yAudit

November 24, 2025

Prepared for
Yield Basis

Audited by
fedebianu
adriro

Yield Basis DAO Security Review

Smart Contract Security Assessment

Contents

1 Review Summary

1.1 Protocol Overview

Yield Basis is a protocol that features a new type of AMM that focuses on solving impermanent loss. The current review targets the DAO contracts of the protocol, including the governance token, along with the mechanism to vote, incentivize pools, and distribute rewards.

1.2 Audit Scope

This audit covers 6 smart contracts across 5 days of review.

```
contracts/dao
├── CliffEscrow.vy
├── GaugeController.vy
├── LiquidityGauge.vy
├── VestingEscrow.vy
├── VotingEscrow.vy
└── YB.vy
```

1.3 Risk Assessment Framework

1.3.1 Severity Classification

1.4 Key Findings

Breakdown of Finding Impacts

Impact Level	Count
Critical	4
High	4
Medium	1
Low	2
Informational	5

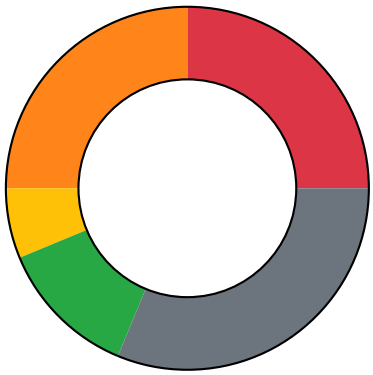

Figure 1: Distribution of security findings by impact level

1.5 Overall Assessment

Given the complexity of the contracts and the high number of severe findings present in this report, the auditors recommend strengthening the testing suite and conducting a new security review.

Severity	Description	Potential Impact
Critical	Immediate threat to user funds or protocol integrity	Direct loss of funds, protocol compromise
High	Significant security risk requiring urgent attention	Potential fund loss, major functionality disruption
Medium	Important issue that should be addressed	Limited fund risk, functionality concerns
Low	Minor issue with minimal impact	Best practice violations, minor inefficiencies
Undetermined	Findings whose impact could not be fully assessed within the time constraints of the engagement. These issues may range from low to critical severity, and although their exact consequences remain uncertain, they present a sufficient potential risk to warrant attention and remediation.	Varies based on actual severity
Gas	Findings that can improve the gas efficiency of the contracts.	Reduced transaction costs
Informational	Code quality and best practice recommendations	Improved maintainability and readability

Table 1: severity classification

2 Audit Overview

2.1 Project Information

Protocol Name: Yield Basis
Repository: https://github.com/yield-basis/yb-core
Commit Hash: f6104fc017a6022f0ad464bd8d5147850d1166f2
Commit URL:
https://github.com/yield-basis/yb-core/commit/f6104fc017a6022f0ad464bd8d5147850d1166f2

2.2 Audit Team

fedebianu, adriro

2.3 Audit Resources

Code repositories and documentation

Category	Mark	Description
Access Control	Good	Correct usage of access control protection.
Mathematics	Low	Several issues related to calculations or scaling were detected.
Complexity	Average	The contracts are well-designed, but contain complex logic, particularly within the GaugeController and LiquidityGauge contracts.
Libraries	Good	The codebase relies on the snekmate library.
Decentralization	Good	The protocol and its emissions are governed by a set of decentralized contracts.
Code Stability	Good	The codebase remained stable during the review.
Documentation	Average	The contracts are decorated with NatSpec metadata. Additional high-level documentation focused on explaining the dynamics of voting and gauges is encouraged.
Monitoring	Good	Monitoring events are in place.
Testing and verification	Low	Despite the presence of fuzzing tests in the codebase, multiple severe issues remained undetected.

Table 2: Code Evaluation Matrix

2.4 Critical Findings

2.4.1 NFT reentrancy allows using voting power when transferring the token

A reentrancy in `safeTransferFrom()` would allow an attacker to use their voting power just before being merged with the recipient.

Technical Details

The implementation of `safeTransferFrom()` does the transfer with the callback after performing the check but before clearing the owner's voting power.

```
538: def safeTransferFrom(owner: address, to: address, token_id: uint256, data: Bytes[1_024] = b""):
539:     assert erc721._is_approved_or_owner(msg.sender, token_id), "erc721: caller is not token owner or approved"
540:     assert self._ve_transfer_allowed(owner, to), "Need max veLock"
541:     erc721._safe_transfer(owner, to, token_id, data)
542:     self._merge_positions(owner, to)
543:     erc721._burn(token_id)
```

This would allow a malicious user to exercise their voting power in between the transfer, as the check is done before the callback, and their votes are reset after the callback.

