MixBytes()

Yield Basis Hybrid Vault Security Audit Report

MARCH 26, 2026

Table of Contents

1. Introduction

1.1 Disclaimer

The audit makes no statements or warranties regarding the utility, safety, or security of the code, the suitability of the business model, investment advice, endorsement of the platform or its products, the regulatory regime for the business model, or any other claims about the fitness of the contracts for a particular purpose or their bug-free status.

1.2 Executive Summary

HybridVault allows a user to manage leveraged positions across multiple YB markets while maintaining stablecoin (crvUSD) backing through deposits into a crvUSD vault (e.g., scrvUSD). Each user can create their own HybridVault instance via the HybridVaultFactory, which deploys minimal proxy clones of the HybridVault implementation. The HybridVault enables a user to deposit collateral assets into YB markets, take on debt, stake positions to earn YB rewards, and automatically manage the required crvUSD backing for their positions through an integrated crvUSD vault. The HybridFactoryOwner contract provides administrative functions for managing pool limits, stablecoin allocations, and factory settings.

This audit covered HybridVault, HybridVaultFactory and HybridFactoryOwner. Other protocol components (AMM, LT, DAO contracts, etc.) were not included in the review, although they interact with the audited contracts through interfaces.

In this audit, we focused on analyzing potential attacks related to the integration of HybridVault and LT. We also went through our detailed checklist, covering other aspects such as business logic, common ERC20 issues, interactions with external contracts, integer overflows, reentrancy attacks, access control, typecast pitfalls, rounding errors and other potential issues.

No significant vulnerabilities were found.

Key notes and recommendations:

1. **Front-Running DoS via LT Token Transfer in** `HybridVault.deposit()`: An attacker can front-run deposit transactions by sending LT tokens directly to the vault, increasing the required CRVUSD amount (calculated based on vault's LT balance). If the user approved the exact required CRVUSD amount (which is less than the amount after the front-run), the transaction will revert. This should be considered on the frontend.
2. The `HybridFactoryOwner` contract functions (`fill_staker_vpool()`, `set_allocator()`, `set_agg()`, `set_flash()`) require ADMIN access and call corresponding Factory.vy functions that also require ADMIN. This design assumes that `HybridFactoryOwner` will be set as the admin/owner of the Factory contract, allowing it to execute these administrative operations.
3. The `lt_distribute_borrower_fees()` function in `HybridFactoryOwner` requires ADMIN access, while the underlying LT function can be called by anyone. When called by the admin, it allows setting an arbitrary discount.

4. There is no mechanism to pause vulnerable HybridVault instances. If a vulnerability is discovered, the admin cannot stop affected vaults from operating, potentially leaving user funds at risk until users manually withdraw.
5. The `HybridVaultFactory.set_vault_impl()` function uses minimal proxy clones, which makes the architecture more decentralized and trustless since each vault's implementation cannot be changed after deployment. However, if there is a need to update all vaults simultaneously (e.g., in case of critical vulnerabilities), a beacon proxy pattern may be preferable.
6. We recommend adding fuzz tests to verify that `assets_for_crvusd()` does not revert and that `_required_crvusd_for(market, 1e18, 1e18)` never returns zero (which would cause a division-by-zero revert in `assets_for_crvusd()`).
7. If the AMM oracle fails for a specific pool, `HybridVault.withdraw()` for that pool will also fail. If, at the same time, the AMM has not been killed, then `HybridVault.emergency_withdraw()` for that pool will also fail.
8. `emergency_withdraw()` does not support unstaking; the user must call `unstake()` separately before calling `emergency_withdraw()` when LT is staked.
9. `emergency_withdraw()` with `crvusd_from_wallet=False` does not return excess stablecoins to the user in the same tx; the user can fully exit the pool first, then call `withdraw_scrvusd()` or `redeem_crvusd()` to retrieve the freed crvUSD.

1.3 Project Overview

Summary

Title	Description
Client	Yield Basis
Category	Leveraged Farming
Project	Hybrid Vault
Type	Vyper
Platform	EVM
Timeline	23.01.2026 - 25.03.2026

Scope of Audit

File	Link
contracts/HybridVault.vy	contracts/HybridVault.vy
contracts/HybridVaultFactory.vy	contracts/HybridVaultFactory.vy
contracts/HybridFactoryOwner.vy	contracts/HybridFactoryOwner.vy

Versions Log

Date	Commit Hash	Note
23.01.2026	cd1217fbc9cac30b6cc274d494c96fe487efd449	Initial Commit
09.02.2026	655b7c1740390a56c5023b1fe94968346d67b018	Commit for re-audit
23.02.2026	24a986d819db4cd786afdf670f3191497b9a4067	Commit for re-audit
04.03.2026	5ac61b50a3905daf14cf1450f434c8d449577329	Commit for re-audit
10.03.2026	4d8d8800426a201b5e251896f69a39b25a54218c	Commit for re-audit
13.03.2026	e90b804dfc0c93e6e3590f2f425c065c5f579b03	Commit for re-audit
16.03.2026	f7d3720b7ff90cfed9359745e1a5c897a5a96bb9	Commit for re-audit
20.03.2026	ad40eff3b4dbfb3f68af57e49821adb47f9e1ca1	Commit for re-audit
23.03.2026	a2b8d5ee27274731d2c00261aff91b3491259f5a	Commit for re-audit
25.03.2026	faadf56967ea1b34097c55c082c3e7df8ac1eca8	Commit for re-audit

Mainnet Deployments

The deployment verification will be conducted later after the full deployment of the protocol into the mainnet.

