Pashov Audit Group

YieldBasis Security Review

March 26th 2025 - April 1st 2025

Contents

1. About Pashov Audit Group

Pashov Audit Group consists of 40+ freelance security researchers, who are well proven in the space - most have earned over $100k in public contest rewards, are multi-time champions or have truly excelled in audits with us. We only work with proven and motivated talent.

With over 300 security audits completed — uncovering and helping patch thousands of vulnerabilities — the group strives to create the absolute very best audit journey possible. While 100% security is never possible to guarantee, we do guarantee you our team's best efforts for your project.

Check out our previous work here or reach out on Twitter @pashovkrum.

2. Disclaimer

A smart contract security review can never verify the complete absence of vulnerabilities. This is a time, resource and expertise bound effort where we try to find as many vulnerabilities as possible. We can not guarantee 100% security after the review or even if the review will find any problems with your smart contracts. Subsequent security reviews, bug bounty programs and on-chain monitoring are strongly recommended.

3. Risk Classification

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

Impact

- **High** - leads to a significant material loss of assets in the protocol or significantly harms a group of users
- **Medium** - leads to a moderate material loss of assets in the protocol or moderately harms a group of users
- **Low** - leads to a minor material loss of assets in the protocol or harms a small group of users

Likelihood

- **High** - attack path is possible with reasonable assumptions that mimic on-chain conditions, and the cost of the attack is relatively low compared to the amount of funds that can be stolen or lost
- **Medium** - only a conditionally incentivized attack vector, but still relatively likely
- **Low** - has too many or too unlikely assumptions or requires a significant stake by the attacker with little or no incentive

4. About YieldBasis

YieldBasis aims to eliminate impermanent loss by leveraging liquidity positions such that their value tracks the underlying asset, while still earning trading fees. By dynamically adjusting leverage within Curve-style AMMs, the approach achieves sustainable yield while closely tracking the price of the underlying asset.

5. Executive Summary

A time-boxed security review of the **yield-basis/yb-core** repository was done by Pashov Audit Group, during which Pashov Audit Group engaged to review **YieldBasis**. A total of **13** issues were uncovered.

Protocol Summary	
Project Name	YieldBasis
Protocol Type	Yield Optimizer
Timeline	March 26th 2025 - April 1st 2025

Review commit hash:

- d29f47000c80851bb6c4ad92463b8ddb48cad944 (yield-basis/yb-core)

Fixes review commit hash:

- d9b8eebf84b2bca4b761a86080fe381ffff6a0ba (yield-basis/yb-core)

Scope

`LT.vy` `AMM.vy` `Factory.vy`

6. Findings

Findings count

Severity	Amount
Medium	9
Low	4
Total findings	**13**

Summary of findings

ID	Title	Severity	Status
[M-01]	`new_total_value` is assumed to be non-negative	Medium	Resolved
[M-02]	`withdraw` does not reduce `staked` data when staker is caller	Medium	Resolved
[M-03]	Token rebase miscalculation during position losses	Medium	Resolved
[M-04]	State not updated when staker address changes	Medium	Resolved
[M-05]	Rebase bypass possible through `_transfer()`	Medium	Resolved
[M-06]	`set_rate()` resets accrued fees causing fee loss	Medium	Resolved
[M-07]	Token miscalculation in `withdraw_admin_fees()` inflates shares	Medium	Resolved
[M-08]	Incorrect total supply calculated during admin fee withdrawal	Medium	Resolved
[M-09]	`set_allocator()` state update missing causes incorrect balances	Medium	Resolved
[L-01]	Staker contract missing in `LT` contract post-creation	Low	Resolved
[L-02]	`min_admin_fee` lacks initialization and update	Low	Resolved
[L-03]	`deposit` fails to account for cases where `value_before` equals 0	Low	Resolved
[L-04]	`fill_staker_vpool()` fails without address setters	Low	Resolved

