cecuro

Audit Report

February 28, 2026

PROJECT

Yearn Finance

Audit Overview

Project:	Yearn Finance
Repository:	https://github.com/yearn/yearn-vaults-v3
Audit Date:	February 28, 2026
Commit:	104a2b23
Scope:	14 files

0	0	6	42	24
CRITICAL	HIGH	MEDIUM	LOW	INFO

Audit Scope

The following 14 files were included in this security audit:

- contracts/VaultFactory.vy
- contracts/VaultV3.vy
- contracts/interfaces/IDeployer.sol
- contracts/interfaces/IVault.sol
- contracts/interfaces/IVaultFactory.sol
- contracts/interfaces/Roles.sol
- contracts/interfaces/VaultConstants.sol
- lib/tokenized-strategy/src/BaseStrategy.sol
- lib/tokenized-strategy/src/TokenizedStrategy.sol
- lib/tokenized-strategy/src/interfaces/IBaseStrategy.sol
- lib/tokenized-strategy/src/interfaces/IEvents.sol
- lib/tokenized-strategy/src/interfaces/IFactory.sol
- lib/tokenized-strategy/src/interfaces/IStrategy.sol
- lib/tokenized-strategy/src/interfaces/ITokenizedStrategy.sol

Findings

MEDIUM

Off-by-one in unrealised loss share rounding can overcharge users and revert withdrawals/maxWithdraw

Locations:

`contracts/VaultV3.vy:669-694`

`contracts/VaultV3.vy:585-636`

`contracts/VaultV3.vy:784-825`

Description:

`_assess_share_of_unrealised_losses()` attempts to compute the portion of *unrealised* strategy losses that a withdrawing user should realize. The function computes:

- `users_share_of_loss = assets_needed - floor(assets_needed * strategy_assets / strategy_current_debt)`
- and then **adds an extra `+1` when there is a remainder**.

Because `assets_needed` is an integer, the expression `assets_needed - floor(...)` is **already** the *ceiling* of the user's loss share (since `ceil(n - x) = n - floor(x)` for integer `n`). The additional `+1` therefore over-rounds by 1 whenever the ratio is non-integer.

This has two concrete consequences:

1. **Systematic overcharging by 1 unit** of `asset` for most withdrawals from a lossy strategy (whenever the multiplication is not exactly divisible).
2. In small-withdraw edge cases, it can produce `users_share_of_loss > assets_needed` (e.g., `assets_needed = 1`) which then causes **checked subtraction underflows and reverts** in callers, breaking `withdraw`, `redeem`, and even view functions like `maxWithdraw`.

Evidence:

Buggy rounding logic:

```
numerator: uint256 = assets_needed * strategy_assets
users_share_of_loss: uint256 = assets_needed - numerator / strategy_current_debt
# Always round up.
if numerator % strategy_current_debt != 0:
 users_share_of_loss += 1
```

(contracts/VaultV3.vy:688-693)

Revert paths (underflow on subtraction):

- `_max_withdraw()` subtracts the computed loss share:

```
unrealised_loss: uint256 = self._assess_share_of_unrealised_losses(strategy, current_debt,
to_withdraw)
realizable_withdraw: uint256 = to_withdraw - unrealised_loss
```

If `unrealised_loss > to_withdraw` (possible due to the extra `+1`), this underflows and reverts, making `maxWithdraw/maxRedeem` revert. (contracts/VaultV3.vy:599-608)

- `_redeem()` uses the computed value in multiple checked subtractions:

```
if max_withdraw < assets_to_withdraw - unrealised_losses_share:
 ...
assets_to_withdraw -= unrealised_losses_share
requested_assets -= unrealised_losses_share
assets_needed -= unrealised_losses_share
current_total_debt -= unrealised_losses_share
```

(contracts/VaultV3.vy:806-825)

Concrete example:

Let `strategy_current_debt = 100`, `strategy_assets = 99` (1% unrealised loss), `assets_needed = 1`:

- `numerator = 1 * 99 = 99`
- `numerator / debt = 0` (floor)
- `users_share_of_loss = 1 - 0 = 1`
- `numerator % debt != 0` so `users_share_of_loss += 1` ’2

A caller then executes `assets_to_withdraw -= unrealised_losses_share` with `assets_to_withdraw = 1`, causing underflow and revert.

Impact:

- **DoS / unexpected reverts** in `withdraw()`/`redeem()` whenever the withdrawal path needs a small residual amount from a strategy with any unrealised loss (notably 1 unit of `asset`).
- **DoS in view functions**: `maxWithdraw()` (and thus `maxRedeem()`) can revert under the same conditions, breaking ERC-4626 integrations.
- **Incorrect accounting**: even when not reverting, users realize **1 extra unit** of loss per lossy-strategy withdrawal (and the vault reduces `total_debt` accordingly), skewing vault accounting and unfairly penalizing withdrawers.

Recommendation:

Fix the rounding math so it computes the intended `ceil(lossShare)` without over-rounding. Specifically, remove the extra `+1` (or re-derive the formula using a single ceil operation) and ensure the result is always `<= assets_needed`.

MEDIUM

Unreported/unrealised strategy losses are not priced in, enabling bank-run withdrawals (idle drain / strategy cherry-picking) that shift losses to remaining holders

Locations:

`contracts/VaultV3.vy:431-435`

`contracts/VaultV3.vy:536-642`

`contracts/VaultV3.vy:715-909`

`contracts/VaultV3.vy:1827-1870`

Description:

Vault share pricing (`_total_assets = total_idle + total_debt`) uses **book debt** and does not mark strategies to market between `process_report()` calls. If a strategy has suffered losses since its last report, `total_debt` can materially overstate real value.

While `_redeem()` *can* charge a withdrawing user their proportional share of a specific strategy's unrealised losses, it only does so **when the withdraw actually pulls from that strategy**. If the withdrawal can be serviced from `total_idle`, or from a user-supplied queue that excludes the lossy strategy, the user can exit at an inflated price and avoid bearing their share of the loss.

This creates a classic bank-run / adverse-selection dynamic:

- Early withdrawers drain idle liquidity at an overstated PPS.
- Remaining users are forced to withdraw from (or eventually report) lossy strategies and eat the entire loss.

Evidence:

Book-value totalAssets:

```
def _total_assets() -> uint256:
 return self.total_idle + self.total_debt
```

Withdraw path only assesses unrealised losses inside the strategy-withdraw loop, which is skipped entirely when `requested_assets <= total_idle`:

```
current_total_idle: uint256 = self.total_idle
...
if requested_assets > current_total_idle:
 for strategy in _strategies:
 unrealised_losses_share: uint256 = self._assess_share_of_unrealised_losses(...)
 if unrealised_losses_share > 0:
 requested_assets -= unrealised_losses_share
 ...
```

User can supply a custom queue (unless `use_default_queue` is forced), enabling cherry-picking away from lossy strategies:

```
if len(strategies) != 0 and not self.use_default_queue:
 _strategies = strategies
```

Exploit sketch:

Assume Strategy A has a large unrealised loss but has not been reported, and the vault holds enough `total_idle` (or enough debt in other strategies) to cover some withdrawals.

1) Attacker withdraws/redeems an amount `<= total_idle` (or uses a custom queue to withdraw only from healthy strategies). 2) No unrealised-loss haircut is applied. 3) Attacker exits at book PPS, draining liquid assets. 4) Honest/late users later withdraw and are forced to tap Strategy A (or wait for a report) and bear the full loss.

Impact:

- Losses are socialized onto remaining holders; early withdrawers extract value.
- Strong MEV incentive when a strategy loss is suspected but not yet reported.
- Custom queues can magnify the problem by letting users systematically avoid specific lossy strategies.

Recommendation:

Consider pricing shares against mark-to-market strategy values (or incorporate a global unrealised-loss buffer into pricing), and/or ensure withdrawals cannot selectively avoid bearing proportional unrealised losses across the vault's whole strategy set.

MEDIUM

buy_debt() can be abused to siphon unreported strategy profits by purchasing shares at book debt value

Location:

`contracts/VaultV3.vy:1648-1696`

Description:

`buy_debt()` is intended for emergency situations to buy **bad debt** from the vault. However, the function prices the strategy position purely off `strategies[strategy].current_debt` (book value) and does **not** ensure that the strategy is actually underwater (assets < debt).

If a strategy has **unreported gains** (common between reports), then `convertToAssets(balanceOf(vault)) > current_debt`. A `DEBT_PURCHASER` can then call `buy_debt()` to transfer a pro-rata portion of the vault's strategy shares out of the vault while paying only the corresponding fraction of `current_debt` in `asset`.

This effectively allows the purchaser to acquire strategy shares worth more than they paid, siphoning profit away from vault share holders.

Evidence:

```
# contracts/VaultV3.vy
current_debt: uint256 = self.strategies[strategy].current_debt
...
# priced purely by current_debt, ignores actual strategy value
shares: uint256 = IStrategy(strategy).balanceOf(self) * _amount / current_debt
...
self._erc20_safe_transfer_from(self.asset, msg.sender, self, _amount)
...
self._erc20_safe_transfer(strategy, msg.sender, shares)
```

There is no check that the strategy's position is worth `<= current_debt` (i.e., actually "bad debt").

Attack scenario:

1. Strategy has `current_debt = 100` but (unreported) is worth `200` assets.
2. `DEBT_PURCHASER` calls `buy_debt(strategy, 100)`.

3. They pay 100 `asset` and receive 100% of the vault's strategy shares (worth ~200 assets).
4. Vault loses ~100 assets of value; depositor share holders are diluted/robbed.

Impact:
High impact value transfer from vault depositors to the `DEBT_PURCHASER` role holder whenever strategies have unreported profits.

Recommendation:
Enforce that `buy_debt()` can only be used when the strategy is actually impaired (e.g., by checking `IStrategy(strategy).convertToAssets(IStrategy(strategy).balanceOf(self)) <= current_debt`) and/or price the shares based on actual strategy value rather than book debt. If conversion is considered untrustworthy, restrict `buy_debt()` further (e.g., only after a report that confirms loss, or only up to the confirmed loss amount).