Impact

Critical. Voting power can be reused multiple times by performing the described attack.

Recommendation

Since the token is immediately burned after transferring, the underlying transfer operation is not needed; the token can be just burned.

Developer Response

Fixed for both `transferFrom()` and `safeTransferFrom()` in 4d6cd7183e39487a80dc51b7976099b156f440a0.

2.4.2 Incorrect scale in `LiquidityGauge` checkpoint

The user reward calculation is incorrectly normalized, leading to an overinflated amount.

Technical Details

In `_checkpoint()`, `integral_inv_supply` is scaled by `1e36` but only normalized by `1e18` when calculating `d_user_reward`.

```
139: def _checkpoint(reward: IERC20, d_reward: uint256, user: address) ->
RewardIntegrals:
140:     r: RewardIntegrals = empty(RewardIntegrals)
141:
142:     r.integral_inv_supply = self.integral_inv_supply
143:     if block.timestamp > r.integral_inv_supply.t:
144:         r.integral_inv_supply.v += unsafe_div(10**36 * (block.timestamp - r.
integral_inv_supply.t), erc4626.erc20.totalSupply)
145:         r.integral_inv_supply.t = block.timestamp
146:
147:     if reward.address != empty(address):
148:         r.reward_rate_integral = self.reward_rate_integral[reward]
149:         if block.timestamp > r.reward_rate_integral.t:
150:             r.reward_rate_integral.v += (r.integral_inv_supply.v - self.
integral_inv_supply_4_token[reward]) * d_reward //\
151:                (block.timestamp - r.reward_rate_integral.t)
152:             r.reward_rate_integral.t = block.timestamp
153:
154:     if user != empty(address):
155:         r.user_rewards_integral = self.user_rewards_integral[user][reward]
156:         if block.timestamp > r.user_rewards_integral.t:
157:             r.d_user_reward = (r.reward_rate_integral.v - self.
reward_rate_integral_4_user[user][reward]) *\
158:                 erc4626.erc20.balanceOf[user] // 10**18
159:             r.user_rewards_integral.v += r.d_user_reward
160:             r.user_rewards_integral.t = block.timestamp
161:
162:     return r
```

Impact

Critical. Reward amounts are inflated by a factor of `1e18`.

Recommendation

In `d_user_reward`, divide by `10**36` instead of `10**18`.

Developer Response

Fixed in dbc12194da1971ba7a32c459374b9de05a428787.

2.4.3 User checkpoint in `LiquidityGauge` fails to store rewards

Rewards assigned to users are lost when the user state is checkpointed.

Technical Details

Unlike `claim()`, the implementation of `_checkpoint_user()` does not transfer the earned rewards to the user. However, these rewards are not stored in the contract state, so they cannot be claimed later.

Impact

Critical. Earned rewards are lost when the user interacts with the vault.

Recommendation

The `_checkpoint_user()` function should store earned rewards (`d_user_reward`) in an accumulator of pending rewards, which can be flushed in `claim()`. An alternative would be to use `user_rewards_integral.v`, which is the historic accumulated rewards, but this would also require an accumulator to track the withdrawn tokens to calculate the difference.

Developer Response

Added such an accumulator in 1b3b6f49c5c5140ffb2f2f5b97851e578e36f12b.

2.4.4 Incorrect interpretation of released rewards in `LiquidityGauge`

The LiquidityGauge contract uses the value of the checkpointed reward rate integral to determine the updated amount of released reward tokens, leading to multiple accounting problems.

