Electisec

August 3, 2025

Prepared for
Yield Basis

Audited by
Panda
HHK

Yield Basis

Smart Contract Security Assessment

Contents

Review Summary

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.

2 Audit Scope

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

3 Key Findings

Breakdown of Finding Impacts

Impact Level	Count
Critical	1
High	1
Medium	4
Low	4
Informational	6

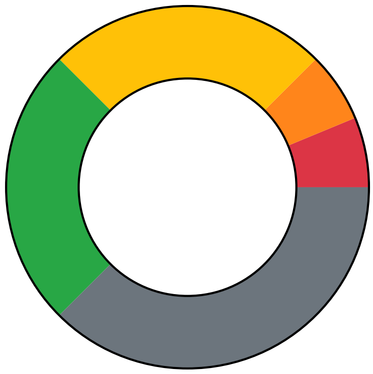

Figure 1: Distribution of security findings by impact level

4 Overall Assessment

Audit Overview

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 Audit Team

Panda, HHK

3 Audit Resources

Code repositories and documentation

Category	Mark	Description
Access Control	Low	Critical vulnerability in VotingEscrow transfer functions allows position theft due to insufficient input validation.
Mathematics	Average	ERC4626 inflation attack vulnerability in LiquidityGauge, slope change accounting issues in position merging.
Complexity	Average	Complex interactions between multiple DAO contracts with various edge cases. Infinite lock mechanics, gauge weight adjustments, and position merging introduce significant complexity that led to several vulnerabilities.
Libraries	Good	Proper use of established libraries like snekmate for ERC20/ERC721 functionality. Clean separation of concerns with library usage.
Decentralization	Good	Well-structured DAO implementation with voting mechanisms, gauge controllers, and reward distribution. Users maintain control over their positions and voting power.
Code stability	Average	Several stability issues, including merge reverts for infinite locks, transfer restrictions due to time calculations.
Documentation	Good	Code is well-documented with clear function purposes and parameter descriptions. Contract interfaces are properly defined.
Monitoring	Good	Adequate event emissions for most operations.
Testing and verification	Average	Various edge cases and attack vectors discovered during audit suggest test coverage could be improved, particularly around infinite lock scenarios and mathematical edge cases.

Table 1: Code Evaluation Matrix

4 Critical Findings

4.1 Insufficient input validation allows anyone to steal other user's voting escrow position

Attacker can sacrifice his own NFT and steal other user's position due to a missing check in `VotingEscrow.transferFrom()` and `VotingEscrow.safeTransferFrom()`.

Technical Details

`VotingEscrow.create_lock()` mints user an NFT, the `token_id` is uint256 format of `user_address`: user address is soulbound to user NFT id.
Inside the transfers functions the contract uses `erc721._is_approved_or_owner(msg.sender, token_id)` for ownership check. The check verifies if `msg.sender` is the owner of `token_id`, or if `msg.sender` has sufficient allowance for `token_id`. But there is no check to verify `token_id` == uint256 format of `owner`.
Say max lock time is reached, and attacker calls `transferFrom(victim_address, attacker_address, attacker_token_id)`. The first check will pass because attacker owns `attacker_token_id`. The second check will pass because max lock time is reached. The internal function `self._merge_positions(owner, to)` is then executed, victim's position is merged with attacker's. Then `erc721._burn(token_id)` burns attacker's NFT. In short, attacker can sacrifice his own NFT and steal victim's position. If attacker's position is tiny compared to victim's position, this attack lets him sacrifice something small and grief something big.
To amplify the impact, consider this intricately designed attack:

1. Attack prepares two wallets A and B, mints NFT for each (call them NFT A and NFT B)
2. Wallets B approves wallet A max allowance (to bypass the snekmate `erc721._is_approved_or_owner` check)
3. Attacker calls `transferFrom(victim_address, wallet_A_address, NFT_B_token_id)`
4. Victim's position is merged to wallet A's position
5. NFT B is burned but attacker can still withdraw all the asset via NFT A.

Impact

Attacker can steal any user's position.