MEDIUM

force_revoke_strategy can be abused (revoke/re-add/report loops) to repeatedly 'recreate' profit and farm performance/protocol fees without real gains

Locations:

```
contracts/VaultV3.vy:931-963
contracts/VaultV3.vy:912-930
contracts/VaultV3.vy:1115-1325
contracts/VaultV3.vy:1718-1729
```

Description:

`force_revoke_strategy()` ultimately calls `_revoke_strategy(strategy, True)`, which **writes off the strategy's entire `current_debt` as a loss** by decrementing `total_debt`, then clears the strategy's params (including `current_debt`) back to 0.

Crucially, this revocation does **not** redeem or otherwise dispose of the vault's existing strategy shares. The vault can still hold `IStrategy(strategy).balanceOf(vault)` after revocation.

If the strategy is later re-added, its `current_debt` restarts at 0. On the next `process_report(strategy)`, the vault computes:

- `total_assets = strategy.convertToAssets(strategy.balanceOf(vault))`
- `current_debt = 0`

So the entire position is treated as **new profit**, and fees can be charged/minted.

This creates a dangerous economic primitive: a privileged actor can cycle **force revoke ' re-add ' report** to repeatedly manufacture "gain" from the same unchanged strategy position and **farm accountant/protocol fees** multiple times, even when the vault has not actually made net profit.

The code comment acknowledges this risk ("loss will be credited as profit. Fees will apply."), but there are no guardrails preventing repeated cycles.

Evidence:

Force revoke writes off debt and clears params:

```
# contracts/VaultV3.vy
if self.strategies[strategy].current_debt != 0:
 assert force, "strategy has debt"
 loss: uint256 = self.strategies[strategy].current_debt
 self.total_debt -= loss
 log StrategyReported(strategy, 0, loss, 0, 0, 0, 0)
...
self.strategies[strategy] = StrategyParams({ activation: 0, last_report: 0, current_debt: 0,
max_debt: 0 })
```

Re-add starts with `current_debt: 0`:

```
self.strategies[new_strategy] = StrategyParams({ ..., current_debt: 0, max_debt: 0 })
```

Report treats existing strategy shares as profit relative to `current_debt`:

```
strategy_shares: uint256 = IStrategy(strategy).balanceOf(self)
total_assets = IStrategy(strategy).convertToAssets(strategy_shares)
current_debt = self.strategies[strategy].current_debt  # 0 after re-add
...
if total_assets > current_debt:
 gain = total_assets - current_debt
...
# fees minted in shares
self._issue_shares(total_fees_shares - protocol_fees_shares, accountant)
```

Exploit sketch (privileged fee farming):

Assume the vault holds strategy shares worth X assets and the accountant charges a nonzero performance fee. 1) Manager calls `force_revoke_strategy(strategy)` ' vault books a loss of X (PPS drops), but still holds the strategy shares. 2) Manager calls `add_strategy(strategy)`. 3) Manager calls `process_report(strategy)` ' vault books a gain of ~X and mints fee shares. 4) Repeat steps 1–3: each cycle mints fees again on the same underlying position.

Net effect: depositors can be drained via repeated fee minting even if the strategy's real value has not increased.

Impact:

- Potentially unbounded dilution / value extraction by privileged roles (or compromised governance keys).
- Also harms users in "honest mistake" scenarios: erroneous force revokes followed by recovery will charge fees on merely restoring previously written-off value.

Recommendation:

Add guardrails to prevent revoking-with-debt while strategy shares remain (or track/zero out the vault's strategy share balance on revoke), and/or ensure that "re-covered" assets after a write-off are not treated as fee-bearing profit (e.g., maintain a loss carryforward per strategy).

MEDIUM

VaultV3 _process_report() can revert due to underflow in ending_supply when loss+fees are large (DoS of reporting/accounting)

Locations:

`contracts/VaultV3.vy:1195-1237`

`contracts/VaultV3.vy:1225`

Description:

`VaultV3._process_report()` computes `shares_to_burn` (rounded up) from `loss + total_fees`, then immediately subtracts it (and `_unlocked_shares()`) from `total_supply` to derive `ending_supply` using *checked* arithmetic.

If `shares_to_burn + _unlocked_shares() > total_supply + shares_to_lock`, the expression underflows and the whole report reverts **before** the code reaches the later `min()` cap (`to_burn = min(..., total_locked_shares)`).

This creates a realistic denial-of-service on `process_report()` in "worst day" scenarios (e.g., near-100% strategy loss) when the accountant returns non-zero fees, or when rounding-up pushes `shares_to_burn` above the effective supply.

Vulnerable Code:

```
# contracts/VaultV3.vy
shares_to_burn: uint256 = 0
if loss + total_fees > 0:
 shares_to_burn = self._convert_to_shares(loss + total_fees, Rounding.ROUND_UP)

# ...
ending_supply: uint256 = total_supply + shares_to_lock - shares_to_burn - self._un-
locked_shares()
```

Impact:

- `process_report()` can become uncallable for a strategy in a high-loss/high-fee report, preventing the vault from recording losses, updating debt, unlocking logic, and fee/refund processing.
- Operationally, this can block emergency response paths that depend on reporting to reconcile accounting.

- Because `process_report()` is privileged, recovery may require governance actions (e.g., changing the accountant/fees) and may not be possible promptly during incidents.

Recommendation:

Clamp `shares_to_burn` (and/or the combined burn amount including `_unlocked_shares()`) to what can actually be burned, and compute `ending_supply` with saturation at zero rather than allowing an underflow revert. Consider using mulDiv-style math for conversions to reduce rounding-up pushing the burn over the available supply.

MEDIUM

Solidity interfaces do not match Vyper implementations (wrong return type + event indexing), breaking integrations

Locations:

`contracts/interfaces/IVault.sol:7-52`

`contracts/interfaces/IVaultFactory.sol:7-24`

`contracts/VaultV3.vy:2104-2113`

`contracts/VaultFactory.vy:45-69`

Description:

The provided Solidity interfaces contain ABI mismatches vs the actual Vyper implementations. This can break on-chain integrations (wrong decoding/types) and off-chain indexing (incorrect event topic layout).

1) Wrong return type for `FACTORY()` in `IVault`:

The Vyper vault exposes `FACTORY() -> address`, but the Solidity interface declares `FACTORY() -> uint256`.

Evidence

- Solidity interface:

```
function FACTORY() external view returns (uint256);
```

- Vyper implementation:

```
@view
@external
def FACTORY() -> address:
 return self.factory
```

2) Event `indexed` layout mismatches:

Several events are declared with different `indexed` parameters in Solidity vs Vyper (e.g., `UpdateProtocolFeeRecipient`, `GovernanceTransferred` in the factory; `RoleSet`, `StrategyChanged` in the vault).

Evidence (factory example)

- Vyper:

```
event UpdateProtocolFeeRecipient:
 old_fee_recipient: indexed(address)
 new_fee_recipient: indexed(address)
```

- Solidity interface:

```
event UpdateProtocolFeeRecipient(address oldProtocolFeeRecipient, address newProtocolFeeRe-
cipient);
```

Impact:

- Off-chain consumers using the Solidity ABI may decode event data incorrectly (parameters are in topics vs data).
- On-chain callers compiled against the mismatched interface can mis-handle return types and introduce downstream logic bugs.

Recommendation:

Align Solidity interface function signatures and event `indexed` declarations exactly with the Vyper contracts' ABIs.

LOW

If `profit_max_unlock_time == 0`, syncing idle gains with `process_report(self)` can create `totalAssets > 0` while `totalSupply == 0`, enabling first depositor to steal all pre-funded/donated assets

Locations:

```
contracts/VaultV3.vy:460-472
contracts/VaultV3.vy:1113-1169
contracts/VaultV3.vy:1213-1236
contracts/VaultV3.vy:1252-1263
contracts/VaultFactory.vy:108-144
```

Description:

VaultV3's share conversion has a special-case: when `_total_supply() == 0`, it returns `assets` (PPS = 1). This is standard.

However, VaultV3 also has a mechanism to sync unaccounted underlying into `total_idle` by calling `process_report(strategy=self)`.

When `profit_max_unlock_time == 0`, `_process_report(self)` will:

- update `total_idle` to the vault's actual `ERC20(asset).balanceOf(vault)` (thereby making `totalAssets() > 0`),
- but **will not mint any "locked shares" to the vault** (because profit locking is disabled),
- meaning it is possible to reach a state where:
  - `totalAssets() > 0` (there are real assets recorded),
  - `totalSupply() == 0`.

In that state, the **first depositor** can mint shares at 1:1 (because supply is 0), and immediately redeem them for *all* assets (including any pre-funded/donated balance). This is a first-depositor windfall / theft of pre-existing assets.

This is especially relevant because the factory allows deploying vaults with an arbitrary `profit_max_unlock_time` (including 0).

Evidence:

Share conversion special-case:

```
def _convert_to_shares(assets: uint256, rounding: Rounding) -> uint256:
 total_supply: uint256 = self._total_supply()
 # if total_supply is 0, price_per_share is 1
 if total_supply == 0:
 return assets
```

Syncing vault balance via `process_report(self)`:

```
if strategy == self:
 total_assets = ERC20(_asset).balanceOf(self)
 current_debt = self.total_idle
...
if total_assets > current_debt:
 gain = total_assets - current_debt
```

But when `profit_max_unlock_time == 0`, the profit-locking mint-to-vault path is skipped:

```
if gain + total_refunds > 0 and profit_max_unlock_time != 0:
 shares_to_lock = self._convert_to_shares(...)
# (otherwise shares_to_lock stays 0)
ending_supply = total_supply + shares_to_lock - shares_to_burn - self._unlocked_shares()
# with supply==0 and shares_to_lock==0 => no shares minted
```

Then for vault self-reporting, idle is set to the real balance:

```
else:
 current_debt = current_debt + total_refunds
 self.total_idle = current_debt
```

Factory allows arbitrary `profit_max_unlock_time` input:

```
def deploy_new_vault(..., profit_max_unlock_time: uint256) -> address:
 ...
 IVault(vault_address).initialize(..., profit_max_unlock_time)
```

Impact:

If the vault address is ever pre-funded with the underlying (e.g., accidental transfer, deterministic CREATE2 address pre-funding, airdrop), and a reporting manager syncs it while supply is still 0, the first depositor can steal the entire pre-existing asset balance.