Technical Details

The implementation of `_get_vested_rewards()` takes the value of `reward_rate_integral[token].v` as the amount of released rewards up to the latest checkpointed time (`used_rewards`).

```
170:     last_reward_time: uint256 = self.reward_rate_integral[token].t
171:     used_rewards: uint256 = self.reward_rate_integral[token].v
172:     finish_time: uint256 = self.rewards[token].finish_time
173:     total: uint256 = self.rewards[token].total
174:     if finish_time > last_reward_time:
175:         new_used: uint256 = (total - used_rewards) * (block.timestamp -
last_reward_time) //\
176:             (finish_time - last_reward_time) + used_rewards
177:         return min(new_used, total) - used_rewards
```

The `used_rewards` variable is then used to compute the remaining amount (`total - used_rewards`) and to calculate the new value of distributed rewards (`new_used`). However, taking the integral of the reward rate is incorrect, as the inverse of the total supply already scales this value as part of the checkpoint process.

```
148:         r.reward_rate_integral = self.reward_rate_integral[reward]
149:         if block.timestamp > r.reward_rate_integral.t:
150:             r.reward_rate_integral.v += (r.integral_inv_supply.v - self.
integral_inv_supply_4_token[reward]) * d_reward //\
151:                 (block.timestamp - r.reward_rate_integral.t)
152:             r.reward_rate_integral.t = block.timestamp
```

Impact

Critical. The issue leads to multiple accounting problems and side effects, potentially causing the vault to malfunction.

Recommendation

Have a dedicated accumulator to measure the amount of distributed rewards per token. This counter should be incremented by the return value of `_vest_rewards()` whenever the state is persisted to storage (`_checkpoint_user()`, `claim()`, `deposit_reward()`).

Developer Response

Fixed in 657e8c679cb575f35681fab810ff6615ed113924.

2.5 High Findings

2.5.1 Incorrect emissions rate in YB token

The emission rate in the Yield Basis token is incorrectly scaled.

Technical Details

The emissions calculation is given by the implementation of `_emissions()`:

```
50: def _emissions(t: uint256, rate_factor: uint256) -> uint256:
51:     assert rate_factor <= 10**18
52:     dt: int256 = convert(t - self.last_minted, int256)
53:     rate: int256 = convert(max_mint_rate * rate_factor // 10**18, int256)
54:     reserve: int256 = convert(self.reserve, int256)
55:     return convert(
56:         reserve * (10**18 - math._wad_exp(-dt * rate // 10**18)) // 10**18,
57:         uint256)
```

The `rate` variable is calculated as the `max_mint_rate` (max rate per second) scaled by the `rate_factor`. However, the `rate` variable is then normalized again by `10**18` in the expression of `-dt * rate // 10**18` (line 56). Given `dt` and `rate` are denominated in seconds, dividing by `10**18` will yield the incorrect result.

Impact

High. The emission rate in YB token is incorrect and will likely result in zero emissions.

Recommendation

The exponential term should multiply `dt` (elapsed seconds) by `rate` (emissions per second).

```
reserve * (10**18 - math._wad_exp(-dt * rate)) // 10**18
```

Developer Response

Fixed in 14ff742f630a64f786beb5589ee978f541950768.

2.5.2 `VotingEscrow.increase_amount()` has a parameter inconsistency that could lead to unauthorized lock modifications

`VotingEscrow.increase_amount()` has a critical parameter inconsistency that creates a mismatch between validation and execution.

Technical Details

`VotingEscrow.increase_amount()` performs validation checks against `msg.sender`'s lock data but then calls `_deposit_for(_for, ...)`, which modifies the lock for the `_for` address. This creates a scenario where the actual token deposit and lock modification occur for the `_for` address. At the same time, the validation is done with `msg.sender`'s lock data, and the `_locked` parameter passed to `_deposit_for` contains `msg.sender`'s lock data, not `_for`'s.

Impact

High. This vulnerability allows:

- Users to call the function for addresses that don't have existing locks
- Bypassing validation that the target lock is valid and non-expired
- Creating inconsistent lock states where the wrong lock parameters are used
- Potential manipulation of the voting escrow system by depositing for invalid targets

Recommendation

Fix the parameter inconsistency by ensuring validation and operation target the same address:

```
    @external
    @nonreentrant
    def increase_amount(_value: uint256, _for: address = msg.sender):
-       _locked: LockedBalance = self.locked[msg.sender]
+       _locked: LockedBalance = self.locked[_for]

        assert _value > 0  # dev: need non-zero value
        assert _locked.amount > 0, "No existing lock found"
        assert _locked.end > block.timestamp, "Cannot add to expired lock.
Withdraw"

        self._deposit_for(_for, _value, 0, _locked, LockActions.INCREASE_AMOUNT)
```

Alternatively, you can remove the `_for` parameter if it's not needed.

Developer Response

Fixed in 80137af5ce7e969f2a10d254b0a05756f3c2c6ce.

