STATE MIND

Yield Basis Core

24-02-2025 - 22-05-2025

Table of contents

1. Project brief

Title	Description
Client	Yield Basis
Project name	Yield Basis Core
Timeline	24-02-2025 - 22-05-2025

Project Log

Date	Commit Hash	Note
04-03-2025	6d46e5751482beea6c476d603cc5ceb678266fd1	Initial Commit
11-03-2025	f3b90d719ee0a19c65389a12a643c0eb24584ff1	Commit with fixes
18-03-2025	8a41e77c179628356e2c391a05ea4e6da3160374	Commit with fixes
21-04-2025	ca95f0bc942d633b2bf4e7b5c2ad78e7f15ab293	Reaudit
30-04-2025	70d54eebedd39cfc46dab5407a5807c5626406e5	Commit with reaudit fixes
19-05-2025	2dd3d03104a34d8e617009bb98a2534998066af3	Commit with fixes and muldiv usage in formulas
22-05-2025	4308d9bd1ed8d5728745988249ffff2d06a5f183	Reaudit commit with final fixes

Short Overview

Yield Basis is a new and ambitious protocol that offers a solution to the problem of Impermanent Loss. By using leverage, LPs provide liquidity in a CryptoSwap pool, and the price change of their position occurs linearly with the change in the asset price (unlike the classical provision, where the dependency follows a square root function). To create and maintain leverage, YB uses a specialized leverage AMM, which ensures a constant level of leverage through arbitrage. The system consists of two contracts working in tandem:

- LT is the user-facing contract that liquidity providers will interact with. It implements the deposit and withdrawal flows, as well as an emergency withdrawal mechanism. In the basic scenario, users deposit in the deposit token (BTC, ETH, etc.), and within the function, a stablecoin is borrowed from the leverage AMM. These two tokens are then jointly supplied to a DEX pool. The emergency withdrawal function allows users to withdraw the deposit token in edge cases or when there are imbalances in the pool.

- AMM is the leverage AMM, which holds LP tokens and stablecoins for borrowing. In this system, LP tokens act as collateral for the leveraged positions of liquidity providers, while the debt is represented by the borrowed stablecoin. With a leverage of 2, half of the position (the collateral) covers the debt. The leverage AMM maintains a constant leverage ratio

through arbitrage opportunities. The curve formula follows the classic invariant xy = k, but with an additional dependency on an external oracle.

Project Scope

The audit covered the following files:

- LT.vy
- AMM.vy
- CryptopoolLPOracle.vy

2. Finding severity breakdown

All vulnerabilities discovered during the audit are classified based on their potential severity and have the following classification:

Severity	Description
Critical	Bugs leading to assets theft, fund access locking, or any other loss of funds to be transferred to any party.
High	Bugs that can trigger a contract failure. Further recovery is possible only by manual modification of the contract state or replacement.
Medium	Bugs that can break the intended contract logic or expose it to DoS attacks, but do not cause direct loss of funds.
Informational	Bugs that do not have a significant immediate impact and could be easily fixed.

Based on the feedback received from the Client regarding the list of findings discovered by the Contractor, they are assigned the following statuses:

Status	Description
Fixed	Recommended fixes have been made to the project code and no longer affect its security.
Acknowledged	The Client is aware of the finding. Recommendations for the finding are planned to be resolved in the future.

3. Summary of findings

Severity	# of Findings
Critical	5 (5 fixed, 0 acknowledged)
High	4 (4 fixed, 0 acknowledged)
Medium	14 (13 fixed, 1 acknowledged)
Informational	17 (16 fixed, 1 acknowledged)
Total	40 (38 fixed, 2 acknowledged)

4. Conclusion

During the audit of the codebase, 40 issues were found in total:

- 5 critical severity issues (5 fixed)
- 4 high seveiry issues (4 fixed)
- 14 medium severity issues (13 fixed, 1 acknowledged)
- 17 informational severity issues (16 fixed, 1 acknowledged)

The final reviewed commit is 4308d9bd1ed8d5728745988249ffff2d06a5f183

5. Findings report

CRITICAL-01	Incorrect token_reduction calculation	Fixed at: 8ceb25a

Description

Line: LT.vy#L200

The **LT** contract contains the **_calculate_values()** function in which **token_reduction** is calculated. However, there is an error in the formula for obtaining it.