This can result in immediate, total loss of those pre-funded assets.

Recommendation:
Disallow initializing vaults with `profit_max_unlock_time == 0`, or ensure syncing idle gains when `total_supply == 0` mints offset shares to a neutral account (e.g., the vault itself or a burn address) so that the first depositor cannot claim pre-existing assets for free.

LOW

ERC4626 flows can permanently brick when totalAssets==0 or when redemption rounds to 0 (ZERO_ASSETS/ZERO_SHARES guards)

Locations:

```
lib/tokenized-strategy/src/TokenizedStrategy.sol:505-508
lib/tokenized-strategy/src/TokenizedStrategy.sol:529-533
lib/tokenized-strategy/src/TokenizedStrategy.sol:629-635
lib/tokenized-strategy/src/TokenizedStrategy.sol:842-851
```

Description:

Several core ERC-4626 entrypoints hard-revert when the share/asset conversion rounds to zero:

- `deposit()` requires `shares != 0`.
- `mint()` requires `assets != 0`.
- `redeem()` requires the previewed `assets != 0`.

Combined with `_convertToShares()` returning `0` whenever `totalAssets == 0` and `totalSupply != 0`, the strategy can become **non-functional** in realistic scenarios:

- After a catastrophic loss or an erroneous `harvestAndReport()` that returns `0` while shares remain outstanding, **all** `deposit/mint/withdraw/redeem` calls can become permanently impossible (until an out-of-band donation + successful report changes `totalAssets`).
- Even when `totalAssets > 0`, small share positions can become **unredeemable** if `convertToAssets(shares)` rounds down to `0` (dust shares), because `redeem()` reverts instead of burning shares for `0` assets.

This creates stuck states and breaks ERC-4626 expectations that small/zero-asset redemptions are possible (burning shares should still be allowed).

Evidence:

Zero conversions are treated as errors:

```
// deposit()
require((shares = _convertToShares(S, assets, Math.Rounding.Down)) != 0, "ZERO_SHARES");

// mint()
require((assets = _convertToAssets(S, shares, Math.Rounding.Up)) != 0, "ZERO_ASSETS");

// redeem()
require((assets = _convertToAssets(S, shares, Math.Rounding.Down)) != 0, "ZERO_ASSETS");
```

`_convertToShares` returns 0 when `totalAssets == 0` but supply is non-zero:

```
uint256 totalSupply_ = _totalSupply(S);
if (totalSupply_ == 0) return assets;

uint256 totalAssets_ = _totalAssets(S);
if (totalAssets_ == 0) return 0;

return assets.mulDiv(totalSupply_, totalAssets_, _rounding);
```

Impact:

- **Permanent/long-lived DoS of core user flows**: If a report sets `totalAssets` to `0` while there is still a non-zero effective supply, users cannot redeem or withdraw at all, even if assets later appear (until a keeper successfully reports a non-zero `harvestAndReport()` result).
- **Dust shares can be permanently stuck**: Users holding small amounts of shares that convert to `< 1` unit of asset can never exit/burn, which can leave residual supply forever and complicate shutdown/finalization.

Recommendation:

- Avoid treating a `0` conversion as an error in `redeem()` (and consider similar handling in `mint/deposit`), at least allowing burning shares for `0` assets.
- Consider explicit handling for the `totalAssets == 0 && totalSupply != 0` state (e.g., allowing deposits to restart/reset pricing, or allowing redemptions for 0 assets to clear supply).

LOW

Emergency shutdown does not disable withdraw-limit module; withdrawals can remain blocked (funds stuck)

Locations:

`contracts/VaultV3.vy:743-747`

`contracts/VaultV3.vy:1764-1788`

Description:

The contract's docstring/storage comment says that when `shutdown` is set, "only withdrawals will be available" (i.e., shutdown is an emergency mode to stop deposits while still allowing exits).

However, `shutdown_vault()` does **not** clear `withdraw_limit_module`, and `_redeem()` enforces the withdraw-limit module unconditionally.

If a withdraw-limit module is configured that returns 0 (pause), misbehaves, or reverts, then **all withdrawals can remain blocked even after shutdown**. Moreover, `shutdown_vault()` only grants the caller `DEBT_MANAGER`, not `WITHDRAW_LIMIT_MANAGER`, so the emergency manager may be unable to remove/replace the module to restore withdrawals.

This violates the "every inlet needs an outlet" principle: emergency shutdown may not restore the exit path.

Evidence:

Withdraw path always enforces withdraw limit module:

```
withdraw_limit_module: address = self.withdraw_limit_module
if withdraw_limit_module != empty(address):
 assert assets <= IWithdrawLimitModule(withdraw_limit_module).available_withdraw_limit(-
owner, max_loss, strategies), "exceed withdraw limit"
```

(contracts/VaultV3.vy:743-747)

Shutdown clears deposit limits but not withdraw limit module:

```
# Set deposit limit to 0.
if self.deposit_limit_module != empty(address):
 self.deposit_limit_module = empty(address)
...
self.deposit_limit = 0
...
new_roles: Roles = self.roles[msg.sender] | Roles.DEBT_MANAGER
self.roles[msg.sender] = new_roles
```

(contracts/VaultV3.vy:1775-1786)

Impact:

In an emergency, governance/management may shut down the vault expecting users to be able to withdraw, but withdrawals can still be fully DoSed by an existing withdraw-limit module. This can lock user funds for an unbounded time, depending on module behavior and governance ability to intervene.

Recommendation:

Consider disabling/bypassing the withdraw-limit module when `shutdown == True` (or explicitly clearing it during `shutdown_vault()` and/or granting emergency manager authority to remove it).

LOW

Emergency shutdown may not enable unwinding funds: `update_debt` forbids debt reduction when strategy has unrealised losses, and shutdown does not grant reporting permissions

Locations:

```
contracts/VaultV3.vy:981-1046
contracts/VaultV3.vy:1015-1018
contracts/VaultV3.vy:1764-1788
contracts/VaultV3.vy:1636-1646
```

Description:

In an emergency shutdown, the intended operational response is typically to unwind strategy positions back to idle via `update_debt()` so user withdrawals can be serviced cheaply.

However, `_update_debt()` contains a hard gate on **debt decreases**:

- It computes `unrealised_losses_share` and then asserts it is zero, reverting otherwise.
- This means if a strategy is underwater (unrealised losses exist), governance cannot reduce its debt via `update_debt()` at all until a `process_report()` is successfully executed.

At the same time, `shutdown_vault()` only adds the `DEBT_MANAGER` role to the caller, **not** `REPORTING_MANAGER`. If the emergency caller does not already have reporting privileges (or if reporting is blocked for other reasons), shutdown can leave the system in a state where funds cannot be unwound by the emergency operator.

This is an execution-flow / stuck-state risk: a realistic emergency condition (strategy losses) can prevent the very management action needed to recover liquidity.

Evidence:

Debt-reduction path forbids unrealised losses:

```
unrealised_losses_share: uint256 = self._assess_share_of_unrealised_losses(strategy, cur-
rent_debt, assets_to_withdraw)
assert unrealised_losses_share == 0, "strategy has unrealised losses"
```

(contracts/VaultV3.vy:1015-1018)

Shutdown grants only DEBT_MANAGER:

```
new_roles: Roles = self.roles[msg.sender] | Roles.DEBT_MANAGER
self.roles[msg.sender] = new_roles
```

(contracts/VaultV3.vy:1784-1786)

Reporting requires `REPORTING_MANAGER`:

```
self._enforce_role(msg.sender, Roles.REPORTING_MANAGER)
```

(contracts/VaultV3.vy:1644-1645)

Impact:

- In a lossy strategy scenario, the emergency operator may be unable to unwind positions after shutdown.
- Liquidity recovery can become dependent on a separate role-holder being available to report (or on users withdrawing and realizing losses themselves), increasing the chance of prolonged inability to service withdrawals.

Recommendation:

Provide an emergency-safe unwind path that does not require unrealised losses to be zero (or ensure shutdown grants/reporting capability or an alternative mechanism to realize losses so debt can be reduced).

LOW

Forced default withdrawal queue can lock withdrawals even when funds exist in other strategies

Locations:

`contracts/VaultV3.vy:762-892`

`contracts/VaultV3.vy:577-640`

`contracts/VaultV3.vy:221-224`

Description:

When `use_default_queue` is enabled, user-supplied withdrawal queues are ignored in both `_redeem()` and `_max_withdraw()`. Withdrawals will only attempt to free funds from `default_queue` (max length `MAX_QUEUE = 10`).

If the vault has debt in strategies that are **not present** in `default_queue` (e.g., added with `add_to_queue=False`, queue filled to MAX_QUEUE, or queue later changed), a withdrawal that requires freeing strategy funds can revert with `"insufficient assets in vault"` even though the vault still holds recoverable value in other active strategies.

This is an execution-flow / stuck-state risk: deposits may be liquid in aggregate, but withdrawals can be temporarily or indefinitely unserviceable unless a privileged role updates `default_queue`, disables `use_default_queue`, or manually pulls funds to idle via `update_debt()`.

Evidence:

`_redeem()` ignores the caller-provided `strategies` when `use_default_queue` is true:

```
# Cache the default queue.
_strategies: DynArray[address, MAX_QUEUE] = self.default_queue

# If a custom queue was passed, and we don't force the default queue.
if len(strategies) != 0 and not self.use_default_queue:
 _strategies = strategies

for strategy in _strategies:
 ...

assert current_total_idle >= requested_assets, "insufficient assets in vault"
```

(contracts/VaultV3.vy:764-892)

`_max_withdraw()` has the same queue-selection logic (contracts/VaultV3.vy:577-584) and will also simulate only over the selected queue.

Impact:

- **High impact DoS**: users can be unable to withdraw expected amounts during normal operation, despite the vault holding assets in strategies.
- In the worst case (e.g., governance/role-manager unavailable, or misconfiguration persists), withdrawals can be effectively locked for long periods.