1.4 Security Assessment Methodology

Project Flow

Stage	Scope of Work
Interim audit	**Project Architecture Review:** · Review project documentation · Conduct a general code review · Perform reverse engineering to analyze the project's architecture based solely on the source code · Develop an independent perspective on the project's architecture · Identify any logical flaws in the design OBJECTIVE: UNDERSTAND THE OVERALL STRUCTURE OF THE PROJECT AND IDENTIFY POTENTIAL SECURITY RISKS.
	Code Review with a Hacker Mindset: · Each team member independently conducts a manual code review, focusing on identifying unique vulnerabilities. · Perform collaborative audits (pair auditing) of the most complex code sections, supervised by the Team Lead. · Develop Proof-of-Concepts (PoCs) and conduct fuzzing tests using tools like Foundry, Hardhat, and BOA to uncover intricate logical flaws. · Review test cases and in-code comments to identify potential weaknesses. OBJECTIVE: IDENTIFY AND ELIMINATE THE MAJORITY OF VULNERABILITIES, INCLUDING THOSE UNIQUE TO THE INDUSTRY.
	Code Review with a Nerd Mindset: · Conduct a manual code review using an internally maintained checklist, regularly updated with insights from past hacks, research, and client audits. · Utilize static analysis tools (e.g., Slither, Mythril) and vulnerability databases (e.g., Solodit) to uncover potential undetected attack vectors. OBJECTIVE: ENSURE COMPREHENSIVE COVERAGE OF ALL KNOWN ATTACK VECTORS DURING THE REVIEW PROCESS.

Stage	Scope of Work
	Consolidation of Auditors' Reports: · Cross-check findings among auditors · Discuss identified issues · Issue an interim audit report for client review OBJECTIVE: COMBINE INTERIM REPORTS FROM ALL AUDITORS INTO A SINGLE COMPREHENSIVE DOCUMENT.
Re-audit	**Bug Fixing & Re-Audit:** · The client addresses the identified issues and provides feedback · Auditors verify the fixes and update their statuses with supporting evidence · A re-audit report is generated and shared with the client OBJECTIVE: VALIDATE THE FIXES AND REASSESS THE CODE TO ENSURE ALL VULNERABILITIES ARE RESOLVED AND NO NEW VULNERABILITIES ARE ADDED.
Final audit	**Final Code Verification & Public Audit Report:** · Verify the final code version against recommendations and their statuses · Check deployed contracts for correct initialization parameters · Confirm that the deployed code matches the audited version · Issue a public audit report, published on our official GitHub repository · Announce the successful audit on our official X account OBJECTIVE: PERFORM A FINAL REVIEW AND ISSUE A PUBLIC REPORT DOCUMENTING THE AUDIT.

1.5 Risk Classification

Severity Level Matrix

Severity	Impact: High	Impact: Medium	Impact: Low
Likehood: High	Critical	High	Medium
Likehood: Medium	High	Medium	Low
Likehood: Low	Medium	Low	Low

Impact

- **High** – Theft from 0.5% OR partial/full blocking of funds (>0.5%) on the contract without the possibility of withdrawal OR loss of user funds (>1%) who interacted with the protocol.
- **Medium** – Contract lock that can only be fixed through a contract upgrade OR one-time theft of rewards or an amount up to 0.5% of the protocol's TVL OR funds lock with the possibility of withdrawal by an admin.
- **Low** – One-time contract lock that can be fixed by the administrator without a contract upgrade.

Likelihood

- **High** – The event has a 50–60% probability of occurring within a year and can be triggered by any actor (e.g., due to a likely market condition that the actor cannot influence).
- **Medium** – An unlikely event (10–20% probability of occurring) that can be triggered by a trusted actor.
- **Low** – A highly unlikely event that can only be triggered by the owner.

Action Required

- **Critical** – Must be fixed as soon as possible.
- **High** – Strongly advised to be fixed to minimize potential risks.
- **Medium** – Recommended to be fixed to enhance security and stability.
- **Low** – Recommended to be fixed to improve overall robustness and effectiveness.

Finding Status

- **Fixed** – The recommended fixes have been implemented in the project code and no longer impact its security.
- **Partially Fixed** – The recommended fixes have been partially implemented, reducing the impact of the finding, but it has not been fully resolved.
- **Acknowledged** – The recommended fixes have not yet been implemented, and the finding remains unresolved or does not require code changes.