Recommendation

Add validation to ensure the token_id corresponds to the owner parameter in both functions:

```
@external
@payable
def transferFrom(owner: address, to: address, token_id: uint256):
    assert token_id == convert(owner, uint256), "token_id must match owner"  # ← ADD THIS
    assert erc721._is_approved_or_owner(msg.sender, token_id), "erc721: caller is not token owner or approved"
    assert self._ve_transfer_allowed(owner, to), "Need max veLock"
    self._merge_positions(owner, to)
    erc721._burn(token_id)

@external
@payable
def safeTransferFrom(owner: address, to: address, token_id: uint256, data: Bytes[1_024] = b""):
```

```
    assert token_id == convert(owner, uint256), "token_id must match owner"  # ← ADD THIS
    assert erc721._is_approved_or_owner(msg.sender, token_id), "erc721: caller is not token owner or approved"
    assert self._ve_transfer_allowed(owner, to), "Need max veLock"
    self._merge_positions(owner, to)
    erc721._burn(token_id)
```

Developer Response

Fixed in 3af52777ac4a199a8e1b97f6a557ea06ab642a0d.

5 High Findings

5.1 LiquidityGauge is prone to inflation attacks such as the first share deposit one

The `LiquidityGauge` contract is vulnerable to an ERC4626 inflation attack that allows malicious actors to manipulate the share price and steal funds from subsequent depositors. The vulnerability stems from the contract's reliance on `asset.balanceOf(self)` for calculating total assets, which can be artificially inflated through direct token transfers.

Technical Details

The `LiquidityGauge` contract inherits the default `_preview_deposit` and `_preview_redeem` implementations from `erc4626.vy`, which depend on `_total_assets()`. This function simply returns the current balance of the asset token held by the contract (`asset.balanceOf(self)`). This design is problematic because it can be manipulated by an attacker transferring tokens directly to the contract outside of the intended deposit flow.

- The `LiquidityGauge` inherits `_preview_deposit` and `_preview_redeem` from `erc4626.vy`, both of which rely on `_total_assets()`
- `_total_assets()` calls `asset.balanceOf(self)`, returning the total token balance of the contract
- Direct transfers to the contract (outside the intended deposit flow) inflate `totalAssets`, skewing the share minting calculation
- The ERC4626 share calculation formula `shares = assets * totalSupply / totalAssets` becomes vulnerable when `totalAssets` is artificially inflated

Proof of Concept

A malicious actor can exploit this vulnerability through the following steps:

1. **Initial Setup**: Transfer a large amount of tokens directly to the contract (bypassing the deposit function)
2. **Minimal Deposit**: Make a small deposit (e.g., 1 wei) to mint exactly 1 share
3. **Price Manipulation**: The inflated `totalAssets` value causes the share price to become extremely high
4. **Victim Impact**: Subsequent deposits from legitimate users mint zero shares due to rounding down in the share calculation
5. **Profit Extraction**: The attacker can later redeem their single share for a disproportionate amount of the total assets

```python
def test_inflation_attack(gauge_and_lp_token):
    gauge, lp_token, reward_token, YB_token, gauge_controller = gauge_and_lp_token

    # Attacker (USER2) performs inflation attack
    with boa.env.prank(USER2):
        gauge = boa.load("contracts/dao/LiquidityGauge.vy", lp_token.address, YB_token, ADMIN, gauge_controller)
        # Step 1: Transfer tokens directly to inflate totalAssets
        lp_token.transfer(gauge.address, 2500 * 10**18)

    with boa.env.prank(USER2):
        lp_token.approve(gauge.address, 5000 * 10**18)
        # Step 2: Deposit minimal amount to mint 1 share
        gauge.deposit(2500*10**18+1, USER2)
        print("Attacker shares:", gauge.balanceOf(USER2))

    # Victim (USER) attempts to deposit
    with boa.env.prank(USER):
        lp_token.approve(gauge.address, 10000 * 10**18)
        # Multiple deposits that mint zero shares due to inflated price
        gauge.deposit(2000*10**18, USER)
        gauge.deposit(2000*10**18, USER)
        gauge.deposit(2000*10**18, USER)
        gauge.deposit(2000*10**18, USER)
        gauge.deposit(2000*10**18, USER)
        print("Victim shares:", gauge.balanceOf(USER))

    # Attacker redeems their single share
    with boa.env.prank(USER2):
        gauge.redeem(1, USER2, USER2)
        print("Attacker LP tokens balance:", lp_token.balanceOf(USER2))
        print("Victim LP tokens balance:", lp_token.balanceOf(USER))
        print("LP tokens stuck in gauge:", lp_token.balanceOf(gauge.address))
```