Medium findings

[M-01] `new_total_value` is assumed to be non-negative

Severity

Impact: High

Likelihood: Low

Description

When `_calculate_values` is performed, it will calculate `new_total_value` based on the `value_change`, and will return it as `total` where it is converted to `uint256`.

```
@internal
@view
def _calculate_values(p_o: uint256) -> LiquidityValuesOut:
    prev: LiquidityValues = self.liquidity
    staker: address = self.staker
    staked: int256 = 0
    if staker != empty(address):
        staked = convert(self.balanceOf[self.staker], int256)
    supply: int256 = convert(self.totalSupply, int256)

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

    cur_value: int256 = convert((staticcall self.amm.value_oracle()).value * 10**18 // p_o,
int256)
    prev_value: int256 = convert(prev.total, int256)
>>> value_change: int256 = cur_value - (prev_value + prev.admin)

    v_st: int256 = convert(prev.staked, int256)
    v_st_ideal: int256 = convert(prev.ideal_staked, int256)
    # ideal_staked is set when some tokens are transferred to staker address

>>> dv_use: int256 = value_change * (10**18 - f_a) // 10**18
    prev.admin += (value_change - dv_use)

    dv_s: int256 = dv_use * staked // supply
    if dv_use > 0:
        dv_s = min(dv_s, max(v_st_ideal - v_st, 0))

>>> new_total_value: int256 = prev_value + dv_use
    new_staked_value: int256 = v_st + dv_s

    # Solution of:
    # staked - token_reduction       new_staked_value
    # -------------------------  =  -------------------
```

```
    # supply - token_reduction         new_token_value
    token_reduction: int256 = unsafe_div(staked * new_total_value - new_staked_value * supply,
new_total_value - new_staked_value)
    # token_reduction = 0 if nothing is staked
    # XXX need to consider situation when denominator is very close to zero

    # Supply changes each time:
    # value split reduces the amount of staked tokens (but not others),
    # and this also reduces the supply of LP tokens

    return LiquidityValuesOut(
        admin=prev.admin,
>>>     total=convert(new_total_value, uint256),
        ideal_staked=prev.ideal_staked,
        staked=convert(new_staked_value, uint256),
        staked_tokens=convert(staked - token_reduction, uint256),
        supply_tokens=convert(supply - token_reduction, uint256)
    )
```

It is possible for `dv_use` to exceed `prev_value`, resulting in a negative `new_total_value`. This would also cause the conversion to `uint256` to revert.

Recommendations

Consider setting `new_total_value` to 0 when it becomes negative.

[M-02] `withdraw` does not reduce `staked` data when staker is caller

Severity

Impact: High

Likelihood: Low

Description

When `withdraw` is called, it updates `liquidity.total` based on the amount of shares burned, but it doesn't check if the caller is the `staker`. If the `staker` calls the `withdraw` operation, it should also update and decrease the `staked` value.

```
@external
@nonreentrant
def withdraw(shares: uint256, min_assets: uint256, receiver: address = msg.sender) -> uint256:
    """
    @notice Method to withdraw assets (e.g. like BTC) by spending shares (e.g. like yield-
bearing BTC)
    @param shares Shares to withdraw
    @param min_assets Minimal amount of assets to receive (important to calculate to exclude
sandwich attacks)
    @param receiver Receiver of the shares who is optional. If not specified - receiver is the
```

```
sender
    """
    assert shares > 0, "Withdrawing nothing"

    amm: LevAMM = self.amm
    liquidity_values: LiquidityValuesOut = self._calculate_values(self._price_oracle_w())
    supply: uint256 = liquidity_values.supply_tokens
    self.liquidity.admin = liquidity_values.admin
    self.liquidity.total = liquidity_values.total
    self.liquidity.staked = liquidity_values.staked
    self.totalSupply = supply
    staker: address = self.staker
    if staker != empty(address):
        self.balanceOf[staker] = liquidity_values.staked_tokens
    state: AMMState = staticcall amm.get_state()

    admin_balance: uint256 = convert(max(liquidity_values.admin, 0), uint256)

    withdrawn: Pair = extcall amm._withdraw(10**18 * liquidity_values.total //
(liquidity_values.total + admin_balance) * shares // supply)
    assert extcall COLLATERAL.transferFrom(amm.address, self, withdrawn.collateral)
    crypto_received: uint256 = extcall
COLLATERAL.remove_liquidity_fixed_out(withdrawn.collateral, 0, withdrawn.debt, 0)

    self._burn(msg.sender, shares)  # Changes self.totalSupply
>>> self.liquidity.total = liquidity_values.total * (supply - shares) // supply
    if liquidity_values.admin < 0:
        # If admin fees are negative - we are skipping them, so reduce proportionally
        self.liquidity.admin = liquidity_values.admin * convert(supply - shares, int256) //
convert(supply, int256)
    assert crypto_received >= min_assets, "Slippage"
    assert extcall STABLECOIN.transfer(amm.address, withdrawn.debt)
    assert extcall DEPOSITED_TOKEN.transfer(receiver, crypto_received)

    log Withdraw(sender=msg.sender, receiver=receiver, owner=msg.sender, assets=crypto_received,
shares=shares)
    return crypto_received
```