1.6 Summary of Findings

Findings Count

Severity	Count
Critical	0
High	0
Medium	10
Low	16

Findings Statuses

ID	Finding	Severity	Status
M-1	Broken Oracle Causes Skipped Allocation Reduction in `withdraw()`	Medium	Fixed
M-2	Unused `receiver` Parameter in `HybridVault.withdraw()`	Medium	Fixed
M-3	`HybridVault.set_crvusd_vault()` Front-Running DoS	Medium	Fixed
M-4	`HybridVault.claim_reward()` Reverts When Reward Tokens Differ Across Gauges	Medium	Fixed
M-5	`HybridVault._required_crvusd()` Reverts When Pool is Empty	Medium	Fixed
M-6	Denial of Service (DoS) on Withdrawals via Front-Running	Medium	Fixed
M-7	`totalSupply()` Read After Burning LT Shares Causes Allocation Reduction Inaccuracy	Medium	Fixed
M-8	`withdraw()` with `withdraw_stablecoins=True` Reverts Due to Overflow in `_downscale()`	Medium	Fixed
M-9	`withdraw()` Silently Ignores `withdraw_stablecoins=True` When Pool Oracle is Broken	Medium	Fixed

M-10	Skipped Solvency Check in withdraw() Allows Unbacked Pools	Medium	Fixed
L-1	Missing Approval Revocation in HybridVault.set_crvusd_vault()	Low	Fixed
L-2	No Ownership Transfer Function in HybridVault	Low	Acknowledged
L-3	Typos	Low	Fixed
L-4	Unused Code in Constructor	Low	Fixed
L-5	Unnecessary is_killed() Check in HybridVault.emergency_withdraw()	Low	Fixed
L-6	Missing Admin Liquidity Correction in emergency_withdraw() Allocation Reduction	Low	Fixed
L-7	Underflow in _remove_from_used() When Global Allocation Decreases	Low	Fixed
L-8	Removing HybridVaultFactory from limit_setters Blocks All Hybrid Vault Operations	Low	Acknowledged
L-9	withdraw() is Blocked While the Oracle is Broken	Low	Acknowledged
L-10	_redeem_crvusd(), withdraw_scrvusd(), and redeem_crvusd() Revert When Any Oracle is Broken	Low	Acknowledged
L-11	Stale crvusd_vault_required and crvusd_vault_total_required When Oracle is Broken in Another Pool	Low	Acknowledged
L-12	Potential Underflow in _allocate_stablecoins() Call in withdraw() and emergency_withdraw()	Low	Fixed
L-13	Potential Underflow in update_vault_required() When Decreasing crvusd_vault_total_required	Low	Acknowledged
L-14	Desynchronization of crvusd_vault_required from Actual _required_crvusd()	Low	Acknowledged
L-15	Skipped Allocation Reduction in emergency_withdraw() After Favorable Unwind Under Oracle Failure	Low	Acknowledged
L-16	Division by Zero in assets_for_crvusd()	Low	Acknowledged

2. Findings Report

2.1 Critical

Not Found

2.2 High

Not Found

2.3 Medium

M-1	Broken Oracle Causes Skipped Allocation Reduction in withdraw()		
Severity	Medium	**Status**	Fixed in f7d3720b

Description

When a vault has positions in two pools (A and B) and the AMM oracle (value_oracle()) fails for pool B, calling withdraw() on pool A skips the allocation reduction entirely — even though pool A's oracle works correctly and the LT withdrawal succeeds.

This happens because _required_crvusd() iterates over **all** used_vaults. If any pool's oracle is broken, _pool_crvusd() returns max_value(uint256) for that pool, causing the entire _required_crvusd() to return max_value(uint256):

```
required_before: uint256 = self._required_crvusd()
# max_value(uint256) due to pool B
# ... LT shares for pool A are successfully withdrawn here ...
required_after: uint256 = self._required_crvusd()
# max_value(uint256) due to pool B

if required_before > required_after:
  # False — skipped entirely
  # allocation reduction for pool A never executes
```

contracts/HybridVault.vy

Assets are successfully withdrawn from pool A, but stablecoin_allocation[pool_A] and the global stablecoin_allocation() on the LT are never reduced. Once the oracle recovers, the allocation remains inflated relative to the vault's actual position. This inflated allocation can be exploited by other participants outside the HybridVault system.

Recommendation

We recommend handling the max_value(uint256) case explicitly in withdraw().

M-2	Unused receiver Parameter in HybridVault.withdraw()		
Severity	Medium	**Status**	Fixed in 655b7c17

Description

In the HybridVault.withdraw() function, the receiver parameter is declared but never used. Instead, the function always sends withdrawn assets to msg.sender when calling LT.withdraw().

- contracts/HybridVault.vy

Recommendation

We recommend using receiver instead of msg.sender when transferring withdrawn assets to the user.

M-3	HybridVault.set_crvusd_vault() Front-Running DoS		
Severity	Medium	**Status**	Fixed in 655b7c17

Description

The HybridVault owner can change the crvUSD vault via set_crvusd_vault() only if the vault is empty:

```
assert staticcall self.crvusd_vault.balanceOf(
    self) == 0, "Current vault not empty"
```

· contracts/HybridVault.vy

A griefer can repeatedly send 1 wei share to the user's vault before the user attempts to change the vault, ensuring the balance is never strictly equal to zero and causing set_crvusd_vault() to revert.

Recommendation

We recommend allowing atomic withdrawal of excess crvUSD vault shares and vault change in a single function, ensuring the required_crvusd() invariant is preserved.