Recommendation:

Add guardrails to prevent enabling `use_default_queue` unless the queue is known to cover all funded strategies (or provide an on-chain escape hatch that can free funds across strategies without relying on `default_queue`). Also consider enforcing invariants in debt/strategy management that prevent allocating debt to strategies that are not withdraw-reachable under current configuration.

LOW

Overcharging performance/protocol fees when profitMaxUnlockTime is set to 0 (no profit locking)

Locations:

`lib/tokenized-strategy/src/TokenizedStrategy.sol:1081-1256`

`lib/tokenized-strategy/src/TokenizedStrategy.sol:1583-1617`

Description:

`report()` calculates fee shares (`totalFeeShares`) from `sharesToLock`, where `sharesToLock` is computed as the shares-equivalent of the *profit* at the **pre-report PPS**:

- `sharesToLock = _convertToShares(profit, Down)`
- `totalFeeShares = (sharesToLock * performanceFee) / MAX_BPS`
- fee shares are minted to recipients.

This fee-share math implicitly assumes profit locking will keep PPS constant at report time (by minting/burning shares to `address(this)`), so that "shares at pre-report PPS" represent the intended % of profit.

However, when `profitMaxUnlockTime == 0`, the profit-locking adjustment block is skipped entirely:

- no shares are minted/burned to/from `address(this)` for `sharesToLock`
- PPS immediately increases because `totalAssets` is updated to `newTotalAssets` while supply only increases by fee shares

As a result, the **asset value** of the minted fee shares becomes **strictly greater than `performanceFee%` of profit** (and therefore the protocol fee, defined as a % of performance fees, is also overcharged in asset terms).

This is not just rounding: it grows with `profit / oldTotalAssets` and can be multiple-x for large profits.

Evidence (code):

Fee shares are derived from profit shares at pre-report PPS:

```
sharesToLock = _convertToShares(S, profit, Math.Rounding.Down);
...
totalFees = (profit * fee) / MAX_BPS;
totalFeeShares = (sharesToLock * fee) / MAX_BPS;
...
_mint(S, protocolFeesRecipient, protocolFeeShares);
_mint(S, S.performanceFeeRecipient, totalFeeShares - protocolFeeShares);
```

(see `report()` at `TokenizedStrategy.sol:1115-1161`)

Profit-locking (which would keep PPS constant) is conditional and **skipped when `_profitMaxUnlockTime == 0`**:

```
if (_profitMaxUnlockTime != 0) {
 sharesToLock -= totalFeeShares;
 ...
 _mint/_burn(address(this), ...);
}
```

(see `TokenizedStrategy.sol:1164-1182`)

And `profitMaxUnlockTime` can explicitly be set to 0 by management:

```
if (_profitMaxUnlockTime == 0) {
 uint256 shares = S.balances[address(this)];
 if (shares != 0) _burn(S, address(this), shares);
 S.profitUnlockingRate = 0;
 S.fullProfitUnlockDate = 0;
}
S.profitMaxUnlockTime = uint32(_profitMaxUnlockTime);
```

(see `TokenizedStrategy.sol:1596-1616`)

Why this overcharges (math):

Let:

- `A0` = old total assets (`oldTotalAssets`)
- `S0` = effective supply (`_totalSupply()` before minting)
- `P` = profit
- `f` = performanceFee / 10_000

Ignoring rounding, the code mints:

- `feeShares = f * (P * S0 / A0)`

If `profitMaxUnlockTime == 0`, no offsetting “profit lock shares” are minted to keep PPS stable, so after report:

- assets `A1 = A0 + P`

- supply `S1 = S0 + feeShares`

The asset value of minted fee shares is: `feeShares * (A1 / S1) = f*P * (A0 + P) / (A0 + f*P)` which is **> f*P** for any `P>0` and `0<f<1`.

Impact:

If management sets `profitMaxUnlockTime` to 0 (a supported configuration), a `report()` can mint fee shares worth substantially more than the configured performance fee percentage of profit, transferring value from share holders to fee recipients.

This is especially severe when `profit` is large relative to existing assets (e.g., harvest of large rewards or a large external airdrop recognized on report).

Recommendation:

When `profitMaxUnlockTime == 0`, compute fee shares based on a PPS that reflects the post-profit state (or otherwise adjust minting so that fee recipients receive exactly `performanceFee%` of profit in asset terms), rather than reusing the “pre-profit PPS + profit-locking” fee-share formula.

LOW

Stale totalAssets enables withdrawal bank-run and unfair loss distribution (early withdrawers exit at inflated PPS)

Locations:

`lib/tokenized-strategy/src/TokenizedStrategy.sol:650-667`

`lib/tokenized-strategy/src/TokenizedStrategy.sol:822-867`

`lib/tokenized-strategy/src/TokenizedStrategy.sol:894-918`

`lib/tokenized-strategy/src/TokenizedStrategy.sol:989-1050`

`lib/tokenized-strategy/src/TokenizedStrategy.sol:1081-1256`

Description:

`TokenizedStrategy` intentionally **manually tracks** `totalAssets` and only synchronizes it to reality on `report()` (keeper/management). Between reports, `convertToAssets`/`maxWithdraw`/`withdraw` operate on this **stale accounting value**.

If the strategy suffers an **unreported loss** (e.g., underlying position devalues) but the strategy still has enough *liquid* `asset` on hand to satisfy some withdrawals, **early withdrawers can withdraw at the stale (overstated) price**, leaving later withdrawers to realize a disproportionate share of the loss once liquidity is exhausted.

This creates a classic **bank-run / MEV race**: anyone monitoring the strategy's real backing (offchain or via external protocols) can rush to withdraw first.

Evidence:

`totalAssets()` returns a stored value, not live holdings:

```
function _totalAssets(StrategyData storage S) internal view returns (uint256) {
 return S.totalAssets;
}
```

Conversion and limits depend on this stored value:

```
function _convertToAssets(...) internal view returns (uint256) {
 uint256 supply = _totalSupply(S);
 return supply == 0 ? shares : shares.mulDiv(_totalAssets(S), supply, _rounding);
}

function _maxWithdraw(...) internal view returns (uint256 maxWithdraw_) {
 maxWithdraw_ = IBaseStrategy(address(this)).availableWithdrawLimit(owner);
 ...
 maxWithdraw_ = _convertToAssets(S, _balanceOf(S, owner), Math.Rounding.Down);
}
```

`totalAssets` only becomes “real” on `report()`:

```
uint256 newTotalAssets = IBaseStrategy(address(this)).harvestAndReport();
...
S.totalAssets = newTotalAssets;
```

Withdrawals are allowed against stale `totalAssets` until an on-chain shortfall manifests:

```
require(assets <= _maxWithdraw(S, owner), ...);
...
uint256 idle = _asset.balanceOf(address(this));
if (idle < assets) { IBaseStrategy(address(this)).freeFunds(assets - idle); ... }
```

Exploit sketch:

1. Strategy suffers an unreported loss in its yield source; `S.totalAssets` remains high.
2. Attacker (or MEV bot) withdraws/redeems while there is still enough liquid `asset` to satisfy their request at the stale PPS.
3. Subsequent users hit `idle < assets`, and the shortfall is recorded as `loss` and borne by them.

Impact:

- **Value extraction / unfair loss allocation**: early withdrawers receive more assets than their fair pro-rata share relative to real strategy backing.
- **MEV race** around adverse events (depeg, liquidation, exploit) can materially harm passive users.
- Can escalate to a **run** where rational users all attempt to exit pre-report.

Recommendation:

Mitigate stale-accounting bank-runs by ensuring withdrawals are priced against more up-to-date reality and/or by limiting withdrawals when accounting is stale (e.g., require fresh reports, cap withdrawals to idle, or socialize losses rather than first-come-first-served).

LOW

report(): profit unlocking recalculation can divide by zero and permanently DoS reporting when strategy holds shares outside the locked-profit flow

Locations:

`lib/tokenized-strategy/src/TokenizedStrategy.sol:1208-1238`

`lib/tokenized-strategy/src/TokenizedStrategy.sol:1132-1152`

`lib/tokenized-strategy/src/TokenizedStrategy.sol:1596-1616`

Description:

`report()` recalculates the profit unlocking schedule using `totalLockedShares = S.balances[address(this)]` and computes:

```
uint256 newProfitLockingPeriod = (previouslyLockedTime + sharesToLock * _profitMaxUnlockTime)
/ totalLockedShares;
S.profitUnlockingRate = (totalLockedShares * MAX_BPS_EXTENDED) / newProfitLockingPeriod;
```

This assumes an invariant that **if `S.balances[address(this)] > 0` then `newProfitLockingPeriod > 0`**. That invariant can be broken whenever the strategy ends up holding shares that are *not* part of the intended “locked profit” mechanism.

A concrete reachable case exists because `report()` mints protocol fee shares to the recipient returned by the external `FACTORY.protocol_fee_config()` without validating it (it may be set to `address(this)`):

```
(uint16 protocolFeeBps, address protocolFeesRecipient) = IFactory(FACTORY).protocol_fee_con-
fig();
...
_mint(S, protocolFeesRecipient, protocolFeeShares);
```

If protocol fees are minted to `address(this)`, those shares are counted in `totalLockedShares` even though they are not included in `sharesToLock` (which represents newly locked profit after fees). If `profitMaxUnlockTime` is configured to a very small value (notably `0` or `1`), integer division can yield `newProfitLockingPeriod == 0`, causing a division-by-zero revert when computing `profitUnlockingRate`.

Example DoS configurations:

- `setProfitMaxUnlockTime(0)` (explicitly allowed).
- Factory returns `protocolFeesRecipient == address(this)` and `protocolFeeBps != 0`.
- Any profitable `report()` call then attempts to mint protocol fee shares to `address(this)`, making `totalLockedShares > 0` while `_profitMaxUnlockTime == 0`, which forces `newProfitLockingPeriod == 0` and reverts.

This revert happens *after* the external `harvestAndReport()` call, so keepers can be griefed into repeatedly spending gas on the strategy harvest only to revert at the end.

Impact:

- **Permanent DoS of `report()`** under certain (mis)configurations.
- `totalAssets` will never be updated again, leaving the ERC-4626 share/asset conversions stale.
- Keepers can be griefed: strategy harvest work may be performed but the transaction reverts at the unlocking-rate computation.