The current **token_reduction** formula looks like this:

```
token_reduction: int256 = unsafe_div(staked * new_total_value - new_staked_value * total, total - staked)
```

The correct formula looks like this:

$$tokenReduction = \frac{staked * newTotalValue - totalSupply * newStakedValue}{newTotalValue - newStakedValue}$$

Incorrect **token_reduction** calculation leads to incorrect calculation of the YB totalSupply. This leads to incorrect behavior of all functions that use **LT._calculate_values()** function. It also affects the incorrect calculation of withdrawal values.

Recommendation

We recommend bringing the formula to the correct form.

CRITICAL-02	Missing precisions in fee calculation	Fixed at: 2daf4ba

Description

Line: LT.vy#L174

The **LT** contract contains the **_calculate_values()** function in which **f_a** is calculated. However, there is an error in the formula for obtaining it.

The current **f_a** formula looks like this:

```
f_a: int256 = convert(
    10**18 - (10**18 - self.min_admin_fee) * self.sqrt(convert(10**18 - staked // total, uint256)) // 10**18,
    int256)
```

The current formula does not take into account that **staked** value is always less than **total**. This means that **staked // total** value will always be zero.

Incorrect **f_a** calculation leads to incorrect calculation of **admin** and **dv_use** parts. Accordingly, this leads to an incorrect calculation of **new_total_value**, **new_staked_value**, **token_reduction** and YB staker & total values.

Recommendation

We recommend changing the formula like this:

```
f_a: int256 = convert(10**18 - (10**18 - self.min_admin_fee) * self.sqrt(convert(10**36 - staked * 10**36 // total, uint256)) // 10**18, int256)
```

CRITICAL-03	Accumulated interest nullification	Fixed at: 62543da

Description

Lines:

- AMM.vy#L255
- AMM.vy#L266
- AMM.vy#L382

The contract has two functions for tracking current debt: **AMM._debt()** and **AMM._debt_w()**. Calls to **AMM._debt_w()** update **self.rate_time** with the current **block.timestamp**, marking all previously accumulated interest as accounted.
AMM.exchange() function calculates debt with interest via **_debt_w()**, but then modifies **self.debt** directly without applying the accrued interest.
AMM.collect_fees() does not store value, returned by **_debt_w()**, in **self.debt**.
Therefore, accumulated interest can be nullified by performing any exchange or collect_fees operation, causing financial loss to the allocator.
POC: **test_incorrect_debt_update**

Recommendation

We recommend always updating the global **self.debt** with the value returned by **_debt_w()** after each **_debt_w()** call to maintain an accurate accounting of accrued interest.

CRITICAL-04	LT does not account for admin fees	Fixed at: 8a41e77

Description

The total value calculated via **AMM.value_oracle()** contains the total value, including the admin fees.
At the same time, the **LT** contract uses the **LT.liquidity.total** variable to store total value without admin fees. Which leads to a mismatch in calculating user & admin values.
For the **LT._calculate_values()** function, such discrepancy means that **prev_value + dv_use** won't be close enough to the **cur_value**. This, in turn, creates an artificial difference in values, increasing/decreasing both user value and admin fees for the next invocations with no yield (one could invoke transfers to the staker with 0 shares).
This also creates inconsistencies in the deposits/withdrawals, as shares/output values are dependent on the value without accounting for the admin fees.
The **LT.withdraw()** implementation calculates the output value from AMM as a **shares/totalSupply** proportion. If this is expressed in terms of the values, then the fraction is calculated as **token_value/liquidity.total**, hence users will withdraw the admin portion of value.
However, to calculate the actual proportion, it is necessary to take into account the admin fees. This proportion will look like this: **token_value/(admin_fees + liquidity.total)**.

Recommendation

We recommend deducting the admin fees from the total value of the **AMM**.

CRITICAL-05	Unbounded max_token_reduction	Fixed at: 35a14e7

Description

Line: LT.vy#L305

The **max_token_reduction** formula does not include the **staked** variable. As the comment states, the maximum should be enforced only when **eps = (supply – staked) / supply < 1e-8**.

Unbounded **max_token_reduction** forces losses on all users by minting shares to staker, even when there is no stake.

For example:

1. **staked = 0**, hence **token_reduction = 0**.
2. **value_change < 0**, then **max_token_reduction < 0**.
3. **token_reduction = min(token_reduction, max_token_reduction) = max_token_reduction**.

Recommendation

We recommend applying **max_token_reduction** when the **eps < 1e-8** condition is met.