M-4	HybridVault.claim_reward() Reverts When Reward Tokens Differ Across Gauges		
Severity	Medium	**Status**	Fixed in 655b7c17

Description

The HybridVault.claim_reward() function iterates through all used_vaults and calls market.staker.claim(token, self) for each gauge. When a reward token (other than YB) is not added to a gauge via add_reward(), the LiquidityGauge._get_vested_rewards() function reverts with "No reward" due to the assertion assert self.rewards[token].distributor != empty(address).

If a user has positions in multiple gauges with different reward tokens, attempting to claim a token that exists in only some gauges will cause the entire transaction to revert, preventing reward claims from any gauge.

- contracts/HybridVault.vy
- contracts/dao/LiquidityGauge.vy

Recommendation

We recommend modifying claim_reward() to handle cases where a reward token is not available in all gauges.

M-5	HybridVault._required_crvusd() Reverts When Pool is Empty		
Severity	Medium	**Status**	Fixed in 655b7c17

Description

The _required_crvusd() function performs division operations on liquidity.total and lt_total without checking if these values are zero. If all funds are withdrawn from a pool (e.g., pool.lt.totalSupply() == 0 or pool.lt.liquidity().total == 0), the function will revert due to division by zero.

Since _required_crvusd() is called in multiple critical functions (deposit(), withdraw(), _withdraw_crvusd(), _redeem_crvusd(), withdraw_scrvusd()), this prevents users from performing any operations on other markets even if they still have positions there. The _remove_from_used() function exists but is never called in withdraw(), so empty pools remain in used_vaults.

- contracts/HybridVault.vy
- contracts/HybridVault.vy

Recommendation

We recommend accounting for the case when one of the pools is empty in _required_crvusd().

M-6	Denial of Service (DoS) on Withdrawals via Front-Running		
Severity	Medium	**Status**	Fixed in 655b7c17

Description

Withdrawals can be blocked by front-running the transaction with a small transfer of LT to the contract. This manipulation increases `_required_crvusd` above the contract's current crvUSD balance, ensuring that a small amount of LT remains after the withdrawal attempt. Since withdrawals specify an exact number of shares (up to the user's balance), front-running with an LT transfer forces a full withdrawal to be treated as a partial one. During the transaction, both the crvUSD balance and `_required_crvusd` are reduced. However, because the balance decreases by the difference between `required_before` and `required_after`, the solvency check within `_withdraw_crvusd` will fail, causing the transaction to revert.
Example Scenario:

1. Following a deposit, the contract holds 100 crvUSD, with `_required_crvusd == 100` and minted LT == X (we assume that `_downscale` equals 1e18, which does not affect the validity of our reasoning).

2. A depositor attempts to withdraw all X shares.

3. An attacker front-runs this transaction by transferring a small amount of LT to the contract, raising `_required_crvusd` to 110.

4. During the victim's transaction execution, `_required_crvusd` decreases to 10, and the crvUSD balance drops to 0 (calculated as `required_before - required_after`). Consequently, the check inside `_withdraw_crvusd` fails due to insufficient funds, reverting the transaction.

- contracts/HybridVault.vy
- contracts/HybridVault.vy

Recommendation

We recommend calculating the crvUSD withdrawal amount with `staticcall self.crvusd_vault.balanceOf(self) - required_after`. This ensures that `required_after` of crvUSD always remains on the contract. Additionally, we recommend maintaining an internal record of LT minted via the HybridVault. This ensures that transfers of external LT do not alter `required_crvusd`.

M-7	totalSupply() Read After Burning LT Shares Causes Allocation Reduction Inaccuracy		
Severity	Medium	Status	Fixed in f7d3720b

Description

In the withdraw() function, lt_supply is read via market.lt.totalSupply() **after** the LT shares have been burned by market.lt.withdraw(). This occurs in the fallback branch where required_before == max_value(uint256) and pool_crvusd_before == max_value(uint256) (i.e., when value_oracle() reverts for all pools including the current one).
Since lt_supply is already reduced by lt_shares at the time of reading, the fraction lt_shares / lt_supply is inflated. For example, withdrawing 50 out of 100 shares yields 50 / 50 = 100% reduction instead of the correct 50 / 100 = 50%. In the extreme case of withdrawing all shares, lt_supply becomes zero, causing a division-by-zero revert.

Recommendation

We recommend reading totalSupply() before the burn call.

M-8	withdraw() with withdraw_stablecoins=True Reverts Due to Overflow in _downscale()		
Severity	Medium	**Status**	Fixed in f7d3720b

Description

In the withdraw() function, when required_before == max_value(uint256) (another pool's oracle is broken) but the current pool's oracle works (pool_crvusd_before != max_value(uint256)), calling with withdraw_stablecoins=True triggers _withdraw_crvusd():

```
if required_before == max_value(uint256):
    if pool_crvusd_before != max_value(uint256):
        pool_crvusd_after: uint256 = self._pool_crvusd(market)
        if pool_crvusd_before > pool_crvusd_after:
            ...
            if withdraw_stablecoins:
                self._withdraw_crvusd(
                    self._downscale(...), receiver)
```

contracts/HybridVault.vy

Inside _withdraw_crvusd(), the invariant check calls self._downscale(self._required_crvusd()):

```
assert self._crvusd_available(
) >= self._downscale(self._required_crvusd())
```

contracts/HybridVault.vy

Since _required_crvusd() returns max_value(uint256) (due to the broken oracle on another pool), _downscale() attempts to compute max_value(uint256) * stablecoin_fraction, which overflows and reverts.