Recommendation:

- Enforce invariants before the division:
  - Require `totalLockedShares == 0` when `profitMaxUnlockTime == 0`.
  - Guard `newProfitLockingPeriod != 0` (and handle the edge case explicitly).
- Validate factory outputs (e.g., reject `protocolFeesRecipient == address(this)` and `protocolFeesRecipient == address(0)`), or exclude non-locked-profit shares from `totalLockedShares` accounting.

LOW

withdraw/redeem allow receiver==vault, breaking total_idle accounting and potentially DoSing withdrawals

Locations:

`contracts/VaultV3.vy:514-517`

`contracts/VaultV3.vy:715-908`

Description:

`_redeem()` validates `receiver != empty(address)` but does **not** forbid `receiver == self`.

Because the vault maintains its own `total_idle` accounting (instead of relying on `ERC20(asset).balanceOf(self)`), a withdrawal to `receiver == self` can leave the vault's actual ERC-20 balance unchanged while still decrementing `total_idle` as if assets left the contract.

This creates a persistent mismatch (`ERC20(asset).balanceOf(vault) > total_idle`) that can:

- force future withdrawals to unnecessarily try freeing funds from strategies (even though the vault actually has enough idle tokens), and
- revert with `"insufficient assets in vault"` if strategies are illiquid / maxRedeem-limited,

until a privileged `process_report(self)` call re-syncs `total_idle` to the actual balance.

The codebase already treats `self` as an invalid receiver in other contexts (e.g., `_max_deposit()` returns 0 for `receiver == self`), suggesting this omission is unintended.

Evidence:

Receiver validation in `_redeem()`:

```
assert receiver != empty(address), "ZERO ADDRESS"
...
self.total_idle = current_total_idle - requested_assets
self._erc20_safe_transfer(_asset, receiver, requested_assets)
```

Contrast: deposits disallow `receiver == self`:

```
def _max_deposit(receiver: address) -> uint256:
 if receiver in [empty(address), self]:
 return 0
```

Impact:

A user can call `withdraw()`/`redeem()` with `receiver == vault` to create unaccounted idle assets. If the vault normally relies on `total_idle` as a withdrawal buffer, this can eliminate the buffer from accounting and cause **unexpected withdrawal failures** (DoS) when strategies cannot redeem quickly.

Even when it doesn't DoS, it undermines core accounting invariants (`total_idle` no longer reflects the vault's real idle balance), affecting `totalAssets()`/conversion previews and operational decisions.

Recommendation:

Reject `receiver == self` in `_redeem()` (or otherwise special-case self-transfers so that `total_idle` is only decremented when assets actually leave the vault).

LOW

Admin setters allow setting critical module addresses to non-contracts (EOA), which can brick deposits/withdrawals/reports

Locations:

`contracts/VaultV3.vy:1350-1359`

`contracts/VaultV3.vy:1435-1462`

`contracts/VaultV3.vy:1463-1475`

Description:

Several privileged setters accept arbitrary addresses for core modules but do not validate that the address is either zero (disable) or a deployed contract implementing the expected interface:

- `set_accountant(new_accountant)`
- `set_deposit_limit_module(deposit_limit_module, ...)`
- `set_withdraw_limit_module(withdraw_limit_module)`

Example:

```
def set_withdraw_limit_module(withdraw_limit_module: address):
 self._enforce_role(msg.sender, Roles.WITHDRAW_LIMIT_MANAGER)
 self.withdraw_limit_module = withdraw_limit_module
```

These modules are later called via typed interface calls without `default_return_value`. In the EVM, calling an EOA returns `success` with **empty returndata**, and Vyper ABI decoding of the expected return value will revert. As a result:

- If `withdraw_limit_module` is an EOA (or a contract with incompatible ABI), **all `withdraw()` / `redeem()`** will revert at:

```
IWithdrawLimitModule(withdraw_limit_module).available_withdraw_limit(...)
```

- If `deposit_limit_module` is an EOA / incompatible, **all deposits/mints** can revert when `_max_deposit()` calls `available_deposit_limit()`.
- If `accountant` is an EOA / incompatible, **all `process_report()`** calls revert when calling `IAccountant(accountant).report(...)`.

Impact:

A privileged but potentially operationally separated role (or an honest misconfiguration) can cause a **hard DoS of core user flows** (deposits/withdrawals) or critical maintenance (reporting) until governance fixes the address.

If governance/role management is unavailable (e.g., key loss, or other governance-bricking scenarios), this can become **permanent**.

Recommendation:

In these setters, enforce:

- `addr == empty(address)` is allowed (disables the module), otherwise
- `extcodesize(addr) > 0` and optionally interface/behavior sanity checks.

This reduces the chance that an accidental EOA or wrong contract address bricks the vault.

LOW

Default queue allows duplicate strategies, wasting bounded queue slots and increasing per-withdraw computation/DoS risk

Locations:

`contracts/VaultV3.vy:1361-1376`

`contracts/VaultV3.vy:784-889`

`contracts/VaultV3.vy:585-636`

Description:

`set_default_queue()` validates that each entry is active but explicitly does **not** check for duplicates. If duplicates are present:

- Withdrawals and max-withdraw simulations may perform redundant per-strategy external calls, increasing execution cost.
- Duplicate entries waste scarce `MAX_QUEUE` slots, reducing the number of unique strategies that can be tapped in a single withdrawal. This can cause withdrawals to revert for large requests even when sufficient liquidity exists across strategies, especially if `use_default_queue` is enabled.

Evidence:

```
@dev ... will not check that the same strategy is not added twice.
...
for strategy in new_default_queue:
 assert self.strategies[strategy].activation != 0, "!inactive"
self.default_queue = new_default_queue
```

(contracts/VaultV3.vy:1364-1376)

Withdraw/maxWithdraw iterate over the queue and perform multiple external calls per entry (contracts/VaultV3.vy:585-636, 784-889).

Impact:

- Increased gas cost for withdrawals and `maxWithdraw`-style simulations.
- Higher chance of withdrawal DoS for large withdrawals (needs more unique strategies than available queue slots due to duplicates).

Recommendation:
Enforce uniqueness in `default_queue` (and potentially in user-provided queues) or at least provide tooling/guards to prevent accidental duplicate configuration.

LOW

Deposits/withdrawals/reports are tightly coupled to strategy callbacks with no batching, risking gas-based DoS for complex/scale-dependent strategies

Locations:

```
lib/tokenized-strategy/src/TokenizedStrategy.sol:954-970
lib/tokenized-strategy/src/TokenizedStrategy.sol:989-1045
lib/tokenized-strategy/src/TokenizedStrategy.sol:1081-1097
lib/tokenized-strategy/src/TokenizedStrategy.sol:1314-1319
lib/tokenized-strategy/src/TokenizedStrategy.sol:1356-1364
```

Description:

Key ERC-4626 state transitions synchronously invoke strategy-specific callback hooks via external self-calls:

- `deposit/mint` → `IBaseStrategy(address(this)).deployFunds(...)`
- `withdraw/redeem` → `IBaseStrategy(address(this)).freeFunds(...)`
- `report` → `IBaseStrategy(address(this)).harvestAndReport()`
- `tend` → `IBaseStrategy(address(this)).tendThis(...)`
- `emergencyWithdraw` → `IBaseStrategy(address(this)).shutdownWithdraw(...)`

These callbacks are implemented by the strategist (via `BaseStrategy`'s `_deployFunds/_freeFunds/_harvestAndReport/_tend/_emergencyWithdraw`). `TokenizedStrategy` provides **no built-in mechanism** to:

- bound their computational complexity,
- split work across transactions (batching), or
- recover if the callback becomes too expensive to run within block gas limits.

As a result, a strategy whose internal accounting unwinds positions, harvests rewards, or iterates over an ever-growing set of positions/tokens can reach a point where these core flows revert due to out-of-gas.

Impact:

- **DoS risk for core user flows**:
  - `withdraw/redeem` can become unexecutable if `_freeFunds` can't complete within gas limits for requested amounts, effectively trapping funds (especially for large share holders).
  - `report` can become uncallable if `_harvestAndReport` becomes too expensive, freezing `totalAssets` updates and profit/loss accounting.
- The protocol has only limited recovery via `shutdownStrategy` + `emergencyWithdraw`, which itself is also synchronous and can suffer the same computational bounds.

Recommendation:

Consider making the interface and recommended strategy patterns explicitly support bounded computation, e.g.:

- design strategy callbacks to be incremental (process N items per call),
- add strategy-side pagination parameters or internal state machines,
- add alternative "partial free" flows that can be executed over multiple txs,
- ensure docs/spec require strategists to keep callback complexity O(1) with respect to user count / position count.

LOW

EIP-2612 permit DOMAIN_SEPARATOR hard-codes name 'Yearn Vault' (mismatched with to-ken/vault name)

Locations:

`contracts/VaultV3.vy:357-392`

`contracts/VaultV3.vy:2178-2198`

`lib/tokenized-strategy/src/TokenizedStrategy.sol:433-474`

`lib/tokenized-strategy/src/TokenizedStrategy.sol:2018-2031`

Description:

Both the Vyper Vault (`VaultV3.vy`) and the Solidity TokenizedStrategy implementation hardcode the EIP-712 domain `name` as `"Yearn Vault"` when computing `DOMAIN_SEPARATOR`, while simultaneously exposing a **user-configurable name** via `name`/`setName`.

This creates a practical incompatibility with common EIP-2612 / EIP-712 signing flows:

- Wallets/frontends typically build the EIP-712 domain using the on-chain token `name()`.
- Here, the on-chain `DOMAIN_SEPARATOR` uses a constant name hash, so signatures created using the visible token name will not verify.

Result: `permit()` will frequently revert with an invalid signature even for correct signers, effectively breaking gasless approvals for many integrations.