If the `liquidity.staked` value is not updated properly, it will use the wrong value when `_calculate_values` is called, resulting in incorrect `staked_tokens` and `supply_tokens`.

```
@internal
@view
def _calculate_values(p_o: uint256) -> LiquidityValuesOut:
    prev: LiquidityValues = self.liquidity
    staker: address = self.staker
    staked: int256 = 0
    if staker != empty(address):
        staked = convert(self.balanceOf[self.staker], int256)
    supply: int256 = convert(self.totalSupply, int256)

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

```
    cur_value: int256 = convert((staticcall self.amm.value_oracle()).value * 10**18 // p_o,
int256)
    prev_value: int256 = convert(prev.total, int256)
    value_change: int256 = cur_value - (prev_value + prev.admin)

    v_st: int256 = convert(prev.staked, int256)
    v_st_ideal: int256 = convert(prev.ideal_staked, int256)
    # ideal_staked is set when some tokens are transferred to staker address

    dv_use: int256 = value_change * (10**18 - f_a) // 10**18
    prev.admin += (value_change - dv_use)

    dv_s: int256 = dv_use * staked // supply
    if dv_use > 0:
        dv_s = min(dv_s, max(v_st_ideal - v_st, 0))

    new_total_value: int256 = prev_value + dv_use
    new_staked_value: int256 = v_st + dv_s

    # Solution of:
    # staked - token_reduction       new_staked_value
    # ------------------------  =  -------------------
    # supply - token_reduction         new_token_value
>>> token_reduction: int256 = unsafe_div(staked * new_total_value - new_staked_value * supply,
new_total_value - new_staked_value)
    # token_reduction = 0 if nothing is staked
    # XXX need to consider situation when denominator is very close to zero

    # Supply changes each time:
    # value split reduces the amount of staked tokens (but not others),
    # and this also reduces the supply of LP tokens

    return LiquidityValuesOut(
        admin=prev.admin,
        total=convert(new_total_value, uint256),
        ideal_staked=prev.ideal_staked,
        staked=convert(new_staked_value, uint256),
>>>     staked_tokens=convert(staked - token_reduction, uint256),
>>>     supply_tokens=convert(supply - token_reduction, uint256)
    )
```

Recommendations

Update the staker's `liquidity.staked` if the staker calls `withdraw`, or prevent the `staker` from calling the `withdraw` operation.

[M-03] Token rebase miscalculation during position losses

Severity

Impact: Low

Likelihood: High

Description

When the position is at a loss (dv_use < 0), the token reduction should theoretically be zero:

```
@internal
@view
def _calculate_values(p_o: uint256) -> LiquidityValuesOut:
    ...
    token_reduction: int256 = unsafe_div(staked * new_total_value - new_staked_value * supply,
new_total_value - new_staked_value)
    ...
```

We have the numerator of the calculation should evaluate to zero:

```
  staked * new_total_value - new_staked_value * supply
= staked * (prev_value + dv_use) - (v_st + dv_s) * supply
= staked * (prev_value + dv_use) - (v_st + dv_use * staked / supply) * supply
= staked * prev_value - v_st * supply
= staked * prev_value - (staked * prev_value / supply) * supply
(because v_st / staked == prev_value / supply => v_st = staked *  prev_value / supply)
= 0
```

However, due to integer division rounding, the calculation may result in a non-zero value, causing an incorrect token reduction. It can lead to unfair distribution of value between staked and unstaked liquidity providers.

Recommendations

When the position losses, set `token_reduction` = 0.

[M-04] State not updated when staker address changes

Severity

Impact: High

Likelihood: Low

Description

When changing the staker address via the `set_staker` function in the LT contract, the code fails to update important accounting variables `liquidity.staked` and `liquidity.ideal_staked`. This creates an accounting mismatch in the protocol.