HIGH-01	Griefing LT.distrubute_borrower_fees()	Fixed at: e31555c

Description

Line: AMM.vy#L378

AMM.collect_fees() function doesn't have access control. An attacker can front-run an admin's transaction, which calls **LT.distrubute_borrower_fees()**. Thus, the fees will be stuck on the LT.

Recommendation

We recommend adding only **AMM.DEPOSITOR()** access control to **AMM.collect_fees()** function.

Client's comments

> Instead of access control, used **STABLECOIN.balanceOf(self)**. This allows to also donate something to LT to increase the value used to boost the pool if necessary. We normally do not store and stablecoins in LT

HIGH-02	Precision error can infinitely increase token reduction	Fixed at: 135f993

Description

The **LT._calculate_values()** calculates the **token_reduction** variable that is meant to reduce the staker shares to keep their value constant.

```
token_reduction: int256 = unsafe_div(staked * new_total_value - new_staked_value * supply, new_total_value - new_staked_value)
```

The **new_total_value** changes faster than the **new_staked_value**, but can be very close if the staked shares amount is almost equal to the total supply. The denominator **new_total_value - new_staked_value** value can become very small, but not zero.
The precision error arises in the **d_staked_value** calculation.

```
elif _to == staker:
    # Increase the staked part
    d_staked_value: uint256 = liquidity.total * _value // liquidity.supply_tokens
```

This formula rounds down the **d_staked_value**, which leads to cases when **staked_value != total_value** for **staked = totalSupply**. But we can't round up in this case, as the same issue will arise if one stakes all his shares minus 1.
The scenario is possible if two users decide to deposit into the **LT** and after some time they stake their shares.
Or, if the first (and only) depositor already staked all their shares, an arbitrary user can force the denominator to +-1 (with as small as possible input collateral), which leads to **token_reduction >= totalSupply**, which, essentially, permanently breaks the contract.

Recommendation

We recommend depositing a small amount of collateral during deployment to avoid cases when **staked_value ~ total_value**.

Client's comments

> Added limits on token_reduction in calculate_values. The issue appears because unstaked APR goes to infinity which means that we need to burn "staked" shares too fast to keep up with that. That singularity is not good, so worked around that when too small fraction is unstaked (effectively that'd limit the unstaked APR to not go to infinity)

HIGH-03	Admin fees in the killed state	Fixed at: 70d54ee

Description

Line: LT.vy#L672

The **LT.withdraw_admin_fees()** function is callable when **AMM** is killed, which creates ambiguity around the killed state:

1. **LT.emergency_withdraw()** already handles admin fees.
2. Normally **LT._calculate_values()** can not be invoked during the pause.

This also allows minting tokens in the killed state, potentially DOSing **LT.emergency_withdraw()** for the last user.

Recommendation

We recommend restricting admin withdrawals during the killed state.

HIGH-04	Insufficient precision in value deltas	Fixed at: 6220f1f 2dd3d03

Description

Line: LT.vy#L274

The token reduction formula is introduced to maintain the staker value constant despite a positive yield.

Intuitively, a negative value change should not trigger token reduction, although in reality, **LT** will produce negative **token_reduction**, which slightly increases unstaked losses.

If we look at the token reduction formula:

1. Let **token_reduction < 0**;
2. **new_total_value - new_staked_value >= 0** is always true;
3. **staked * new_total_value - new_staked_value * supply < 0** =>
4. **new_staked_value > new_total_value * staked / supply**.

By design, the staked value is at most a proportion of the total (**ideal_staked**); hence, condition **4.** must become true due to precision errors.

The closer **staked** shares to **supply**, the bigger fee admin takes: **(10**18 – f_a)** -> **0**. Which affects the precision of the **dv_use = value_change * (10**18 – f_a) // 10**18** variable.

The **dv_s = dv_use * staked // supply** formula rounds towards zero, which makes condition **4.** true if **value_change < 0**.

We discovered, that by increasing the precision of **dv_use** the **token_reduction** values are more predictable, reducing the need/complexity of **max_token_reduction**.

Recommendation

We recommend increasing the precision of the delta and value calculations.

```
dv_use: int256 = value_change * (10**18 - f_a)
prev.admin += (value_change * 10**18 - dv_use) // 10**18
...
new_total_value: int256 = max(prev_value * 10**18 + dv_use, 0)
new_staked_value: int256 = max(v_st * 10**18 + dv_s, 0)
...
total=convert(new_total_value // 10**18, uint256),
staked=convert(new_staked_value // 10**18, uint256),
```

MEDIUM-01	LT.preview_deposit() inaccuracy in calculating shares	Acknowledged

Description

Lines:

- LT.vy#L228
- AMM.vy#L338-L339

CurveCryptoPool.calc_token_amount() calculates LP amount, but does not simulate adding asset, debt to the pool reserves LT.vy#L228. Virtual reserves are calculated at the old price(excluding asset, debt) AMM.vy#L338-L339, which leads to inaccuracies in the calculation of the share.