Expected Output:

```
Attacker shares: 1
Victim shares: 0
Attacker LP tokens balance: 12500000000000000000000
Victim LP tokens balance: 0
LP tokens stuck in gauge: 7500000000000000000000
```

Impact

Malicious users can manipulate the share price to steal funds from legitimate depositors. Victims' deposits mint zero shares, locking their funds with no claim on withdrawals. While attackers cannot fully drain the contract due to the `totalSupply + 1` denominator protection, they can still achieve significant profits by frontrunning deposits.

Recommendation

Implement a protected balance mechanism that maintains separate accounting for total deposited assets or mint dead shares and pre-deposit in the vault at deployment time.

Developer Response

Fixed in : c0d12d65ec3121e3505053f0675ba0370d3087d9

6 Medium Findings

6.1 Infinite-Lock Merge will always revert

Attempting to merge two “infinite” ve-NFT locks via `transferFrom()` or `safeTransferFrom()` will always reverts due to an out-of-range signed cast, making the core “infinite-lock transfer” feature unusable.

Technical Details

After calling `infinite_lock_toggle()`, the user lock’s end is set to `max_value(uint256)`. Users can merge locks together by transferring them to another user, by calling `transferFrom()` or `safeTransferFrom()`.
Inside the internal function `_merge_positions()` the logic is as follow:

```
slope = new_locked.amount // MAXTIME
bias  = slope * convert(new_locked.end - block.timestamp, int256)
```

This is an issue for infinite locks, since it will result in `new_locked.end - now` $2^2 - 1$, which exceeds the signed-256 range ($\pm 2^2 - 1$). And since `convert(uint256, int256)` is range-checked, this always reverts.

Impact

Medium. Users cannot transfer or merge permanent locks. This core feature is non-functional.

Recommendation

Introduce a special-case before the cast to handle infinite locks explicitly. Something of the below order.

```
def _merge_positions(owner: address, to: address):
    ...
    new_locked.amount += locked.amount
    self.locked[to] = new_locked

    if new_locked.end == max_value(uint256):
        # Special-case infinite lock: no decay
        slope = 0
        bias  = convert(new_locked.amount, int256)
    else:
        slope = new_locked.amount // MAXTIME
        bias  = slope * convert(new_locked.end - block.timestamp, int256)
    ...
```

This restores the intended “merge two infinite locks” functionality without causing an overflow revert.

Developer Response

Fixed in 6a7b17e2b5bf50f9378a21748831b35c956afc0a

6.2 `UMAXTIME` is not a multiple of 7 days

Technical Details

The constant `UMAXTIME` is widely used throughout the code to represent the maximum possible lock duration. It is currently set to 4 * 365 days. However, this duration leaves a remainder of 4 days when divided by 7, which is suboptimal since lock end times are aligned to 7-day intervals. As a result, in the `_ve_transfer_allowed()` function, a user's lock is unlikely to exactly match `max_time`. Consequently, the conditions `owner_time // WEEK * WEEK == max_time` and `to_time // WEEK * WEEK == max_time` will evaluate to false, preventing transfers from being allowed.

Impact

Medium. Users will not be able to max lock and transfer their NFTs

Recommendation

Set `UMAXTIME` to a multiple of 7 days, such as `UMAXTIME = 4 * 52 * 7 days` which would by 4 years minus 4 days.