```
    @external
    @nonreentrant
    def set_staker(staker: address):
        self._check_admin()
        self.staker = staker
        log SetStaker(staker=staker)
```

This creates a mismatch between:

- The staker address (`self.staker`) which points to the new staker
- The accounting variables (`liquidity.staked` and `liquidity.ideal_staked`) which still reflect values from the previous staker

When `_calculate_values` runs after the staker has been changed. `staked` is token balance from the new staker address. `v_st` uses historical accounting values from the old staker.

`token_reduction` is calculated:

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

Let's consider a scenario:

- Admin calls `set_staker` , changing self.staker from Addr_A (holding many tokens) to Addr_B (holding very few tokens).
- `liquidity.staked` (the value variable) remains high, reflecting value accrued/assigned historically to the staked portion when Addr_A was the staker.
- The next time `_calculate_values` runs:
- It uses `staked` = self.balanceOf[Addr_B] (very low token count).
- It uses `new_staked_value` , derived from the high historical `liquidity.staked` .
- The ratio `staked / supply` (low / total) will be much smaller than the ratio `new_staked_value / new_total_value` (high / total).

Since the token ratio is too low compared to the value ratio, the system needs to increase the number of tokens held by the staker (Addr_B). The system mints new tokens out of thin air and assigns them to the new staker (Addr_B) simply because the staker address was changed. This newly minted value comes from diluting all other token holders.

Recommendations

Update the `staked` and `ideal_staked` if there's a change in staker address.

[M-05] Rebase bypass possible through `_transfer()`

Severity

Impact: High

Likelihood: Low

Description

The token reduction mechanism, which is critical for maintaining accurate staked liquidity accounting, is applied only during direct deposits to the staker.

```
# File: LT.vy
...
238:     token_reduction: int256 = unsafe_div(staked * new_total_value - new_staked_value *
supply, new_total_value - new_staked_value)
...
246:     return LiquidityValuesOut(
247:         admin=prev.admin,
248:         total=convert(new_total_value, uint256),
249:         ideal_staked=prev.ideal_staked,
250:         staked=convert(new_staked_value, uint256),
251:@>      staked_tokens=convert(staked - token_reduction, uint256),
252:@>      supply_tokens=convert(supply - token_reduction, uint256)
253:     )
```

However, if a user deposits to a `non-staker` account and later transfers the shares to the staker, the token reduction is not applied because the recalculation in the `_transfer` function only adjusts the staked liquidity based on the transfer amount without invoking the token reduction logic. This can be seen in the `_transfer` function:

```
# File: LT.vy
548: @internal
549: def _transfer(_from: address, _to: address, _value: uint256):
...
564:         elif _to == staker:
565:             # Increase the staked part
566:             d_staked_value: uint256 = liquidity.total * _value // liquidity.supply_tokens
567:@>          liquidity.staked += d_staked_value
568:             if liquidity.staked_tokens > 10**10:
569:                 liquidity.ideal_staked = liquidity.ideal_staked * (liquidity.staked_tokens
+ _value) // liquidity.staked_tokens
570:             else:
571:                 # To exclude division by zero and numerical noise errors
572:                 liquidity.ideal_staked += d_staked_value
573:@>      self.liquidity.staked = liquidity.staked
574:         self.liquidity.ideal_staked = liquidity.ideal_staked
575:
576:     self.balanceOf[_from] -= _value
577:     self.balanceOf[_to] += _value
...
```

The following test shows how depositing to a non-staker and then transferring to the `staker` will make both `staked` and `staked_balance` non-zero (token reduction bypass).

```
# File: tests/lt/test_unitary.py
def test_deposit_then_transfer_to_staker(yb_lt, collateral_token, yb_allocated, seed_cryptopool,
yb_staker, accounts, admin):
    user = accounts[0]
    p = 100_000
    amount = 10**18
```

```python
    with boa.env.prank(admin):  # Set the staker
        yb_lt.set_staker(yb_staker.address)
        assert yb_lt.staker() == yb_staker.address

    # First deposit just to populate the pool and set the staker
    collateral_token._mint_for_testing(accounts[1], amount)
    with boa.env.prank(accounts[1]):
        shares = yb_lt.deposit(amount, p * amount, int(amount * 0.9999))

    # 1. Deposit but staking this time using the deposit -> transfer method
    collateral_token._mint_for_testing(user, amount)
    with boa.env.prank(user):
        shares = yb_lt.deposit(amount, p * amount, int(amount * 0.9999))
    with boa.env.prank(user): # Transfer to staker
        yb_lt.transfer(yb_staker, shares)

    # 2. Rebase mechanism has applied but the `staked` is not zero
    post_values = yb_lt.internal._calculate_values(100_000 * 10**18)
    assert post_values[3] > 0 # staked > 0
    assert post_values[4] > 0 # staked_tokens > 0
```

There is a discrepancy because if user desposits directly to the `staker` account, the token reduction would have been applied, resulting in zero staked amounts.