CurveCryptoPool.add_liquidity() calculates LP and adds asset, debt to the pool reserves LT.vy#L305. Virtual reserves are calculated at the updated price(taking into account asset, debt) AMM.vy#L281, AMM.vy#L295.

LP call preview deposit:

```
assets=1000000000000000000, debt=1000000000000000000000000
totalSupply = 0
lp_tokens=316224603739177764819

p_o=63245532033675866398 - the oracle price does not take into account deposited funds
value=99997999979999000140450 - virtual reserves are calculated based on the old price
collateral_price=100000000000000000000000
deposit_amount=999979999799990001
```

LP call deposit:

```
assets=1000000000000000000, debt=1000000000000000000000000
totalSupply = 0
lp_tokens=316224603739177764819

p_o=63245869432714750213 5 - the price changed after taking into account the addition of the deposited amount
value_after=99998999989999800192291 - virtual reserves are calculated based on the updated price
collateral_price=100000000000000000000000
deposit_amount=999989999899998001
`
```

Recommendation

We recommend adding a function that provides a price that takes into account the amount to be deposited into the **CurveCryptoPool** contract.

Client's comments

The difference in p_o is rather small: it can be due to depositor's own fees earned if out deposit is asymmetric. This has really minor effect, but fairly complicated to fix because this would involve simulating cryptopool logic to both calculate fees and do **tweak_price()** after. Doesn't look feasible for the minor effect here, especially considering that ideal deposit to do is almost symmetric anyway

MEDIUM-02	LT.preview_deposit() incorrect calculation if the full stake	Fixed at: 0b1fc0b

Description

Lines: LT.vy#L231-L232

In the case when all deposits were at staked or minted on the balance of **LT.staker()**, the contract calculates the shares based on the amount deposited LT.vy#L236-L237, but this is correct only for the first deposit.

Let's look at the example of **LT._calculate_values()**

```
staked == total - all deposits staked to staker

token_reduction = N
staked_tokens=staked - token_reduction
supply_tokens=total - token_reduction
```

Recommendation

We recommend calculating the shares based on the delta **v** and removing the condition LT.vy#L232.

MEDIUM-03	Staker can transfer shares to himself	Fixed at: de858a8

Description

Lines: LT.vy#L500-L520

LT._transfer() function allows users to send shares to other accounts. This function has the following structure:

```
if staker in [_from, _to]:
    if _from == staker:
        ...
    else:
        ...
self.balanceOf[_from] -= _value
self.balanceOf[_to] += _value
```

In case if the staker will send shares to himself, only first **if** will be executed that reduces the staked part of the sender. It allows the staker to reduce **liquidity.staked** variable to zero, which makes **token_reduction** inside **LT._calculate_values()** also zero and increases **supply_tokens** for everyone.

Recommendation

We recommend prohibiting sending tokens when **address to == address from**.

Client's comments

Staker is going to be a staker contract which would not be able to do it. Probably not an issue, but good to check, so included this check inside the condition

MEDIUM-04	Accumulated interest nullification by set_rate	Fixed at: b2dde57

Description

Lines: AMM.vy#L152-L164

The **AMM.set_rate()** function misses saving accumulated interest, along with overwriting **self.rate_time**, which leads to the reset of accumulated fees.

POC: **test_set_rate_debt_nullification**

Recommendation

We recommend updating **self.debt** with accumulated fees in **AMM.set_rate()** function.

```
@external
@nonreentrant
def set_rate(rate: uint256) -> uint256:
    assert msg.sender == DEPOSITOR, "Access"
    debt: uint256 = self._debt_w()
    rate_mul: uint256 = self.rate_mul
    self.rate = rate
    self.debt = debt
    log SetRate(rate=rate, rate_mul=rate_mul, time=block.timestamp)
    return rate_mul
```

MEDIUM-05	Depositing to the staker address	Fixed at: 11c306f

Description

Line: LT.vy#L257

The **LT.deposit()** function allows users to provide a receiver address.

One could provide **receiver = staker**, which bypasses all the logic defined in the **LT._transfer()** function.