Evidence:

VaultV3 hardcodes domain name:

```
# contracts/VaultV3.vy
return keccak256(
 concat(
 DOMAIN_TYPE_HASH,
 keccak256(convert("Yearn Vault", Bytes[11])),
 keccak256(convert(API_VERSION, Bytes[28])),
 convert(chain.id, bytes32),
 convert(self, bytes32)
 )
)
```

(contracts/VaultV3.vy:2180-2189)

`permit()` uses this separator:

```
digest: bytes32 = keccak256(concat(b'\x19\x01', self.domain_separator(), ...))
```

(contracts/VaultV3.vy:369-384)

Vault name is configurable:

- `initialize(... name, ...)` sets `self.name`
- `setName()` allows role_manager to change it

(contracts/VaultV3.vy:286-325, 1329-1338)

TokenizedStrategy hardcodes domain name:

```
// lib/tokenized-strategy/src/TokenizedStrategy.sol
keccak256(
 abi.encode(
 keccak256("EIP712Domain(string name,string version,uint256 chainId,address verifyingCon-
tract)"),
 keccak256("Yearn Vault"),
 keccak256(bytes(API_VERSION)),
 block.chainid,
 address(this)
 )
)
```

(lib/tokenized-strategy/src/TokenizedStrategy.sol:2018-2030)

But strategy `name()` comes from storage set in `initialize(_name)`:

- `S.name = _name` in `initialize`
- `function name()` returns `S.name`

(lib/tokenized-strategy/src/TokenizedStrategy.sol:433-474, 1636-1638)

Impact:

- `permit()` signatures produced using the token's visible `name()` (the default behavior for many wallets/SDKs) will fail verification.
- Breaks gasless approvals and can break integrations that assume standard ERC20Permit semantics.

Recommendation:

Use `keccak256(bytes(name()))` / `keccak256(bytes(S.name))` as the domain name hash (OpenZeppelin ERC20Permit pattern), or clearly document and expose the constant domain name as the canonical name used for signing (including ensuring UIs use it).

LOW

EIP-2612 permit accepts malleable signatures (missing low-s / v normalization)

Locations:

`contracts/VaultV3.vy:357-392`

`lib/tokenized-strategy/src/TokenizedStrategy.sol:1965-2009`

Description:

Both `VaultV3` and `TokenizedStrategy` implement `permit` using raw `ecrecover` without enforcing canonical ECDSA signature constraints.

Neither implementation:

- enforces `v` to be 27/28 (or normalizes 0/1), nor
- enforces the EIP-2 low-`s` requirement (`s <= secp256k1n/2`).

As a result, malleable signatures (high-`s` variants and alternate `v`) are accepted.

Evidence:

VaultV3:

`contracts/VaultV3.vy`:

```
assert ecrecover(digest, v, r, s) == owner, "invalid signature"
self.allowance[owner][spender] = amount
self.nonces[owner] = nonce + 1
```

No `v`/`s` canonicality checks.

TokenizedStrategy:

`lib/tokenized-strategy/src/TokenizedStrategy.sol`:

```
address recoveredAddress = ecrecover(digest, v, r, s);
require(recoveredAddress != address(0) && recoveredAddress == owner, "ERC20: INVALID_SIGNER");
```

No low-`s` / `v` checks.

Impact:

- Enables alternate signatures for the same signed message, which can:
  - complicate off-chain signature tracking and invalidate assumptions that a signature is unique,
  - enable griefing/front-running scenarios where a relayer's signature is "stolen" using a malleated variant (nonce is still consumed, but attribution/monitoring can break).

While nonce-based replay protection still prevents reusing the same permit twice, malleability remains a standards-compliance and integration risk.

Recommendation:

Adopt OpenZeppelin `ECDSA.recover` (or equivalent) with:

- strict `v` normalization,
- low-`s` enforcement,
- explicit `recovered != address(0)` check.

In Vyper, explicitly validate `v` and `s` against secp256k1 curve order half.

LOW

ERC-4626 redeem() defaults to unlimited loss (maxLoss=100%), which is not visible in ITokenizedStrategy and can surprise integrators

Locations:

`lib/tokenized-strategy/src/interfaces/ITokenizedStrategy.sol:8-83`

`lib/tokenized-strategy/src/TokenizedStrategy.sol:538-603`

`lib/tokenized-strategy/src/TokenizedStrategy.sol:989-1050`

Description:

`ITokenizedStrategy` advertises ERC-4626 compatibility by inheriting `IERC4626`, but the concrete `TokenizedStrategy` implementation applies **different default loss policies** between the standard ERC-4626 entrypoints:

- `withdraw(assets, receiver, owner)` forwards to the non-standard overload with `maxLoss = 0` (no loss allowed ' revert on any shortfall).
- `redeem(shares, receiver, owner)` forwards to the non-standard overload with `maxLoss = MAX_BPS` (100% loss allowed ' can succeed even if most/all assets cannot be freed).

This asymmetric default is not communicated in the `ITokenizedStrategy` interface. Any integrator that calls the standard ERC-4626 `redeem()` expecting similar "no-loss unless specified" behavior (like `withdraw()` provides) can unintentionally allow large losses.

Evidence:

Defaulting behavior:

```
// withdraw defaults to maxLoss = 0
function withdraw(uint256 assets, address receiver, address owner) external returns (uint256
shares) {
 return withdraw(assets, receiver, owner, 0);
}

// redeem defaults to maxLoss = MAX_BPS (10_000)
function redeem(uint256 shares, address receiver, address owner) external returns (uint256) {
 return redeem(shares, receiver, owner, MAX_BPS);
}
```

Loss acceptance is enforced in `_withdraw()`:

```
if (idle < assets) {
 IBaseStrategy(address(this)).freeFunds(assets - idle);
 idle = _asset.balanceOf(address(this));
 if (idle < assets) {
 loss = assets - idle;
 if (maxLoss < MAX_BPS) {
 require(loss <= (assets * maxLoss) / MAX_BPS, "too much loss");
 }
 assets = idle;
 }
}
```

When `maxLoss == MAX_BPS`, the guard is skipped, so redemption can proceed with arbitrarily large realized loss.

Impact:

- **User/integrator surprise**: calling the standard ERC-4626 `redeem()` can realize up to 100% loss without reverting (subject to available liquidity), while `withdraw()` reverts on any loss.
- **Incorrect safety assumptions**: contracts or frontends that only expose/implement ERC-4626's standard `redeem()` path may unintentionally provide a "no slippage protection" exit.

Recommendation:

Make this default-loss asymmetry explicit in the `ITokenizedStrategy` interface documentation (and ideally in public docs), and encourage integrators to use the 4-arg overloads when they need explicit loss bounds.

LOW

ERC20 share transfers forbid sending to vault itself or address(0) (nonstandard behavior)

Location:

`contracts/VaultV3.vy:1882-1904`

Description:

`transfer` and `transferFrom` revert if `receiver` is the vault itself or the zero address:

```
assert receiver not in [self, empty(address)]
```

This deviates from typical ERC-20 behavior where `transfer(0x0, amount)` is often treated as a burn pattern (even if not formally required), and transfers to `address(this)` are usually allowed.

Because the contract presents itself as ERC20/ERC4626 compatible, this semantic difference can break integrations that assume these transfers are valid.

Evidence:

```
@external
def transfer(receiver: address, amount: uint256) -> bool:
 assert receiver not in [self, empty(address)]
 self._transfer(msg.sender, receiver, amount)

@external
def transferFrom(sender: address, receiver: address, amount: uint256) -> bool:
 assert receiver not in [self, empty(address)]
 return self._transfer_from(sender, receiver, amount)
```

(contracts/VaultV3.vy:1882-1904)

Impact:

- Some generic ERC-20 tooling and DeFi integrations may revert unexpectedly when attempting to burn shares or escrow them in the token contract address.
- Potential UX breakage for wallets/dapps that allow “send to 0x0” as a burn.

Recommendation:

If intentional, document this explicitly as a compatibility caveat. Otherwise, consider allowing these transfers (with clear semantics) or providing dedicated burn/escrow flows.

LOW

Fee-on-transfer / rebasing ERC20 assets break accounting (share over-minting, insolvency/DoS)

Locations:

`contracts/VaultV3.vy:644-666`

`lib/tokenized-strategy/src/TokenizedStrategy.sol:954-978`

`contracts/VaultV3.vy:1086-1105`

`contracts/VaultV3.vy:1022-1044`

Description:

Both `VaultV3` and `TokenizedStrategy` assume that the `assets` argument equals the actual amount moved by ERC20 transfers. For fee-on-transfer, rebasing-on-transfer, or otherwise non-standard ERC20s where `transferFrom`/`transfer` succeeds but the received amount differs, internal accounting (`total_idle` / `totalAssets`) becomes incorrect.

This can lead to share over-minting on deposit (depositor receives shares for assets that never arrived), insolvency, and/or permanent DoS of withdraw/redeem/report paths due to inconsistent accounting.

Evidence:

VaultV3: deposit credits `total_idle` by the requested amount, not the received amount:

`contracts/VaultV3.vy`:

```
# _deposit
self._erc20_safe_transfer_from(self.asset, msg.sender, self, assets)
# Record the change in total assets.
self.total_idle += assets
# Issue shares based on pre-transfer conversion
self._issue_shares(shares, recipient)
```

If the vault receives `< assets` (fee-on-transfer), `total_idle` is overstated and `shares` are over-issued.

TokenizedStrategy: deposit credits `totalAssets` by the requested amount, not the received amount:

`lib/tokenized-strategy/src/TokenizedStrategy.sol`:

```
_asset.safeTransferFrom(msg.sender, address(this), assets);
IBaseStrategy(address(this)).deployFunds(_asset.balanceOf(address(this)));
S.totalAssets += assets;
_mint(S, receiver, shares);
```

`S.totalAssets` is increased by `assets` even if actual balance delta is smaller.

VaultV3: debt updates also assume balance deltas are well-behaved:

`contracts/VaultV3.vy` `_update_debt`:

- Withdraw path computes `withdrawn = min(post_balance - pre_balance, current_debt)` (reverts if `post_balance < pre_balance`).
- Deposit path computes `assets_to_deposit = pre_balance - post_balance` after calling `strategy.deposit` (reverts if `post_balance > pre_balance`).

These assumptions can break with rebasing/fee mechanics.

Impact:

- **Fund loss / value transfer:** users can mint more shares than backed (over-mint), diluting existing holders; withdrawals can fail once the vault/strategy becomes undercollateralized.
- **DoS:** balance-delta underflows can revert critical operations (debt management, withdrawals) for some ERC20 behaviors.