As a result, users cannot withdraw with withdraw_stablecoins=True from a healthy pool whenever any other pool in used_vaults has a broken oracle.

Recommendation

We recommend handling the max_value(uint256) case in _withdraw_crvusd() before calling _downscale().

M-9	withdraw() Silently Ignores withdraw_stablecoins=True When Pool Oracle is Broken		
Severity	Medium	**Status**	Fixed in f7d3720b

Description

In the withdraw() function, when both required_before == max_value(uint256) and pool_crvusd_before == max_value(uint256) (i.e., the oracle is broken for the pool being withdrawn from), the code falls into the else branch that calculates allocation reduction proportionally but completely ignores the withdraw_stablecoins flag:

```
lt_supply: uint256 = staticcall market.lt.totalSupply()
reduction = min(
    previous_allocation * lt_shares // lt_supply,
    self.stablecoin_allocation[pool_id])
```

contracts/HybridVault.vy

If a caller (e.g., an external integrator) sets withdraw_stablecoins=True expecting either crvUSD to be returned or the transaction to revert, the function silently succeeds without withdrawing any crvUSD. This breaks the expected contract behavior and can lead to integration issues, as callers have no way to know that the stablecoin withdrawal was skipped.

Recommendation

We recommend reverting when withdraw_stablecoins is True if _withdraw_crvusd() cannot be called due to a broken oracle.

M-10	Skipped Solvency Check in withdraw() Allows Unbacked Pools		
Severity	Medium	**Status**	Fixed in ad40eff3

Description

In the withdraw() function, when the oracle is broken for another pool (required_before == max_value(uint256) or required_after == max_value(uint256)) but the current pool's oracle works, _withdraw_crvusd() is called with post_check_crvusd=False:

```
if withdraw_stablecoins:
    self._withdraw_crvusd(
        self._downscale_w(pool_id, pool_crvusd_before - pool_crvusd_after, False),
        receiver, False)
```

contracts/HybridVault.vy

The post_check_crvusd=False flag was introduced to avoid the overflow in _downscale(self._required_crvusd()) when _required_crvusd() returns max_value(uint256). However, this completely disables the solvency invariant check:

```
if post_check_crvusd:
    assert self._crvusd_available(
    ) >= self._downscale(self._required_crvusd())
```

If the global stablecoin_fraction has increased since the user deposited, the withdrawal amount is computed using the new (higher) downscale factor, allowing the user to withdraw more stablecoin for the same pool than they initially deposited. In the worst case, this allows a user to withdraw all available crvUSD from the vault, leaving their other pools unbacked.

Recommendation

We recommend implementing a solvency check that handles the max_value(uint256) case without overflowing.

Client's Commentary:

Mixbytes:

The current implementation caches the downscale factor per pool via _downscale_w(). However, if the admin lowers stablecoin_fraction from a high value to a low one, the cached factor remains stale, and withdrawals may release more crvUSD than the new fraction would allow, potentially leaving other pools unbacked.

2.4 Low

L-1	Missing Approval Revocation in HybridVault.set_crvusd_vault()		
Severity	Low	**Status**	Fixed in 5ac61b50

Description

When changing the crvUSD vault via set_crvusd_vault(), the function sets approval on the new vault but does not revoke approval on the old vault. If the old vault was compromised or vulnerable, it retains unlimited approval on the HybridVault's CRVUSD tokens.

- contracts/HybridVault.vy

Recommendation

We recommend revoking approval on the old vault when setting approval on the new vault.

L-2	No Ownership Transfer Function in HybridVault		
Severity	Low	**Status**	Acknowledged

Description

The HybridVault contract does not provide a function to transfer ownership to a new address. The owner is set during initialization and cannot be changed afterwards. If the vault owner loses access to their private key or wants to transfer control to another address, they cannot do so, potentially resulting in permanently locked funds.

All critical functions in the vault require msg.sender == self.owner, including deposit(), withdraw(), stake(), unstake(), claim_reward(), redeem_crvusd(), withdraw_scrvusd(), and recover_tokens(). Without the ability to transfer ownership, users have no way to recover access if they lose their private key.

- contracts/HybridVault.vy
- contracts/HybridVault.vy

Recommendation

We recommend implementing an ownership transfer function (e.g., transfer_ownership(new_owner: address)) that allows the current owner to transfer control to a new address. This function should follow a two-step transfer pattern (similar to OpenZeppelin's Ownable2Step) to prevent accidental transfers to invalid addresses.

L-3	Typos		
Severity	Low	**Status**	Fixed in 655b7c17

Description

Small number of typos and documentation inconsistencies were found in the codebase:

1. contracts/HybridVault.vy

2. contracts/HybridVault.vy: The documentation states that it "Returns the maximum of personal limit and global factory limit", but the actual implementation in _pool_limits() returns the sum of these values, not the maximum.

Recommendation

We recommend fixing the typos.