Generally, such action would simply donate minted LPs to shareholders due to token reduction, but there are corner cases if **LT** is empty:

1. Bob is attempting to deposit to the **LT** contract.
2. Alice front-runs Bob's tx and deposits to the staked address, which keeps **ideal_staked = 0** & **staked = 0** but increases the **staker** balance.
3. Bob's deposit tx will invoke **LT._calculate_values()** and calculate token reduction:

```
token_reduction: int256 = unsafe_div(staked * new_total_value - new_staked_value * total, new_total_value - new_staked_value)
```

Where **new_staked_value = 0**, hence **token_reduction = balanceOf(staker) = self.totalSupply**.

4. Once tx is outside of the **LT._calculate_values()**, the **supply** variable will hold the previous value. It will enter a conditional block, where it will use the new **totalSupply = 0**, which results in 0 shares minted for Bob.

```
supply: uint256 = self.totalSupply
...
liquidity_values: LiquidityValuesOut = empty(LiquidityValuesOut)
if supply > 0:
    liquidity_values = self._calculate_values()
...
if supply > 0:
    supply = liquidity_values.supply_tokens # = 0
    ...
    shares = supply * v.value_after // v.value_before - supply # = 0
```

5. Alice deposits her own assets to take control of Bob's deposit value.

This scenario will only work if someone deposits with **min_shares = 0**, this can be used to DOS the contract until someone decides to donate his funds. Thus, an attacker can constantly front-run valid deposits to DOS, the cost of DOS is equal to the cost of unblocking the contract.

Recommendation

We recommend restricting direct deposits to the staker address.

MEDIUM-06	AMM.get_dy isn't limited by AMM debt amount	Fixed at: e97c5f4

Description

AMM.get_dy() can be used by external contracts to calculate the **amount_out**.
If **i=0**, **j=1** and **amount_in > AMM.get_debt()** then **AMM.exchange()** revert AMM.vy#L256, because a pool can sell collateral only for the amount of debt.
However, the **AMM.get_dy()** function isn't limited by AMM debt amount and returns incorrect value.
Let's look at the example:

AMM.get_debt() = 100000000000000000000000

External contract call AMM.get_dy(0, 1, 200000000000000000000000)

amount_in = 200000000000000000000000 - amount_in X2 debt
collateral = 316224603739177764819
x = 3999969999699994005768 75
y = 158111116006572139345
amount_out 157006693318477386095 - incorrect amount

Recommendation

We recommend adding the check**in_amount <= AMM.get_debt()** for this case.

MEDIUM-07	DOS when trying to repay % on debt	Fixed at: fbf682e

Description

Lines: AMM.vy#L396-L398
AMM.collect_fees() calculates fees amount based on the accumulated % on the debt AMM.vy#L393. Debt increases over time. If the AMM has not earned enough fees, then an attempt to transfer fees reverts due to an insufficient balance of stablecoins on the AMM contract.
Thus, the admin can't partially compensate % of the debt.

Recommendation

We recommend limiting the collected amount by the available stablecoin balance in the **AMM.collect_fees()** function.

Client's comments

Checking balanceOf to not revert at the fee claim

MEDIUM-08	Incorrect staker migration process	Fixed at: f797e30

Description

Lines: LT.vy#L416-L421

LT.set_staker() function allows to set a new staker address. This function does not take into account existing shares of the new staker, which may influence **total_supply** and **LT.pricePerShare()** significantly. When admin calls **LT.set_staker()**, users can frontrun this transaction and send shares directly to future staker using **LT.transfer()** functionality to influence **LT.pricePerShare()** and staked value without using intended **LT.transfer()** code parts for sending shares to the staker. The attached test demonstrates that **LT.pricePerShare()** increases if the user will frontrun **LT.set_staker()** call and send shares to a future staker compared to the situation when shares are transferred as intended after **LT.set_staker()** call. The **ideal_staked** variable is not considered during the migration process.

POC: **test_change_staker**

Recommendation

We recommend implementing a safe migration process. Existing shares of the new staker can be sent to admin to avoid incorrect staked value calculations. It is also advisable to allow the staker to be set only once.

MEDIUM-09	Staker can call **LT.withdraw()** function	Fixed at: 084c208

Description

Line: LT.vy#L309

The **LT.withdraw()** function allows any LT shares holder to call themselves.

However, **liquidity.staked** and **liquidity.ideal_staked** are not recalculated. If the staker calls the withdrawal function, he will receive his funds but leave behind an incorrect state. This leads to undefined behavior of the contract and violation of its logic.