Developer Response

This is partially expected, the lock should always be less, however fixed the `_ve_transfer_allowed()` check in 7215a47023e025a3941b29c6af5c844653bcc673.

6.3 Infinite locks won't be able to vote for gauges

The `vote_for_gauge_weights()` function uses `slope` to determine voting power which will be 0 for infinite locks.

Technical Details

When a user calls `infinite_lock_toggle()` it saves a checkpoint and sets the `slope` to 0. Later on when a user tries to vote for a gauge on the gauge controller by calling `vote_for_gauge_weights()` it will result in no power added to that gauge as the `slope` for the infinite lock is 0.

Impact

Medium. Users with infinite locks won't be able to vote, they can still toggle off the infinite lock to bypass the issue.

Recommendation

Modify the gauge voting to take into account infinite locks.

Developer Response

Fixed in: f46a125604c6f16d3522ddd4a40969dbfd3ff8e6

6.4 VotingEscrow _merge_positions Missing Checkpoint

The `_merge_positions` function in VotingEscrow bypasses the proper `_checkpoint()` mechanism, leading to incorrect slope change accounting. This results in permanent corruption of the global voting weight decay timeline and miscalculated voting weights for future periods.

7 Technical Details

The `_merge_positions` function bypasses proper slope change accounting by directly updating `user_point_history` instead of calling `_checkpoint()` for each user. When positions are merged, the function:

1. Directly sets `self.user_point_history[owner][user_epoch] = Point(bias=0, slope=0, ts=block.timesta`
2. Directly sets `self.user_point_history[to][user_epoch] = Point(bias=slope * convert(new_locked.end`

This bypasses the `slope_changes` tracking logic in `_checkpoint()` that properly schedules slope decreases at lock expiration times. The missing logic includes:

- Removing old slope changes: `old_dslope += u_old.slope` and `self.slope_changes[old_locked.end] = old_dslope`
- Adding new slope changes: `new_dslope -= u_new.slope` and `self.slope_changes[new_locked.end] = new_dslope`

Impact

Medium. The global voting weight decay timeline becomes corrupted.

Recommendation

Only allow merging to locks with similar duration.

Developer Response

Fixed in: 6a7b17e2b5bf50f9378a21748831b35c956afc0a

8 Low Findings

8.1 Disabled user's unvested tokens can't be reallocated

Technical Details

In VestingEscrow, when the owner disables a recipient during the vesting period, the unvested tokens allocated to that recipient become permanently locked in the contract.
When tokens are funded to a recipient via `fund()`, the `unallocated_supply` is decreased by the full allocation amount. If a recipient is later disabled, they can only claim tokens that have already vested. The remaining unvested tokens cannot be reallocated to other recipients because `unallocated_supply` was already reduced to account for the full original allocation, these unvested tokens become permanently inaccessible.

Impact

Low. Unvested tokens allocated to disabled recipients become permanently locked in the contract, preventing their reallocation to other recipients.

Recommendation

Add a function to reclaim unvested tokens from disabled recipients and withdraw them or return them to the `unallocated_supply` for reallocation.

Developer Response

8.2 Killed Gauges Continue to receive Emissions

Technical Details

The `set_killed()` function marks a gauge address in `is_killed[gauge] = true`.
The only place this `is_killed[gauge]` flag is consulted is in `vote_for_gauge_weights()`, preventing new votes on a killed gauge.
`_checkpoint_gauge()`, the sole place where fresh YB are minted and a gauge's weight is refreshed, never reads that flag. The first assert doesn't guard against the killed ones so the function proceeds for killed gauges exactly as for live ones.
Likewise, emit which is the public entry-point gauges use to pull their share, also omits any kill check. It calls `_checkpoint_gauge(msg.sender)` and pays out whatever weight the controller calculated.
Consequently, killed gauges continue to accrue emissions and allow users to claim tokens, bypassing the kill mechanism entirely.

Impact

Low. Killed gauges continue to draw from the inflation reserve for the full lifespan of any residual vote weight, diluting live gauges and miss-allocating YB.
If a malicious user or bribed voters pushes significant weight just before an admin kill, that gauge continues to receive emissions for months despite it being deprecated.