```python
# File: tests/lt/test_unitary.py
def test_deposit_directly_to_staker(yb_lt, collateral_token, yb_allocated, seed_cryptopool,
yb_staker, accounts, admin):
    user = accounts[0]
    p = 100_000
    amount = 10**18

    # First deposit just to populate the pool and set the staker
    with boa.env.prank(admin):  # Set the staker
        yb_lt.set_staker(yb_staker.address)
        assert yb_lt.staker() == yb_staker.address
    collateral_token._mint_for_testing(accounts[1], amount)
    with boa.env.prank(accounts[1]):
        shares = yb_lt.deposit(amount, p * amount, int(amount * 0.9999))

    # 1. Deposit but staking directly to staker
    collateral_token._mint_for_testing(user, amount)
    with boa.env.prank(user):
        # Deposit
        shares = yb_lt.deposit(amount, p * amount, int(amount * 0.9999), yb_lt.staker())
        assert shares == yb_lt.balanceOf(yb_lt.staker())

    # 2. Rebase mechanism has applied so the `staked` is zero
    post_values = yb_lt.internal._calculate_values(100_000 * 10**18)
    assert post_values[3] == 0 # staked == 0
    assert post_values[4] == 0 # staked_tokens == 0
```

Recommendations

Ensure that the token reduction mechanism is consistently applied regardless of whether the deposit occurs directly to the staker or via a subsequent transfer.

[M-06] `set_rate()` resets accrued fees causing fee loss

Severity

Impact: High

Likelihood: Low

Description

The `AMM::set_rate` function is used to change the interest rate applied to borrowed debt over time. Internally, it resets the rate multiplier (`rate_mul`) based on the current time and accrued interest up to that point.

```
# File: AMM.vy
def set_rate(rate: uint256) -> uint256:
...
    assert msg.sender == DEPOSITOR, "Access"
    rate_mul: uint256 = self._rate_mul()
@>  self.rate_mul = rate_mul
@>  self.rate_time = block.timestamp
    self.rate = rate
    log SetRate(rate=rate, rate_mul=rate_mul, time=block.timestamp)
    return rate_mul
```

However, **if** `AMM::collect_fees()` **is not called beforehand**, any accrued interest from the old rate is **not collected**, meaning that fees owed to the protocol for the previous period are effectively erased.

This occurs because `AMM::collect_fees()` relies on `AMM::_debt_w()`, which uses the formula:

```
# File: AMM.vy
    debt: uint256 = self.debt * rate_mul // self.rate_mul
```

In this formula:

- `self.rate_mul` is updated during `set_rate()`.
- If `set_rate()` is called before fees are collected, the base `rate_mul` is reset to the *new* value, and the accrued delta is lost.
- Subsequent calls to `collect_fees()` will compute fees relative only to the time after the new rate was set, **not accounting for the previous interest period**.

The provided test confirms this behavior:

- After interest accrues, `admin_fees()` correctly reflects fees > 0 (step 2).
- After `set_rate` is called, `admin_fees()` drops to 0 (step 4), proving that the fees were silently wiped due to the rate reset.

```
def test_fees_loss_on_set_rate(token_mock, price_oracle, amm_deployer, accounts, admin):
    # Deploy tokens and AMM (using 18 decimals for simplicity)
    stablecoin = token_mock.deploy('Stablecoin', 'USD', 18)
    collateral_decimals = 18
    collateral_token = token_mock.deploy('Collateral', 'COL', collateral_decimals)
    with boa.env.prank(admin):
        price_oracle.set_price(10**18)
        amm = amm_deployer.deploy(
            admin,
            stablecoin.address,
            collateral_token.address,
            2 * 10**18,    # leverage = 2x
            10**16,        # fee
            price_oracle.address
        )
        amm.set_rate(10**18)  # Set initial rate
    # Fund AMM with tokens
    with boa.env.prank(admin):
        stablecoin._mint_for_testing(amm.address, 10**12 * 10**18)
        stablecoin._mint_for_testing(admin, 10**12 * 10**18)
        collateral_token._mint_for_testing(admin, 10**12 * 10**18)
        stablecoin.approve(amm.address, 2**256 - 1)
        collateral_token.approve(amm.address, 2**256 - 1)