2.5.3 Token emission calculations use stale weights for non-checkpointed gauges

When checkpointing a specific gauge, the global emission rate calculation uses potentially stale weights from other gauges that haven't been recently checkpointed.

Technical Details

`GaugeController._checkpoint_gauge()` updates weights only for the target gauge, but then uses global weight sums (`aw_sum`, `w_sum`) to calculate the emission rate factor.
If other gauges have changed their weights but haven't been checkpointed, their stale weights are still included in the global sums, leading to incorrect emission rate calculations.

Impact

High. Incorrect emission rates affect the entire protocol's token distribution, potentially over-minting or under-minting tokens based on outdated weight information.

Recommendation

Implement global gauge aggregation and update it during gauge checkpoints, similar to Curve's `_get_total()` approach, adapted for Yield Basis with no gauge types.

Developer Response

Disagree. That's the thing. Weights are only ever updated with per-gauge checkpoint.
That checkpoint happens at voting as well as at claim. But if gauge, for example, had adjustment number changed somehow - a stale number is applied all across the board before checkpoint or claim happens.
It is by design. So weights CANNOT actually update without being checkpointed.
Disadvantage of this approach is that weights cannot be a pure function of time: they are only updated in actions which cause checkpoints. But this is not a big disadvantage because timescale of vote weight changes (years) is much larger than times between checkpoints (e.g. between claims, or deposits/withdrawals).

2.5.4 Incorrect finish time calculation changes reward distribution rate instead of maintaining it

`LiquidityGauge.deposit_reward()` incorrectly calculates the new finish time when adding rewards to an ongoing distribution period, resulting in unintended changes to the reward distribution rate.

Technical Details

`LiquidityGauge.deposit_reward()` contains logic to extend the finish time when adding rewards to maintain the current rate:

```
# Keep the reward rate
assert r.finish_time > block.timestamp, "Rate unknown"
r.finish_time = block.timestamp + (r.finish_time - block.timestamp) * (r.total + amount)
 // r.total
```

However, this formula uses the cumulative total deposited tokens instead of the remaining undistributed tokens, causing the distribution rate to change:
Example scenario:

- Initial: 1000 tokens over 10 days → rate = 100 tokens/day
- `r.total = 1000` (cumulative total)
- After 6 days: 600 tokens distributed, 400 remaining
- Add 500 tokens using the current formula:
- `remaining_time = 4 days`
- `r.total + amount = 1000 + 500 = 1500`
- `ratio = 1500 / 1000 = 1.5`
- `new_remaining_time = 4 * 1.5 = 6 days`
- New rate = `900 / 6 = 150 tokens/day`

The formula should use remaining tokens (400) instead of the cumulative total (1000) to maintain the same rate:

- Correct ratio = `(400 + 500) / 400 = 2.25`
- Correct new time = `4 * 2.25 = 9 days`
- Correct rate = `900 / 9 = 100 tokens/day`

Impact

High. The reward distribution rate changes unexpectedly when adding tokens to ongoing distributions, violating the stated intention to ”keep the reward rate”.

Recommendation

Calculate the extension based on the original rate to maintain consistent distribution: `block.timestamp + (undistributed_reward + amount) / reward_rate`.

Developer Response

Fixed in b5a9150825ae440e77a512f90ed3a44d126c63ad.

2.6 Medium Findings

2.6.1 `LiquidityGauge.withdraw()` is broken due to incorrect `assert` logic

`LiquidityGauge.withdraw()` contains an incorrect `assert` statement that validates the wrong account's withdrawal capacity, causing legitimate withdrawal operations to fail when transferring assets to third parties.

Technical Details

In `LiquidityGauge.withdraw()` the `assert` checks `_max_withdraw(receiver)` instead of `_max_withdraw(owner)`. The validation should ensure the `owner` has sufficient withdrawable balance, not the `receiver`.

Impact

Medium. This bug breaks core functionality, but the severity is mitigated because `redeem()` is correct and can be used instead.

Recommendation

Fix the assert to validate the `owner`'s withdrawal capacity instead of the `receiver`'s.

Developer Response

Fixed in 58ca825891930e069ad1a1f2c82581399a5004fa.

2.7 Low Findings

2.7.1 Wrong condition in `preview_emissions()`

The early return in `preview_emissions()` implements the wrong condition.