Recommendation:

- Enforce that `asset` is a strictly standard ERC20 with 1:1 transfer semantics (no fees/rebases), or
- Calculate actual received/spent amounts via `balanceBefore/balanceAfter` deltas and use those deltas for accounting and share issuance/burn logic.
- Consider explicit runtime checks that revert on unexpected balance deltas to prevent silent insolvency.

LOW

ITokenizedStrategy overloads maxWithdraw/maxRedeem with a `maxLoss` parameter that is ignored (ABI footgun for loss-aware integrations)

Locations:

`lib/tokenized-strategy/src/interfaces/ITokenizedStrategy.sol:75-83`

`lib/tokenized-strategy/src/TokenizedStrategy.sol:782-816`

Description:

`ITokenizedStrategy` declares overloads of `maxWithdraw`/`maxRedeem` that accept an extra `uint256 maxLoss` parameter:

```
function maxWithdraw(address owner, uint256 /*maxLoss*/) external view returns (uint256);
function maxRedeem(address owner, uint256 /*maxLoss*/) external view returns (uint256);
```

However, the TokenizedStrategy implementation explicitly **ignores** the `maxLoss` parameter and returns the same value as the standard ERC-4626 `maxWithdraw(owner)` / `maxRedeem(owner)`.

This creates an API trap for integrators who (reasonably) assume the overload exists to compute a loss-tolerance-aware maximum. Such integrations may: 1) Query `maxWithdraw(owner, maxLoss=0)` (expecting a conservative limit), 2) Attempt `withdraw(..., maxLoss=0)` up to that amount, 3) Encounter reverts due to loss checks even though they stayed within the queried “max”.

Evidence:

Implementation comment and behavior:

```
function maxWithdraw(address owner, uint256 /*maxLoss*/) external view returns (uint256) {
 return _maxWithdraw(_strategyStorage(), owner);
}

function maxRedeem(address owner, uint256 /*maxLoss*/) external view returns (uint256) {
 return _maxRedeem(_strategyStorage(), owner);
}
```

Impact:

Low direct fund risk, but can cause:

- Automated withdrawal/rebalance logic to revert unexpectedly.
- Frontends/bots to miscalculate executable withdraw/redeem amounts under a specified `maxLoss`, leading to repeated failures and operational DoS.

Recommendation:

Either (a) compute a loss-aware maximum based on `maxLoss`, or (b) make the “ignored for ABI compatibility” behavior explicit in the interface-level NatSpec and/or rename to avoid implying semantic relevance.

LOW

Management can be set to the strategy itself, potentially causing irreversible loss of admin control (self-owned strategy lockout)

Locations:

`lib/tokenized-strategy/src/TokenizedStrategy.sol:433-474`

`lib/tokenized-strategy/src/TokenizedStrategy.sol:1502-1520`

Description:

`management` is only checked against `address(0)` during initialization and when setting `pendingManagement`. There is **no check preventing `management` from becoming `address(this)`**.

If `management` ever becomes the strategy address, all `onlyManagement` functions become effectively **un-callable by any external account** (since `msg.sender` can't equal `address(this)` in a normal external call). This can permanently lock configuration updates (fee recipient, profit unlock time, keeper/emergency admin updates, etc.).

This can happen via:

- `initialize()` setting `_management = address(this)` (possible if the initializer is malicious or misconfigured)
- `setPendingManagement(address(this))` followed by a self-call path to `acceptManagement()` (possible if a derived strategy introduces any external self-call helper, or via future extensions)

Evidence:

Initialization only forbids zero:

```
require(_management != address(0), "ZERO ADDRESS");
S.management = _management;
```

Pending management only forbids zero:

```
function setPendingManagement(address _management) external onlyManagement {
 require(_management != address(0), "ZERO ADDRESS");
 _strategyStorage().pendingManagement = _management;
}
```

Impact:

If `management == address(this)`:

- `setKeeper`, `setEmergencyAdmin`, `setPerformanceFee`, `setPerformanceFeeRecipient`, `setProfitMaxUnlockTime`, `setName`, etc. become permanently inaccessible to operators.
- Emergency response can also be impaired if `emergencyAdmin` is unset/mis-set.

This is an access-control footgun that can permanently strand governance/configuration control.

Recommendation:

Disallow `address(this)` as `management` (and likely as `pendingManagement`) in `initialize()` and `setPendingManagement()`. Consider additional checks to ensure the new management address is capable of accepting/operating (policy-dependent).

LOW

Management transfer does not revoke/rotate other privileged roles (keeper/emergencyAdmin), allowing outgoing management to retain power after handover

Locations:

`lib/tokenized-strategy/src/TokenizedStrategy.sol:1502-1520`

`lib/tokenized-strategy/src/TokenizedStrategy.sol:1540-1546`

`lib/tokenized-strategy/src/TokenizedStrategy.sol:1337-1364`

Description:

The contract uses a 2-step process to transfer `management` (`setPendingManagement` ' `acceptManagement`), but **does not automatically rotate or revoke other privileged roles** (`keeper`, `emergencyAdmin`) when `management` changes.

Because `keeper` can call `report()`/`tend()` and `emergencyAdmin` can call `shutdownStrategy()`/`emergencyWithdraw()`, an outgoing management can: 1) set `keeper` and/or `emergencyAdmin` to themselves (or another address they control) 2) transfer `management` to a new party 3) **retain privileged operational/emergency powers indefinitely** unless the new management actively changes these roles.

This creates a privilege-transfer gap: recipients of `management` may assume they have exclusive control, but previous controllers can still execute sensitive operations.

Evidence:

Management transfer only updates management/pendingManagement:

```
function acceptManagement() external {
 StrategyData storage S = _strategyStorage();
 require(msg.sender == S.pendingManagement, "!pending");
 S.management = msg.sender;
 S.pendingManagement = address(0);
}
```

Emergency actions gated by `management || emergencyAdmin`:

```
modifier onlyEmergencyAuthorized() {
 requireEmergencyAuthorized(msg.sender);
 _;
}
...
function shutdownStrategy() external onlyEmergencyAuthorized { ... }
function emergencyWithdraw(uint256 amount) external nonReentrant onlyEmergencyAuthorized { ...
}
```

Impact:

If management is transferred (e.g., to a new multisig), the previous management can continue to:

- shutdown the strategy (permanently halting deposits)
- initiate emergency withdrawals after shutdown
- perform keeper operations (report/tend), potentially at adversarial timing

While the new management can later rotate these roles, the window and the “hidden retained privilege” are a real operational/security risk.

Recommendation:

On `acceptManagement()`, consider automatically resetting `keeper` and/or `emergencyAdmin` (or emitting explicit events / requiring explicit acceptance/confirmation for these roles). At minimum, document clearly that transferring `management` does not revoke other roles and that the new management must rotate them immediately.

LOW

Multiplication-before-division in share/asset conversions and profit unlocking can overflow and revert (DoS on extreme values)

Locations:

`contracts/VaultV3.vy:439-485`

`contracts/VaultV3.vy:412-415`

`contracts/VaultV3.vy:1296-1301`

Description:

Several core accounting paths multiply two potentially large `uint256` values before dividing. In Vyper, these `*` operations are overflow-checked and will **revert** if the intermediate product exceeds `2**256-1`. This can cause unexpected denial-of-service for deposits/withdrawals/reporting when values become very large.

While these extremes may be unlikely for typical 18-decimal assets, the vault does not enforce caps that guarantee safety (e.g., via `deposit_limit = max_uint256`, unusually large-precision assets, or very large share supply).

Evidence:

Conversions:

```
numerator: uint256 = shares * self._total_assets()
amount: uint256 = numerator / total_supply
```

(contracts/VaultV3.vy:451-452)

```
numerator: uint256 = assets * total_supply
shares: uint256 = numerator / total_assets
```

(contracts/VaultV3.vy:479-480)

Profit unlocking arithmetic:

```
unlocked_shares = self.profit_unlocking_rate * (block.timestamp - self.last_profit_update) /
MAX_BPS_EXTENDED
```

(contracts/VaultV3.vy:414)

```
self.profit_unlocking_rate = total_locked_shares * MAX_BPS_EXTENDED / new_profit_locking_pe-
riod
```

(contracts/VaultV3.vy:1298)

Impact:

If intermediate products overflow, **core user actions** (`deposit`, `mint`, `withdraw`, `redeem`, `max*` views) and/or **reporting** may revert unexpectedly once values grow large enough.

Recommendation:

Use a mulDiv-style computation (512-bit intermediate) for `a*b/c` style expressions, or restructure math to divide earlier where safe and consistent with desired rounding.

LOW

No validation on accountant-returned `total_fees` can trigger arithmetic underflow/overflow and DoS reporting

Locations:

`contracts/VaultV3.vy:1173-1203`

`contracts/VaultV3.vy:1221-1237`

Description:

`_process_report()` trusts the external `accountant.report()` return value `total_fees` (denominated in assets) with no bounds checking. This value is later used in arithmetic that can overflow or underflow:

- `shares_to_burn = _convert_to_shares(loss + total_fees, ROUND_UP)`
- `ending_supply = total_supply + shares_to_lock - shares_to_burn - _unlocked_shares()`
- proportional fee calculations `shares_to_burn * total_fees / (loss + total_fees)`

If `total_fees` is unexpectedly large (e.g., due to a buggy/malicious accountant implementation or misconfiguration), these computations can:

- overflow intermediate products (checked arithmetic) and revert,
- underflow `ending_supply` and revert,
- effectively **brick `process_report()`** until governance updates the accountant.

`total_refunds` is clamped to available balance/allowance, but `total_fees` is not clamped relative to vault assets, gains, or debt.

Evidence:

```
if accountant != empty(address):
 total_fees, total_refunds = IAccountant(accountant).report(strategy, gain, loss)
...
if loss + total_fees > 0:
 shares_to_burn = self._convert_to_shares(loss + total_fees, Rounding.ROUND_UP)
...
ending_supply: uint256 = total_supply + shares_to_lock - shares_to_burn - self._un-
locked_shares()
```

(contracts/VaultV3.vy:1177-1198, 1225)

Impact:

- **Reporting DoS**: `process_report()` can revert, preventing profit/loss realization and potentially freezing core accounting updates.
- Downstream effects: inaccurate maxWithdraw/maxRedeem behavior (since `total_debt` / `current_debt` are not updated) and operational disruption.