    # 1. Make a deposit to generate some minted debt (and thus potential fees)
    d_collateral = 10**18
    d_debt = 10**17
    with boa.env.prank(admin):
        amm._deposit(d_collateral, d_debt)

    # 2. Simulate passage of time to increase the accrued interest
    boa.env.time_travel(60 * 60 * 24)
    fees_before_set_rate = amm.admin_fees()
    assert fees_before_set_rate > 0

    # 3. Call set_rate without prior fee collection, which resets the rate multiplier
    new_rate = 11**17  # arbitrary new rate
    with boa.env.prank(admin):
        amm.set_rate(new_rate)

    new_fees = amm.admin_fees()

    # 4. The test asserts that the new computed fees are lower than before,
    # proving that fees accrued (if any) are lost when set_rate is called without collecting
fees.
    assert new_fees == 0
    assert fees_before_set_rate > new_fees
```

Recommendations

Enforce fee collection before rate updates. Add logic in `set_rate()` to require `collect_fees()` to be called first. For example:

```
# File: AMM.vy
def set_rate(rate: uint256) -> uint256:
    self.collect_fees()
    ...
```

[M-07] Token miscalculation in `withdraw_admin_fees()` inflates shares

Severity

Impact: Medium

Likelihood: Medium

Description

In the LT contract, the `withdraw_admin_fees` function is designed to mint Yield Basis tokens to the fee receiver. However, there's a calculation error that results in excessive token minting. The issue lies in the `to_mint` calculation:

```
def withdraw_admin_fees():
    ...
    to_mint: uint256 = v.supply_tokens * new_total // v.total
    ...
```

This formula mints an amount equal to the entire new supply, not just the incremental difference representing admin fees. It leads to inflation of token supply and dilution of existing holders.

Recommendations

To mint only the incremental number of tokens representing the admin fees:

```
- to_mint: uint256 = v.supply_tokens * new_total // v.total
+ to_mint: uint256 = v.supply_tokens * new_total // v.total - v.supply_tokens
```

[M-08] Incorrect total supply calculated during admin fee withdrawal

Severity

Impact: Medium

Likelihood: Medium

Description

In the LT contract, when the `withdraw_admin_fees` function is called, the `_calculate_values` function is executed to recalculate all liquidity values. During this calculation, the total supply can be reduced due to a token reduction mechanism (downward rebasing) that adjusts the number of tokens based on value changes.

However, while the function updates various liquidity parameters, it fails to update `self.totalSupply` to the latest value `v.supply_tokens` before minting new tokens to the fee receiver. This mistake leads to an incorrect total supply calculation after admin fees are withdrawn.

```
def withdraw_admin_fees():
    ...
    v: LiquidityValuesOut = self._calculate_values(self._price_oracle_w())
    ...
    self.liquidity.total = new_total
    self.liquidity.admin = 0
    self.liquidity.staked = v.staked
    staker: address = self.staker
    if staker != empty(address):
        self.balanceOf[staker] = v.staked_tokens

    log WithdrawAdminFees(receiver=fee_receiver, amount=to_mint)
```

Recommendations

Update `self.totalSupply` to the recalculated value before minting tokens to the fee receiver.

[M-09] `set_allocator()` state update missing causes incorrect balances

Severity

Impact: Medium

Likelihood: Medium

Description

In the `set_allocator` function, the contract intends to adjust the allocation for a given allocator by comparing the new `amount` with the `old_allocation` stored in `self.allocators[allocator]`. However, the function does not update the `self.allocators` mapping with the new `amount`. As a consequence, every call to `set_allocator` will always see the previous allocation (old_allocation) as zero, and any subsequent logic that relies on `self.allocators` (for example, when an allocator attempts to withdraw their assets) will be based on zero values.

```
# File: Factory.vy
212: @external
213: @nonreentrant
214: def set_allocator(allocator: address, amount: uint256):
215:     assert msg.sender == self.admin, "Access"
216:     assert allocator != self.mint_factory, "Minter"
217:     assert allocator != empty(address)
218:
219:     old_allocation: uint256 = self.allocators[allocator]
220:     if amount > old_allocation:
221:         # Use transferFrom
222:         extcall STABLECOIN.transferFrom(allocator, self, amount - old_allocation)
223:     elif amount < old_allocation:
224:         # Allow to take back the allocation via transferFrom, but not more than the
allocation reduction
225:         extcall STABLECOIN.approve(allocator, (staticcall STABLECOIN.allowance(self,
allocator)) + old_allocation - amount)
226:
227:     log SetAllocator(allocator=allocator, amount=amount)
```

In this function, after comparing the new allocation amount with `old_allocation` (line 220) and performing the appropriate token transfers or approvals (lines 220-225), the contract never updates the state variable. Thus, `self.allocators[allocator]` remains unchanged (likely zero), so any future operations (such as withdrawing assets) will calculate allocation based on an incorrect or zero value.

The following test demostrates how the `self.allocators` is not updated:

```
# File: tests/lt/test_factory.py
def test_allocator_not_registered(factory, admin, accounts, stablecoin):
    # Mint tokens and set allocator using admin privileges
    with boa.env.prank(accounts[0]):
        stablecoin.approve(factory.address, 2**256-1)
    with boa.env.prank(admin):
        stablecoin._mint_for_testing(accounts[0], 10**18)
        factory.set_allocator(accounts[0], 10**18)
    # Verify that the allocator deposit amount is not registered in self.allocators (remains
zero)
    deposit = factory.allocators(accounts[0])
    assert deposit == 0
```

Recommendations

Modify the function to update `self.allocators[allocator]` with the new allocation value.

```
def set_allocator(allocator: address, amount: uint256):
    ...
+   self.allocators[allocator] = amount
    ...
```

Low findings

[L-01] Staker contract missing in `LT` contract post-creation

In the Factory contract, the `add_market` function creates a staker contract for the market if `staker_impl` is provided. However, the code doesn't set this newly created staker in the corresponding LT contract.

```
def add_market(
    pool: CurveCryptoPool,
    fee: uint256,
    rate: uint256,
    debt_ceiling: uint256
) -> Market:
    ...
    if self.staker_impl != empty(address):
        market.staker = create_from_blueprint(
            self.staker_impl,
            market.lt)

    ...
```

Therefore, the staker exists but isn't properly configured in the LT contract. As a result, the staking functionality won't work properly until the staker is manually set in a separate transaction.

It's recommended to set the staker for the LT in add_market function.

```
    if self.staker_impl != empty(address):
        market.staker = create_from_blueprint(
            self.staker_impl,
            market.lt)
+       extcall LT(market.lt).set_staker(market.staker)
```

[L-02] `min_admin_fee` lacks initialization and update

In `LT.vy`, the variable `min_admin_fee` is declared as a public and is used in the fee calculation within the `_calculate_values` function:

```
# File: LT.vy
136: min_admin_fee: public(uint256)
...
204: def _calculate_values(p_o: uint256) -> LiquidityValuesOut:
...
212:     f_a: int256 = convert(
213:         10**18 - (10**18 - self.min_admin_fee) * self.sqrt(convert(10**36 - staked *
10**36 // supply, uint256)) // 10**18,
214:         int256)
...
```

The calculation for `f_a` uses `self.min_admin_fee` to determine the minimum fee that should be applied. However, there is no function in the contract that allows an administrator to set or update `min_admin_fee`, nor is it initialized to a nonzero value upon contract deployment. As a consequence, `self.min_admin_fee` remains 0.

The following test demostrates how `min_admin_fee` is zero at the contract deploy, also there is no setter to adjust it.

```
File: tests/lt/test_factory.py

def test_min_admin_fee_default(factory, cryptopool, seed_cryptopool, lt_interface, admin):
    fee = int(0.007e18)
    rate = int(0.1e18 / (365 * 86400))
    ceiling = 100 * 10**6 * 10**18

    with boa.env.prank(admin):
        market = factory.add_market(cryptopool.address, fee, rate, ceiling)
    # Assert that LT.min_admin_fee remains 0 (no setter to adjust it)
    lt = market[3]
    assert lt_interface.at(lt).min_admin_fee() == 0
```

Update the constructor set `self.min_admin_fee` to a nonzero value that reflects the intended minimum admin fee. Also, implement a function (with proper access control) to update `min_admin_fee`.

[L-03] `deposit` fails to account for cases where `value_before` equals 0

When `deposit` is called with a non-zero total supply, it calculates shares using the formula: `supply * value_after // value_before - supply`.