Recommendation

Guard Emissions in `_checkpoint_gauge()`. Before any emission logic add
`assert not self.is_killed[gauge], "Gauge is killed"`
Prevent claims on killed Gauges. In `emit()`, likewise add extra validation
`assert not self.is_killed[msg.sender], "Gauge is killed"`
Alternatively, in `set_killed()` immediately zero out that gauge's weight (and adjust the global sums) so even if any existing guard is bypassed, the killed gauge has no residual bias left to harvest.

Developer Response

Acknowledged, this is by design to be able to reverse the killing. Same logic is present in Curve.

8.3 VestingEscrow view functions incorrectly report vested amount for disabled addresses

The `_total_vested_of` function in VestingEscrow.vy does not account for the `disabled_at` timestamp, causing view functions like `vestedOf`, `balanceOf`, and `lockedOf` to return incorrect values for disabled addresses.

Technical Details

The VestingEscrow contract includes a disable mechanism where the admin can call `toggle_disable(_recipient)` to prevent an address from claiming tokens that have not yet vested at the time of disabling. This is implemented using the `disabled_at` mapping:

```
@external
def toggle_disable(_recipient: address):
    # ...
    is_disabled: bool = self.disabled_at[_recipient] == 0
    if is_disabled:
        self.disabled_at[_recipient] = block.timestamp  # Set disable timestamp
    else:
        self.disabled_at[_recipient] = 0  # Re-enable
```

The `claim` function correctly handles this by using the `disabled_at` timestamp when calculating vested amounts:

```
@external
def claim(addr: address = msg.sender):
    t: uint256 = self.disabled_at[addr]
    if t == 0:
        t = block.timestamp
    claimable: uint256 = self._total_vested_of(addr, t) - self.total_claimed[addr]
    # ...
```

owever, the view functions `vestedOf`, `balanceOf`, and `lockedOf` all call `_total_vested_of` with the default `block.timestamp` parameter, ignoring the `disabled_at` state:

```
@external
@view
def vestedOf(_recipient: address) -> uint256:
    return self._total_vested_of(_recipient)  # Uses block.timestamp by default

@external
@view
def balanceOf(_recipient: address) -> uint256:
    return self._total_vested_of(_recipient) - self.total_claimed[_recipient]  # Uses block.timestamp by default

@external
@view
def lockedOf(_recipient: address) -> uint256:
    return self.initial_locked[_recipient] - self._total_vested_of(_recipient)  # Uses block.timestamp by default
```

Impact

Low. This inconsistency leads to misleading information for disabled addresses.

Recommendation

Modify the view functions to account for the `disabled_at` timestamp when calculating vested amounts. The fix should ensure consistency with the `claim` function's behavior.

Developer Response

Fixed at: f563fb5

8.4 `CliffEscrow` can't use `VotingEscrow` infinite lock

Technical Details

`VotingEscrow` implements an `infinite_lock_toggle()` function that allows users to create or cancel ever-extending locks by setting the lock end time to `max_value(uint256)`. However, `CliffEscrow` only exposes a subset of `VotingEscrow` methods (`create_lock`, `increase_amount`, `increase_unlock_time`, `withdraw`, and `transferFrom`) but does not include `infinite_lock_toggle()` in its interface or implementation.
The missing method prevents users who interact with `VotingEscrow` through `CliffEscrow` from accessing this functionality.

Impact

Low. Users interacting with `VotingEscrow` through `CliffEscrow` cannot benefit from the infinite lock feature.

Recommendation

Add the `infinite_lock_toggle()` method to `CliffEscrow`.

```
@external
def infinite_lock_toggle():
    self._access()
    extcall VE.infinite_lock_toggle()
```

Developer Response

Fixed in 4378752a0bb17169648a8711598a18372a93de7f.

9 Gas Savings Findings

None.

10 Informational Findings

10.1 Transfer clearance checker should be enforced during deployment

The VotingEscrow contract allows veNFT transfers only when both sender and receiver hold max-duration locks and the sender has zero active votes. However, the zero-vote validation relies on an optional external checker that may not be set during deployment.