Recommendation

We recommend limiting the **LT.withdraw()** function call for the staker.

MEDIUM-10	pricePerShare inflation	Fixed at: 8cbf5b9

Description

Line: LT.vy#L393

pricePerShare is calculated as the ratio of **liquidity.total** to **liquidity.supply_tokens** — essentially, it represents the share of the AMM value that belongs to users (excluding admin value), relative to the total supply. In the absence of admin fees, users can fully withdraw all value from the AMM or leave a very small portion behind.

If a tiny number of shares is left, for example, several weis — the low values and limited precision can allow **pricePerShare** to be manipulated. Such a price shift can occur during a withdrawal process when a user leaves just 1 share. Since the **pool.remove_liquidity_fixed_out()** function also takes small fees, those are treated as yield and will be reflected in the next **LT._calculate_values()** call. The situation becomes significantly worse if an admin fee is set in the **LT** or if **liquidity.admin** is much larger than **liquidity.total**. In such a case, an attacker, right after creating a new YB market, can deposit the **LT**, then perform a series of swaps in the crypto pool to drive up the LP token price and accumulate yield for both **liquidity.total** and **liquidity.admin**.

After that, the attacker can call **LT.withdraw()** and leave only 1 share in the LT. Since the **AMM** still holds a large admin value, which greatly exceeds the remaining **liquidity.total**, the impact of the fees collected during **pool.remove_liquidity_fixed_out()** will be much more significant (as larger values suffer less from losses due to integer rounding). As a result, the **pricePerShare** will increase substantially.

PoC: **test_inflation**

Recommendation

We recommend calculating share price considering **liquidity.admin** value. For that, shares can be virtually minted to the admin in case of positive yield, so the formula for **pricePerShare**:

```
@external
@view
def pricePerShare() -> uint256:
    v: LiquidityValuesOut = self._calculate_values(self._price_oracle())
    return (v.total + v.admin) * 10**18 // (v.supply_tokens + v.admin_tokens)
```

Additionally, to avoid corner cases when working with very small values, we recommend performing a pre-mint so that both the **LT** and the **AMM** always have non-zero values (a modified version of the **DEAD_SHARES** logic).

Client's comments

> To prevent pumping the price per share, I disallowed withdrawals which leave only a tiny nonzero fraction in the AMM. This still allows to deposit dust to prevent full withdrawal for the last person ("griefing attack"), but it's easy to solve by a small deposit to unblock the last depositor. And it is better than allowing to pump the price per share.

MEDIUM-11	Default return value and assertions	Fixed at: b0e9d4a

Description

Some external calls are made without **default_return_value=True**. It is essential as some tokens do not return boolean values.

1. In the **LT.deposit()** the **STABLECOIN.transferFrom()** and **DEPOSITED_TOKEN.transferFrom()**;
2. In the **LT.withdraw()** the **STABLECOIN.transfer()** and the **DEPOSITED_TOKEN.transfer()**;
3. In the **AMM.__init__()** the **stablecoin.approve()** and **collateral.approve()**.

There are also missing assertions:

1. In the **LT.allocate_stablecoins()**;
2. In the **AMM.collect_fees()**.
3. In the **AMM.__init__()**.

Although stablecoin is meant to be crvUSD, other tokens are possible.

Recommendation

We recommend adding missing assertions and the **default_return_value=True** parameter to token calls.

MEDIUM-12	Pricing value in stablecoin instead of fiat	Fixed at: e61318b

Description

Lines:

- CryptopoolLPOracle.vy#L35
- CryptopoolLPOracle.vy#L40
- LT.vy#L169
- LT.vy#L237
- LT.vy#L329

The **CryptopoolLPOracle** is used in the **LevAMM** contract to provide the current price of the pool's LP tokens in crvUSD, treating it as fiat USD value. The same issue arises in the **LT** contract's internal pool pricing logic **LT._calculate_values()**. This will better reflect the position's value as well as enable an additional peg to the crvUSD.

Recommendation

We recommend converting values to fiat USD.

MEDIUM-13	First depositor can break contract	Fixed at: 9625570 aed08ae

Description

Line: LT.vy#L303

The **token_reduction** calculation includes division by total and staked values difference **(new_total_value - new_staked_value)**.

The first depositor can grief the newly deployed **LT** contract by depositing and staking their shares.

The **_calculate_values()** function invocation will always revert, permanently breaking further deposits.