```
@external
@nonreentrant
def deposit(assets: uint256, debt: uint256, min_shares: uint256, receiver: address = msg.sender)
-> uint256:
    """
    @notice Method to deposit assets (e.g. like BTC) to receive shares (e.g. like yield-bearing
BTC)
    @param assets Amount of assets to deposit
    @param debt Amount of debt for AMM to take (approximately BTC * btc_price)
    @param min_shares Minimal amount of shares to receive (important to calculate to exclude
sandwich attacks)
    @param receiver Receiver of the shares who is optional. If not specified - receiver is the
sender
    """
    amm: LevAMM = self.amm
    assert extcall STABLECOIN.transferFrom(amm.address, self, debt)
    assert extcall DEPOSITED_TOKEN.transferFrom(msg.sender, self, assets)
    lp_tokens: uint256 = extcall COLLATERAL.add_liquidity([debt, assets], 0, amm.address)
    p_o: uint256 = self._price_oracle_w()

    supply: uint256 = self.totalSupply
    shares: uint256 = 0
```

```
    liquidity_values: LiquidityValuesOut = empty(LiquidityValuesOut)
    if supply > 0:
        liquidity_values = self._calculate_values(p_o)

    v: ValueChange = extcall amm._deposit(lp_tokens, debt)
    value_after: uint256 = v.value_after * 10**18 // p_o

    # Value is measured in USD
    # Do not allow value to become larger than HALF of the available stablecoins after the
deposit
    # If value becomes too large - we don't allow to deposit more to have a buffer when the
price rises
    assert staticcall amm.max_debt() // 2 >= v.value_after, "Debt too high"

    staker: address = self.staker

    if supply > 0:
        supply = liquidity_values.supply_tokens
        self.liquidity.admin = liquidity_values.admin
        value_before: uint256 = liquidity_values.total
        value_after = convert(convert(value_after, int256) - liquidity_values.admin, uint256)
        self.liquidity.total = value_after
        self.liquidity.staked = liquidity_values.staked
        self.totalSupply = liquidity_values.supply_tokens  # will be increased by mint
        if staker != empty(address):
            self.balanceOf[staker] = liquidity_values.staked_tokens
        # ideal_staked is only changed when we transfer coins to staker
>>>     shares = supply * value_after // value_before - supply

    else:
        # Initial value/shares ratio is EXACTLY 1.0 in collateral units
        # Value is measured in USD
        shares = value_after
        # self.liquidity.admin is 0 at start but can be rolled over if everything was withdrawn
        self.liquidity.ideal_staked = 0  # Likely already 0 since supply was 0
        self.liquidity.staked = 0        # Same: nothing staked when supply is 0
        self.liquidity.total = shares    # 1 share = 1 crypto at first deposit
        self.liquidity.admin = 0         # if we had admin fees - give them to the first
depositor; simpler to handle
        self.balanceOf[staker] = 0

    assert shares >= min_shares, "Slippage"

    self._mint(receiver, shares)
    log Deposit(sender=msg.sender, owner=receiver, assets=assets, shares=shares)
    return shares
```

This means if `value_before` drops to zero, the `deposit` operation will revert.

Recommendations

Add an additional condition, if supply is non-zero but `value_before` becomes 0, set shares to `value_after`

[L-04] `fill_staker_vpool()` fails without address setters

The `fill_staker_vpool` function in the Factory contract is designed to add missing virtual pool and staker components to existing markets. However, this function cannot work effectively because it depends on implementation addresses that cannot be updated after contract deployment.

The function checks for two conditions:

- If `market.virtual_pool == empty(address) AND self.virtual_pool_impl != empty(address) AND self.flash != empty(address)`
- If `market.staker == empty(address) AND self.staker_impl != empty(address)`

If these implementation addresses are initially set to `empty(address)` during deployment, the `fill_staker_vpool` function will never be able to create virtual pools or stakers for existing markets, as there's no way to set these implementation addresses later.

```
def fill_staker_vpool(i: uint256):
    assert msg.sender == self.admin, "Access"
    market: Market = self.markets[i]
    if market.virtual_pool == empty(address) and self.virtual_pool_impl != empty(address) and
self.flash != empty(address):
        market.virtual_pool = create_from_blueprint(
            self.virtual_pool_impl,
            market.amm,
            self.flash
        )
    if market.staker == empty(address) and self.staker_impl != empty(address):
        market.staker = create_from_blueprint(
            self.staker_impl,
            market.lt)
    self.markets[i] = market
```

Add setter functions to update the implementation addresses after contract deployment:

```
@external
def set_virtual_pool_impl(impl: address):
    assert msg.sender == self.admin, "Access"
    self.virtual_pool_impl = impl

@external
def set_staker_impl(impl: address):
    assert msg.sender == self.admin, "Access"
    self.staker_impl = impl
```