average (EMA) oracle always lags behind the real price of the Cryptopool. The EMA price oracle is used to determine the swap invariant for `AMM.exchange()`. The fee needs to be big enough to discourage systematic value extraction.
2. **Price oracle manipulation protection**: A block producer can move the Cryptopool price a lot in a block and produce block N+1 where the price is moved back. The EMA oracle will record an elevated price in that case. While not completely preventing value extraction, the `AMM.fee` can be a way to elevate the cost of such an attack.

---

**Acknowledged:**

Yield Basis responded: Good points of course. Fee is found by maximizing returns in backtesting and appears fairly high.

## 7.6 `max_token_reduction` Computed With Admin Fees

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

*CS-YBCORE-021*

```
max_token_reduction: int256 = abs(value_change * supply //
    (prev_value + value_change + 1) * (10**18 - f_a) // SQRT_MIN_UNSTAKED_FRACTION)
```

Effectively, this is "How many tokens will represent the token value change."

`value_change * (10**18 - f_a)` is effectively how much value `LT` tokens should get. However, `prev_value + value_change + 1` includes the admin fee in the change of tokens. It is not "value of tokens after gain". While this potentially lowers the `max_token_reduction`, the consequences are minimal and can be disregarded.

---

**Acknowledged:**

Yield Basis responded: `max_token_reduction` was never intended to be precise

# 8 Notes

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

## 8.1 Allocation Adjustment Might Hinder AMM Releverage

Note Version 1

Releverage is achieved via `AMM.exchange()`. When a trader sells collateral, `AMM` needs to send stables to the trader. If `LT.allocate_stablecoins()` tries to lower the allocation, the entire `AMM` balance might be transferred back to the allocator. This can hinder the releverage ability of `AMM`.

## 8.2 Rational User Will Stake Tokens During Losses Recovery

Note Version 4

When `liquidity.staked` is lower than `liquidity.ideal_staked`, following is true:

1. All tokens will have their value recovered at the same rate
2. Staked tokens will earn incentives in some other form (governance tokens)

As a result, any rational user will stake their tokens.

## 8.3 `LT.withdraw()` Might Fail on Low Liquidity

Note Version 1

`CRYPTOPOOL.remove_liquidity_fixed_out()` tries to get the full debt for the user position. In the extreme case, the user might be the only LP in the Cryptopool, and the accumulated debt might not be withdrawable directly from the pool. The `emergency_withdraw()` should still work in this case, and the user will pay the debt separately.