L-4	Unused Code in Constructor		
Severity	Low	**Status**	Fixed in 24a986d8

Description

In the HybridVault constructor, the variable lev_ratio is computed but never used anywhere in the contract:

```
lev_ratio: uint256 = leverage**2 * 10**18 // denominator**2
```

contracts/HybridVault.vy

Recommendation

We recommend removing the unused lev_ratio variable.

L-5	Unnecessary is_killed() Check in HybridVault.emergency_withdraw()		
Severity	Low	**Status**	Fixed in 24a986d8

Description

The HybridVault.emergency_withdraw() function checks that the LT pool is killed before allowing emergency withdrawal:

```
assert staticcall market.lt.is_killed(), "Not killed"
```

contracts/HybridVault.vy

However, the underlying LT.emergency_withdraw() function works for both killed and non-killed pools. The LT contract handles both cases with different logic paths, but the function is fully functional regardless of the pool's killed status.

This unnecessary check prevents users from using emergency_withdraw() when the pool is not killed, even though the LT contract supports it.

Recommendation

We recommend removing the is_killed() check from HybridVault.emergency_withdraw() to allow emergency withdrawals even when the pool is not killed.

L-6	Missing Admin Liquidity Correction in emergency_withdraw() Allocation Reduction		
Severity	Low	**Status**	Fixed in 5ac61b50

Description

In the HybridVault.emergency_withdraw() function, when value_oracle() reverts (required_before == max_value(uint256)), the stablecoin allocation reduction is calculated proportionally to lt_shares / lt_supply:

```
lt_supply: uint256 = staticcall market.lt.totalSupply()
reduction: uint256 = previous_allocation * lt_shares // lt_supply
self._allocate_stablecoins(market.lt, previous_allocation - reduction)
```

contracts/HybridVault.vy

However, this calculation does not account for the admin liquidity correction factor (liquidity.total - max(liquidity.admin, 0)) / liquidity.total that is used in _required_crvusd():

```
crvusd_amount = crvusd_amount * (
    liquidity.total - convert(
      max(liquidity.admin, 0), uint256
    )
  ) // liquidity.total * lt_shares // lt_total
```

contracts/HybridVault.vy

When liquidity.admin > 0, the effective share of crvUSD attributable to a given number of LT shares is smaller than the raw proportional share. As a result, the allocation reduction in emergency_withdraw() may be larger than necessary, leading to an incorrect (over-reduced) stablecoin allocation for the LT pool.

Recommendation

We recommend either improving the formula for allocation reduction in case of value_oracle() revert, or documenting this case explicitly.

L-7	Underflow in _remove_from_used() When Global Allocation Decreases		
Severity	Low	**Status**	Fixed in 24a986d8

Description

In the _remove_from_used() function, the stablecoin allocation reduction is calculated as:

```
remaining_allocation: uint256 = self.stablecoin_allocation[pool_id]
if remaining_allocation > 0:
    market: Market = staticcall FACTORY.markets(pool_id)
    previous_allocation: uint256 = staticcall market.lt.stablecoin_allocation()
    self._allocate_stablecoins(market.lt, previous_allocation - remaining_allocation)
    self.stablecoin_allocation[pool_id] = 0
```

contracts/HybridVault.vy

If the global market.lt.stablecoin_allocation() has decreased below the vault's tracked stablecoin_allocation[pool_id] (e.g., due to other vaults reducing their allocations or external changes), the subtraction previous_allocation - remaining_allocation will underflow and revert. This prevents the user from exiting the pool.

Recommendation

We recommend using safe subtraction: previous_allocation - min(previous_allocation, remaining_allocation).

L-8	Removing HybridVaultFactory from limit_setters Blocks All Hybrid Vault Operations		
Severity	Low	**Status**	Acknowledged

Description

The HybridVault._allocate_stablecoins() calls HybridVaultFactory.lt_allocate_stablecoins(), which in turn calls HybridFactoryOwner.lt_allocate_stablecoins():

```
# HybridVaultFactory.vy
def lt_allocate_stablecoins(lt: address, limit: uint256):
    assert self.vault_to_user[HybridVault(msg.sender)] \
      != empty(address), "Only vaults can call"
    extcall (staticcall FACTORY.admin())
      .lt_allocate_stablecoins(lt, limit)
```

contracts/HybridVaultFactory.vy

The HybridFactoryOwner.lt_allocate_stablecoins() requires the caller to be either ADMIN or a registered limit_setter:

```
assert msg.sender == ADMIN \
  or self.limit_setters[msg.sender], "Access"
```

contracts/HybridFactoryOwner.vy

If the admin removes HybridVaultFactory from limit_setters (via set_limit_setter(factory, False)), all calls to _allocate_stablecoins() will revert. Since _allocate_stablecoins() is called in deposit(), withdraw(), emergency_withdraw(), and _remove_from_used(), this effectively blocks all deposit and withdrawal operations across every deployed HybridVault.

Recommendation

We recommend documenting this dependency clearly and considering additional safeguards to prevent accidental removal of HybridVaultFactory from limit_setters.