Recommendation

We recommend introducing a limit for staking or setting a staker address sometime after the contracts are deployed and deposits are made.

MEDIUM-14	Incorrect application of max_token_reduction	Fixed at: 35a14e7

Description

Line: LT.vy#L313

Current implementation of **max_token_reduction** limits **token_reduction** as following:

```
token_reduction = min(token_reduction, max_token_reduction)
```

This approach is incorrect for negative values. When dealing with negative values, using **min()** will select the more negative value (smaller in absolute terms), which contradicts the intended limiting behavior.

Recommendation

We recommend limiting **token_reduction** to the minimum absolute value, while keeping **token_reduction** sign.

INFORMATIONAL-01	Lack of the LT.min_admin_fee setter	Fixed at: 30bf611

Description

The **LT** contract has **LT.min_admin_fee** parameter. This parameter is not set during initialization and does not have a setter function. In the current implementation the **LT.min_admin_fee** is always zero.

Recommendation

We recommend adding a setter function for this value.

INFORMATIONAL-02	Outdated debt value exposed via public variable	Fixed at: 2e57d3b

Description

Line: AMM.vy#L51

AMM contract exposes debt via the public variable:

```
debt: public(uint256)
```

However, this value is always outdated as it doesn't account for accumulated interest since the contract was last used, leading to potential inaccuracies if external contracts rely on it.

Recommendation

We recommend setting **debt** as private to prevent such cases and ensure external contracts rely on **AMM.get_debt()** for accurate debt retrieval.

INFORMATIONAL-03	LT admin fee collection	Fixed at: 6aaaf2e

Description

Currently, there is no method for admin fee collection in the **LT** contract.
An admin may share losses with all the other users, so the fee value may be negative.
In a scenario, where the next state update makes a total admin fee negative, an admin is incentivized to front-run it to take his fees before an update deducts them.

Recommendation

We recommend implementing the admin fee collection function and invoking the **LT._calculate_values()** before all the state-altering actions.

INFORMATIONAL-04	ERC20.decimals() incorrect interface	Acknowledged

Description

Lines:

- AMM.vy#L10
- LT.vy#L10
- LT.vy#L33

According to the standard **ERC20.decimals() -> uint8** IERC20Detailed.vyi#L17.

Recommendation

We recommend fixing the interface following the standard.

Client's comments

> It is deliberately uint256 in the code, to avoid type conversion. Boudns are checked when we calculate **18 - decimals**

INFORMATIONAL-05	Optional debt breaks the LT.preview_deposit()	Fixed at: 991f057

Description

Lines:

- LT.vy#L226
- AMM.vy#L125

The **LT.preview deposit()** function has an optional **debt** parameter which defaults to **max_value(uint256)**.
However, the transfer of such a parameter completely breaks the logic of calculating the value of the **AMM** contract.
With a large value of the debt in the **AMM.get_x0()** function, the discriminant of the quadratic equation will be less than zero and the function will return an error when trying to calculate the square root.

Recommendation

We recommend implementing logic for the **debt==max_value** case, in which the contract calculates the best possible debt.

Client's comments

> Since **deposit()** has the **debt** argument necessary, I simply removed the default value

INFORMATIONAL-06	Misspelled function name	Fixed at: 3c0b8b3

Description

Line: LT.vy#L408

The function **LT.distrubute_borrower_fees()** has the word **distribute** misspelled.

Recommendation

We recommend fixing the misspelling.

INFORMATIONAL-07	LT.preview_withdraw() and LT.withdraw() produce mismatched amounts	Fixed at: c010116

Description

Lines:

- LT.vy#L273
- LT.vy#L351

LT.preview_withdraw() produces a slightly higher withdrawal amount than **LT.withdraw()**:

```
Preview withdraw: 99996995043667078856
Assets withdrawn: 99996995043667078792
Delta: 64
```

This discrepancy is caused by rounding differences in **LevAMM._withdraw()**. The error magnitude, while small, increases proportionally with the withdrawal amount.

Recommendation

We recommend aligning the calculations in both **LT.preview_withdraw()** and **LT.withdraw()** to ensure consistent results.

INFORMATIONAL-08	Unused state variable	Fixed at: cf575bd

Description

Line: LT.vy#L370

The **LT.withdraw()** function contains an unused **state** variable.

Recommendation

We recommend deleting an unnecessary variable.