Technical Details

The implementation of `preview_emissions()` returns early with zero, using the wrong condition. A gauge isn't registered when `time_weight[gauge] == 0`, a positive value means the gauge has been enabled.

```
325:     if self.time_weight[gauge] > 0:
326:         return 0
```

Impact

Low. `preview_emissions()` always returns zero for registered gauges.

Recommendation

Change the condition to `if self.time_weight[gauge] == 0`.

Developer Response

Fixed in 7192d1b8f7f7d303c3fd6fe39ae2358591ff995b.

2.7.2 Prevent LP token from being registered as rewards

Using the LP token as a reward would conflict with the vault's accounting.

Technical Details

LP tokens sent by the distributor would be mixed with staked tokens and treated as the vault's assets, disrupting the accounting.

Impact

Low.

Recommendation

Check that the reward token is not the LP token in `add_reward()`.

```
    def add_reward(token: IERC20, distributor: address):
        assert token != YB, "YB"
+       assert token != LP_TOKEN, "LP_TOKEN"
```

Developer Response

Fixed at 395d1db03e4a9aa892642022c794a7ef9c066105.

2.8 Gas Savings Findings

2.8.1 Remove unused variables in `VotingEscrow._merge_positions()`

Technical Details

In `VotingEscrow._merge_positions()`, two variables `pt` and `to_pt` are declared and assigned but never used, resulting in unnecessary gas consumption.

Impact

Gas savings.

Recommendation

Remove the unused variables.

Developer Response

Fixed in cdf753d798f7bb8ca2f7c894ee50a54408878072.

2.9 Informational Findings

2.9.1 Follow CEI pattern

Some functions update the contract state after making an external call, violating the Checks-Effects-Interactions (CEI) pattern.

Technical Details

`VotingEscrow.withdraw()` performs the state update `erc721._burn(convert(msg.sender, uint256))` before making the external call to `TOKEN.transfer()`.
`LiquidityGauge.deposit_reward()` performs the state update `self.rewards[token] = r` before making the external call to `token.transferFrom()`.

Impact

Informational.

Recommendation

Follow the CEI pattern as a best practice.

Developer Response

Fixed in 7c28c5b3cff00a71b5c598507b99ea4a8e16e389.

2.9.2 Fix typos

Technical Details

The `CliffEscrow.vy` contract contains a spelling error in an immutable variable name:

```
RECEPIENT: public(immutable(address))
```

Impact

Informational.

Recommendation

Fix typos.

Developer Response

Fixed in d3dd301fe26e47cf575b844c4bd2a2c6efbc50b2.

2.9.3 `VotingEscrow.getPastVotes()` should not return future voting power

Technical Details

`VotingEscrow.getPastVotes()` allows querying voting power at any timestamp without proper validation. However, the function's purpose is to return voting power from the past only.

Impact

Informational.

Recommendation

Add proper `timepoint` validation check:

```
assert timepoint <= block.timestamp, "Timepoint in the future"
```

Developer Response

Acknowledged. It's a view method which is to be called from frontend. Sometimes RPCs are using multiple nodes behind a load balancer which could be not entirely at sync (one can be ahead of another by 1 block). So, I can imagine a situation that check <= block.timestamp will actually fail after reading the current timestamp worked. Moreover, this function will also work in the near future. Overall, I think better leave it without this assert.

2.9.4 Check for array length mismatch

Technical Details

`VestingEscrow.fund()` accepts two arrays but only validates the loop bounds against the recipients array.

Impact

Informational.

Recommendation

Add explicit length validation for better error messaging:

```
assert len(_recipients) == len(_amounts), "Array length mismatch"
```

Developer Response

Fixed in 926621b0e79d8f2c08d7f69f2d24e9b6f02447a7.

2.9.5 Missing exports from modules

There are several functions inherited from modules that are not re-exported from the contract.

Technical Details

- GaugeController.vy:
- `owner()`
- VestingEscrow.vy:
- `owner()`
- VotingEscrow.vy:
- `tokenURI()`
- `supportsInterface()`

Impact

Informational.

Recommendation

Add the missing exports to expose the functions.

Developer Response

Fixed in 199646175b42a081f049f1a26e3362c4fa878450.

2.10 Final Remarks

The DAO contracts of Yield Basis resemble much of the mechanism present in the Curve protocol. While the structure is similar, multiple modifications have been made to simplify the logic and its implementation.
Given the complexity of the contracts and the high number of severe findings present in this report, the auditors recommend strengthening the testing suite and conducting a new security review.