Client's Commentary:

Known, intended

L-9	withdraw() is Blocked While the Oracle is Broken		
Severity	Low	**Status**	Acknowledged

Description

When the AMM oracle (value_oracle()) is broken, _required_crvusd() returns max_value(uint256). In this scenario, calling withdraw() fails. The emergency_withdraw() requires is_killed() to be true, but a broken oracle does not kill the pool.
As a result, while the oracle is broken, the user's crvUSD is completely locked in the vault with no way to withdraw it.

- contracts/HybridVault.vy

Recommendation

We recommend considering a flow to withdraw crvUSD for the case when the user wants to completely exit the LT, but the oracle is broken.

Client's Commentary:

If the AMM oracle fails, we will manually kill this AMM, so emergency_withdraw() will work.

L-10	_redeem_crvusd(), withdraw_scrvusd(), and redeem_crvusd() Revert When Any Oracle is Broken		
Severity	Low	**Status**	Acknowledged

Description

When any pool's oracle is broken, _required_crvusd() returns max_value(uint256), and _downscale(max_value(uint256)) overflows, causing _redeem_crvusd(), withdraw_scrvusd(), and redeem_crvusd() to revert. As a result, the vault owner cannot manage excess crvUSD (e.g., withdraw surplus crvUSD vault shares) while any oracle is down, even if the vault has more than enough backing for all positions.

Recommendation

We recommend handling the max_value(uint256) case in the solvency check across all crvUSD management functions.

Client's Commentary:

Mixbytes:

Users can still withdraw funds by first fully exiting the pool with the broken oracle (via `emergency_withdraw()`), after which `_required_crvusd()` no longer returns `max_value(uint256)` and the crvUSD management functions resume working normally.

L-11	Stale crvusd_vault_required and crvusd_vault_total_required When Oracle is Broken in Another Pool		
Severity	Low	**Status**	Acknowledged

Description

When a vault has positions in multiple pools and the AMM oracle (value_oracle()) is broken for one pool, calling withdraw() or emergency_withdraw() on another pool skips the update_vault_required() call. This happens because required_after is computed as _required_crvusd(), which returns max_value(uint256) when any pool in used_vaults has a broken oracle. The condition required_after != max_value(uint256) is false, so the Factory is never updated.

- contracts/HybridVault.vy
- contracts/HybridVault.vy

As a result, crvusd_vault_required[vault] and crvusd_vault_total_required[crvusd_vault] remain inflated. If a vault limit is set via set_crvusd_vault_limit(), other vaults may be blocked from depositing with "Beyond vault limit" even when capacity exists. The stale state persists until the affected vault performs another operation (deposit/withdraw) while all oracles are healthy, or until the vault calls set_crvusd_vault(). If a vault fully exits and never interacts again, the inflation is permanent.

Recommendation

We recommend calling update_vault_required() with a computed value when required_after == max_value(uint256) — for example, by recalculating _required_crvusd() after _remove_from_used() if the current pool was the last position, or by using a per-pool delta when the current pool's oracle works.

Client's Commentary:

Mixbytes:

When a vault has positions in Pool A and Pool B and Pool B's oracle breaks, withdrawing from Pool A still skips `update_vault_required()` because `_required_crvusd()` returns `max_value(uint256)` due to Pool B — the Factory retains the stale inflated value until Pool B is also exited.

Fully exiting Pool B fixes the stale state for both pools at once: `_remove_from_used()` removes Pool B before `_required_crvusd()` is called, so the Factory gets updated with the correct requirement reflecting that both Pool A and Pool B have been exited.

However, if the user leaves dust when exiting from pools, the `_remove_from_used()` function will not be called and the stale state will remain.

Client:

Acknowledged.

L-12	Potential Underflow in _allocate_stablecoins() Call in withdraw() and emergency_withdraw()		
Severity	Low	Status	Fixed in a2b8d5ee

Description

In withdraw() and emergency_withdraw(), when reducing the stablecoin allocation, the code calls _allocate_stablecoins(market.lt, previous_allocation - reduction). Here previous_allocation is the global market.lt.stablecoin_allocation() (sum across all vaults), while reduction is bounded only by this vault's self.stablecoin_allocation[pool_id]. If other vaults have already reduced the global allocation (e.g., via their own withdrawals), previous_allocation can be lower than reduction, causing an underflow and reverting the transaction. This blocks the user from withdrawing.

- contracts/HybridVault.vy (withdraw)
- contracts/HybridVault.vy (emergency_withdraw)

Recommendation

We recommend using previous_allocation - min(reduction, previous_allocation).

L-13	Potential Underflow in update_vault_required() When Decreasing crvusd_vault_total_required		
Severity	Low	**Status**	Acknowledged

Description

In HybridVaultFactory.update_vault_required(), when new_required < old_required, the code subtracts the decrease from the total:

```
decrease: uint256 = old_required - new_required
self.crvusd_vault_total_required[crvusd_vault] -= decrease
```

If crvusd_vault_total_required[crvusd_vault] is lower than decrease, the subtraction underflows and reverts, blocking the vault from completing its operation.
· contracts/HybridVaultFactory.vy

Recommendation

We recommend using safe subtraction: self.crvusd_vault_total_required[crvusd_vault] -= min(decrease, self.crvusd_vault_total_required[crvusd_vault]).