Technical Details

The `_ve_transfer_allowed()` function delegates zero-vote validation to an external `TransferClearanceChecker`:
When `transfer_clearance_checker` is unset (zero address), the zero-vote check is silently skipped, and only the max-lock condition is enforced.

Impact

Informational. If the deployer forgets to set the transfer clearance checker, the intended zero-vote requirement for transfers will not be enforced, potentially allowing transfers with active votes.

Recommendation

Consider requiring the transfer clearance checker to be set during deployment to ensure the intended transfer restrictions are always enforced. This can be done by:

- Adding the checker address as a constructor parameter
- Adding a check to ensure the checker is set before allowing transfers
- Or implementing the zero-vote validation directly in the contract rather than delegating to an external checker

Developer Response

Fixed in 2be7c58453159370c70dab6199250ddfe0b08a17.

10.2 Unused `NewGaugeWeight` Event in `GaugeController`

Technical Details

`GaugeController` defines a `NewGaugeWeight` event but never emits it. Neither `_checkpoint_gauge()` nor `vote_for_gauge_weights()` ever logs `NewGaugeWeight`. Observers cannot reconstruct historic weight adjustments except via indirect state reads.

Impact

Emit the logs to ensure greater on-chain transparency.

Recommendation

Emit `NewGaugeWeight` after the gauge weights are updated.

Developer Response

Removed the event. Fixed in 37f008a2edcea23b264fe5301ed9c370754740b0.

10.3 Missing Event for `recover_token()` in CliffEscrow

Technical Details

The `recover_token()` function in CliffEscrow allows the designated `RECIPIENT` to sweep any ERC-20 (except YB) out of the contract, but it emits no event to record this action. As a result, any token recovery is completely invisible on-chain.

Impact

Informational. Missing on-chain evidence makes it impossible for off-chain watchers or block explorers to detect token sweeps.

Recommendation

Emit an event indicating token recoveries `log TokenRecovered(token=token.address, to=to, amount=amount)`.

Developer Response

Fixed in cf785ae276f308ba66e40fee8dc3da7bc97dff6c.

10.4 `toggle_disable()` function variables are named in the reversed order

Technical Details

Inside `toggle_disable()` when the address is disabled from claiming, `is_disabled` should be true; and in the reverse scenario (address enabled to claim), `is_disabled` should be `false`. As of now, `is_disabled` holds the boolean value for the currently "not disabled" address as true instead of false.
This does not impact the function to toggle, but the state described by the variable is the opposite of what it truly is.

Impact

Informational: Improve code readability and monitoring via event emissions.

Recommendation

Rename `is_disabled` to `currently_enabled` OR flip the booleans stored.

Developer Response

Fixed in 3b0dc8871f58d5d31aec09120a0208557b7294a3.

10.5 Spelling error in `CliffEscrow.vy`

Technical Details

The Cliff Escrow contract contains a spelling error in the `init` parameter declaration.

```
@deploy
def __init__(token: IERC20, unlock_time: uint256, ve: VotingEscrow, gc: GaugeController,
    RECIPIENT = recepient
    YB = token
    VE = ve
    GC = gc
    assert unlock_time > block.timestamp
    UNLOCK_TIME = unlock_time
    extcall token.approve(ve.address, max_value(uint256))
```

Incorrect spelling: recepient Correct spelling: recipient

Impact

Informational.

Recommendation

Rename the parameter.

Developer Response

TODO : leave empty and fill in with dev response

10.6 Incorrect comment in `YB.sol`'s `__init__()`

Technical Details

Inside `__init__()` there is a comment:

```
# * set_minter(deployer, False)
```

Which may not work as `set_minter()` has a check `assert minter != msg.sender, "erc20: minter is owner address"` which will make that call revert.

Impact

Informational.

Recommendation

Remove this comment and the line from the potential deploy script.

Developer Response

Fixed in 2cd7c1b51fbebee771a521307838f978a87e68fe.