INFORMATIONAL-09	Disallow CryptoPools with more than 2 tokens	Fixed at: 559ed69

Description

Line: LT.vy#L170

The contract does not restrict pools to two tokens, which will break the share calculation logic if more are present.

Recommendation

We recommend adding a check to ensure the pool contains exactly two tokens.

INFORMATIONAL-10	Disallowing swaps in an empty AMM	Fixed at: a40fcdd

Description

Line: AMM.vy#L251

Safe limits prevent excessive destabilization of the AMM, but when the **AMM** is empty, with both collateral and debt equal to zero, it's still possible to execute a trade. This can happen when selling the collateral token into the AMM. If the swap amount and the oracle price are small, the limit checks can be bypassed because the calculated **coll_value** at Line 139 will be zeroed, and safe limit checks will be bypassed.

Recommendation

We recommend disallowing swaps in empty **AMM** for state consistency.

INFORMATIONAL-11	Front-running of allocation decrease	Fixed at: 35ce293

Description

Line: LT.vy#L448

The deposit logic in the **LT** contract allows the creation of a position using only borrowed stablecoins. This doesn't harm other LPs — in fact, it is even beneficial for them — but it enables transferring free stablecoins from the **AMM** to the crypto pool. This property can be exploited to front-run allocation reductions in the **LT**.

Recommendation

We recommended sending transactions with allocation decrease via private mempools or checking the allocation limit during deposit.

Client's comments

Made the method which withdraws the rest of allocation callble by anyone (not just admin). If someone front-runs setting the allocation by the admin, anyone can submit a private mempool tx without requiring a governance process subsequently.
In addition, added a public method to check whether there's a reentrancy in the AMM: it's needed here because we don't call the AMM but directly transfer to/from there, and doing so during an ongoing transfer would be not good.

INFORMATIONAL-12	Missing sanity checks in LT.set_amm()	Fixed at: 1b3ea6b

Description

Line: LT.vy#L403

The **LT.set_amm()** function lacks validation that ensures the AMM contract's tokens match the LT contract's tokens, creating potential inconsistencies.

Recommendation

We recommend adding token-matching validation checks.

INFORMATIONAL-13	Preventing the AMM value from being zero when totalSupply in the LT is positive	Fixed at: 0ba1276

Description

If the **AMM** holds a small amount of value or if LT holders experience prolonged negative APR, which can result from high borrowing rates or very frequent rebalances, this situation may occur. Given the current liquidity levels in crypto pools and the launch of LT for blue-chip tokens, it's an unlikely scenario, but the issue can still arise.
In such a case, when liquidity is almost entirely withdrawn (with only a few wei remaining in the **LT**), the value in the **AMM** can drop to zero. This would block future deposits: with **totalSupply > 0**, the minted shares would be calculated based on **value_after** and **value_before**, leading to a division by zero.

Recommendation

We recommend resetting **totalSupply** when the **AMM** value reaches zero, or using **DEAD_SHARES** mechanics, to ensure the **AMM** always holds a non-zero value.

INFORMATIONAL-14	fee_receiver sanity check	Fixed at: 75bf0d8

Description

Line: LT.vy#L672
The **LT.withdraw_admin_fees()** function mints shares directly to **fee_receiver**.
Minting shares to the staker address creates inconsistencies as it bypasses relevant accountings.

Recommendation

We recommend introducing a sanity check **fee_receiver != staker**.

INFORMATIONAL-15	Add event for AMM.set_killed()	Fixed at: 8b05ee9

Description

Line: AMM.vy#L462
The admin can change the **is_killed** variable, but no event is emitted to record the change, making off-chain monitoring more difficult.

Recommendation

We recommend emitting an event with the new value.

INFORMATIONAL-16	Naming inconsistency	Fixed at: 3b01f01

Description

Line: LT.vy#L183
Comment in **LT.vy** constructor uses **deposit_token**, but the input parameter is named **asset_token**.

Recommendation

We recommend changing **deposit_token = WBTC** to **asset_token = WBTC** to match the function parameter name.

INFORMATIONAL-17	Redundant variable initialization	Fixed at: d444506

Description

Lines:

- LT.vy#L301
- LT.vy#L509
- LT.vy#L557

1. Variable **token_reduction** in **LT._calculate_values()** function is initialized to 0 and then immediately reassigned on the next line. This initialization is unnecessary and can be removed.
2. **max(lv.admin, 0)** operation is redundant because code only executes when **lv.admin > 0**.

Recommendation

We recommend removing redundant initializations and operations.

STATE
MIND