Client's Commentary:

Acknowledged. The invariant crvusd_vault_total_required[cv] == Σ crvusd_vault_required[hv] is maintained across all call sites, making underflow provably impossible

L-14	Desynchronization of crvusd_vault_required from Actual _required_crvusd()		
Severity	Low	**Status**	Acknowledged

Description

crvusd_vault_required[vault] and crvusd_vault_total_required[crvusd_vault] are updated only when vaults call update_vault_required() during deposit(), withdraw(), or emergency_withdraw(). The actual required amount depends on oracle prices, liquidity.admin, and stablecoin_fraction, which can change between transactions. As a result, the tracked values may drift from the real _downscale(_required_crvusd()), causing the limit check to be approximate rather than exact.

- contracts/HybridVaultFactory.vy
- contracts/HybridVaultFactory.vy

Recommendation

We recommend documenting that crvusd_vault_total_required is an approximate, eventually-consistent value used for limit enforcement, not a real-time exact sum of all vaults' required amounts.

Client's Commentary:

accepted, limit check is indeed approximate

L-15	Skipped Allocation Reduction in emergency_withdraw() After Favorable Unwind Under Oracle Failure		
Severity	Low	**Status**	Acknowledged

Description

In HybridVault.emergency_withdraw(), when oracle-based accounting is unavailable (required_before == max_value(uint256) or required_after == max_value(uint256)) and value_oracle() also reverts for the withdrawn pool, the stablecoin allocation reduction falls back to a raw crvUSD balance heuristic:

```
reduction = crvusd_before - min(remaining_crvusd, crvusd_before)
if stablecoin_fraction > 0:
    reduction = reduction * 10**18 // stablecoin_fraction
reduction = min(reduction, self.stablecoin_allocation[pool_id])
```

contracts/HybridVault.vy

If the emergency unwind is favorable (e.g., overcollateralized position produces a positive stables_to_return from the LT), remaining_crvusd >= crvusd_before, making the computed reduction zero. As a result, stablecoin_allocation[pool_id] remains unchanged even though LT shares were burned and actual position exposure was reduced.

This creates a transient accounting inconsistency between the actual position size and the tracked allocation cap, leaving the cap artificially high after partial emergency withdrawals under oracle failure.

The practical impact is limited by several factors:

- Currently, emergency_withdraw() reverts if the AMM is not killed; if the AMM is already killed, excess allocation for that pool is inconsequential since no new borrowing can occur.
- Full withdrawals self-correct via _remove_from_used(), which unconditionally zeros out the allocation.
- The sticky allocation is a permissive cap (too high), not a balance — it does not directly enable fund extraction.
- The scenario requires oracle failure, a favorable unwind, and a partial withdrawal simultaneously.

Note that an alternative approach — using a share-proportional reduction (stablecoin_allocation[pool_id] * lt_shares // lt_supply) — avoids the zero-reduction edge case but may over-reduce the cap for asymmetric positions (heavy collateral, low debt), potentially interfering with subsequent operations under the same degraded oracle conditions.

Recommendation

We recommend documenting this behavior.

Client's Commentary:

accepted

L-16	Division by Zero in assets_for_crvusd()		
Severity	Low	**Status**	Acknowledged

Description

In assets_for_crvusd(), the result is computed by dividing by crvusd_for_test:

```
crvusd_for_test: uint256 = self._downscale(
    self._required_crvusd_for(market, test_assets, test_debt)[1])
return effective_crvusd * test_assets // crvusd_for_test
```

• contracts/HybridVault.vy

If value_in_amm is zero (empty AMM) or the product of _required_crvusd_for() and stablecoin_fraction is below 10**18, _downscale() rounds to zero, and the division reverts.

Recommendation

We recommend returning 0 when crvusd_for_test == 0.

Client's Commentary:

Impossible event.

3. About MixBytes

MixBytes is a leading provider of smart contract auditing and blockchain security research, helping Web3 projects enhance their security and reliability.

Since its inception, MixBytes has been dedicated to safeguarding innovation in **DeFi** through rigorous audits and cutting-edge technical research.

Our team brings together experienced engineers, security auditors, and blockchain researchers with deep expertise in smart contract security and protocol design.

With years of proven experience in Web3, MixBytes combines in-depth technical excellence with a proactive, security-first approach.

Why MixBytes

- **Proven Track Record:** Trusted by leading blockchain protocols such as **Lido**, **Aave**, **Curve**, **1inch**, **Fluid**, **Gearbox**,**Resolv**, and others. MixBytes has successfully secured billions of dollars in digital assets.
- **Technical Expertise:** Our auditors and researchers hold advanced degrees in cryptography, cybersecurity, and distributed systems, backed by years of hands-on experience.
- **Innovative Research:** MixBytes actively contributes to blockchain security research and open-source tools, sharing insights with the global community.

Our Services

- **Smart Contract Audits:** Comprehensive security assessments of DeFi protocols to detect and mitigate vulnerabilities before deployment.
- **Blockchain Research:** In-depth technical studies, economic and protocol-level modeling for Web3 projects.
- **Custom Security Solutions:** Tailored frameworks and advisory for complex decentralized systems and blockchain ecosystems.

Contact Information

 https://mixbytes.io/

 https://github.com/mixbytes/audits_public

 https://x.com/MixBytes

 hello@mixbytes.io