Recommendation:

Add sanity checks/caps on `total_fees` (e.g., relative to `gain`, `total_assets`, or configurable maxima) before using it in conversions and supply arithmetic, and/or handle extreme values gracefully to avoid underflow/overflow reverts.

LOW

Profit-locking mechanism still allows JIT deposits to capture previously-earned profits as they unlock (VaultV3 + TokenizedStrategy)

Locations:

```
contracts/VaultV3.vy:402-428
contracts/VaultV3.vy:1213-1303
contracts/VaultV3.vy:1791-1870
lib/tokenized-strategy/src/TokenizedStrategy.sol:829-867
lib/tokenized-strategy/src/TokenizedStrategy.sol:1081-1256
lib/tokenized-strategy/src/TokenizedStrategy.sol:1275-1288
```

Description:

Both VaultV3 and TokenizedStrategy use a “profit locking” design: profits are accounted and then represented by shares minted to the vault/strategy itself, which are gradually unlocked over `profit_max_unlock_time`.

This smooths immediate PPS spikes, but it does **not** prevent economic extraction via **just-in-time (JIT) deposits**:

- After a profitable report, some profit remains locked and will be unlocked over time.
- A user can deposit *after* the profit was earned (and after the report) but *before* it is unlocked.
- That depositor will share in the future PPS increase as locked shares unlock, capturing yield that was generated before they entered.

Because deposits are permissionless and reports/unlock schedules are predictable, this can be executed via MEV (deposit shortly after a report) and can systematically dilute long-term holders’ yield.

Evidence:

VaultV3: unlocked shares are subtracted from supply over time, increasing PPS for current share holders:

```
def _unlocked_shares() -> uint256:
 if full_profit_unlock_date > block.timestamp:
 unlocked_shares = profit_unlocking_rate * (block.timestamp - last_profit_update) /
MAX_BPS_EXTENDED

def _total_supply() -> uint256:
 return self.total_supply - self._unlocked_shares()
```

VaultV3 locks profit by issuing shares to itself during `_process_report`:

```
if gain + total_refunds > 0 and profit_max_unlock_time != 0:
 shares_to_lock = self._convert_to_shares(gain + total_refunds, Rounding.ROUND_DOWN)
...
ending_supply = total_supply + shares_to_lock - shares_to_burn - self._unlocked_shares()
...
self._issue_shares(..., self)
```

TokenizedStrategy uses the same pattern:

```
function _totalSupply(...) internal view returns (uint256) {
 return S.totalSupply - _unlockedShares(S);
}
...
sharesToLock = _convertToShares(S, profit, Math.Rounding.Down);
...
_mint(S, address(this), sharesToLock - sharesToBurn);
```

Impact:

- Systematic yield leakage: short-term capital can repeatedly enter after reports to capture upcoming unlock, reducing long-term depositor returns.
- MEV amplification: bots can monitor reports and deposit immediately afterward at scale.

Recommendation:

If this behavior is undesirable, consider adding anti-JIT mechanisms (e.g., deposit fees around reports, cooldown/lockup, time-weighted shares, or excluding newly-minted shares from receiving unlock for some period).

LOW

Profit-locking share minting rounds down, allowing immediate PPS increase despite intent to avoid spikes

Locations:

`contracts/VaultV3.vy:1214-1219`

`contracts/VaultV3.vy:439-485`

Description:

During reporting, the vault tries to mint `shares_to_lock` to itself to avoid an immediate PPS increase from `gain + total_refunds`:

```
# Get the amount we will lock to avoid a PPS increase.
if gain + total_refunds > 0 and profit_max_unlock_time != 0:
 shares_to_lock = self._convert_to_shares(gain + total_refunds, Rounding.ROUND_DOWN)
```

(contracts/VaultV3.vy:1216-1219)

However, rounding **down** can mint *fewer* shares than required to fully neutralize the PPS increase (especially when `total_assets` does not divide the numerator). In edge cases (e.g., high PPS and small gains), this can mint 0 shares for a non-zero gain, producing an immediate PPS increase contrary to the stated purpose.

Impact:

- Small but systematic “leakage” of profit into immediate PPS rather than being fully smoothed by the unlock schedule.
- This contradicts the code comments/intent (“avoid a PPS increase”), and can accumulate over many reports.

Recommendation:

Revisit rounding direction for `shares_to_lock` so it matches the intended invariant (no immediate PPS increase), or adjust documentation/comments if the behavior is intentional.

LOW

Read-only reentrancy: view getters can return transiently inconsistent accounting during strategy deposit/withdraw and refund pulls

Locations:

`contracts/VaultV3.vy:1046-1105`

`contracts/VaultV3.vy:714-909`

`contracts/VaultV3.vy:1115-1251`

Description:

The vault performs **external calls while its internal accounting is temporarily inconsistent** (by design, because accounting is updated after measuring pre/post token balances). During these external calls, a malicious callee (strategy/accountant) or token hook can perform **read-only reentrancy** into the vault's `@view` functions and observe (or feed to other protocols) values that are inconsistent with the real in-flight asset location.

Although these are view-only reads, this pattern is the same class of risk as the historical "read-only reentrancy" exploits: external contracts may treat vault views (e.g., `totalIdle`, `totalDebt`, `maxWithdraw`, `pricePerShare`) as reliable in-transaction signals.

Evidence:

1) Debt increase deposits: external call occurs before `total_idle/total_debt` are updated:

In `_update_debt` when increasing a strategy's debt:

- The vault **approves** and then calls `IStrategy(strategy).deposit(...)`.
- Only **after** the external call returns does it update `self.total_idle` / `self.total_debt` based on the observed balance delta.

```
# contracts/VaultV3.vy
self._erc20_safe_approve(_asset, strategy, assets_to_deposit)
pre_balance: uint256 = ERC20(_asset).balanceOf(self)
IStrategy(strategy).deposit(assets_to_deposit, self)  # external call
post_balance: uint256 = ERC20(_asset).balanceOf(self)
...
assets_to_deposit = pre_balance - post_balance
self.total_idle -= assets_to_deposit
self.total_debt += assets_to_deposit
```

(see `contracts/VaultV3.vy:1086-1103`)

During the `deposit()` external call, the vault's token balance may have already decreased (strategy pulled funds), but `total_idle` has not yet been reduced.

2) Withdraw loop: per-strategy debt updated mid-loop while `total_debt` is committed only at the end:

Within `_redeem`, after withdrawing from a strategy, the vault updates `self.strategies[strategy].current_debt` immediately, but `self.total_debt` is only committed after the loop finishes.

```
self.strategies[strategy].current_debt = new_debt
...
# later, after loop
self.total_debt = current_total_debt
```

(see `contracts/VaultV3.vy:874-877` and `contracts/VaultV3.vy:892-894`)

Any external staticcalls made later in the loop (e.g., strategy `maxRedeem/convertToAssets`) can read these transiently inconsistent values via vault view functions.

3) Reporting: shares are minted/burned before refunds are pulled:

In `_process_report`, share supply changes happen before pulling refunds via `transferFrom`:

```
self._issue_shares(...)
# or self._burn_shares(...)
...
self._erc20_safe_transfer_from(_asset, accountant, self, total_refunds)  # external call
self.total_idle += total_refunds
```

(see `contracts/VaultV3.vy:1227-1250`)

During the refund `transferFrom` call, external observers can see share supply already updated while `total_idle` (and thus `totalAssets` accounting) has not yet incorporated refunds.

Impact:

- **Read-only reentrancy:** A malicious strategy/accountant (or any callback invoked by the asset token) can re-enter the vault's view methods during these external calls.
- Protocols that (incorrectly) use vault views like `totalIdle`, `totalDebt`, `pricePerShare`, `maxWithdraw`, etc., as in-transaction oracles may be manipulated/griefed.
- Even within Yearn's own ecosystem, this can create surprising execution-order dependencies if other contracts read these views during callbacks.

Recommendation:

Consider adding a "read-lock" pattern for sensitive view functions during operations that include external calls (common mitigations include:

- returning cached pre-call values,
- reverting view functions when a mutable lock is held, or
- documenting strongly that these view values are not safe during in-transaction callbacks).

At minimum, document which view getters can be transiently inconsistent during `update_debt`, `_redeem`, and `_process_report` external call windows.

LOW

Reporting can become unexecutable due to unbounded external-call complexity (strategy/accountant/factory), freezing accounting and blocking management actions

Locations:

`contracts/VaultV3.vy:1114-1325`

`contracts/VaultV3.vy:1636-1646`

Description:

`process_report()` is a critical state transition for recognizing profit/loss, pulling refunds, charging fees, and updating profit-locking parameters. It performs multiple external calls into:

- The strategy (`balanceOf`, `convertToAssets`)
- The configured accountant (`report`)
- The factory/deployer (`protocol_fee_config`)
- The asset token (`balanceOf`, `allowance`, `transferFrom`)

While there are no explicit loops here, the computational cost is **entirely dependent** on the called contracts' implementations, which can include unbounded iteration or heavy computation.

If any of these dependencies is expensive enough (or reverts), `process_report()` can become practically uncallable under gas limits, freezing reporting. This can cascade into stuck operational states because other management actions (e.g., debt reduction) may rely on fresh reporting to proceed safely (see `_update_debt` unrealised-loss gate).

Evidence:

External calls in `_process_report()` include:

```
strategy_shares = IStrategy(strategy).balanceOf(self)
total_assets = IStrategy(strategy).convertToAssets(strategy_shares)
...
total_fees, total_refunds = IAccountant(accountant).report(strategy, gain, loss)
...
protocol_fee_bps, protocol_fee_recipient = IFactory(self.factory).protocol_fee_config()
...
total_refunds = min(total_refunds, min(ERC20(_asset).balanceOf(accountant), ERC20(_asset).al-
lowance(accountant, self)))
self._erc20_safe_transfer_from(_asset, accountant, self, total_refunds)
```

(contracts/VaultV3.vy:1148-1184, 1205-1211, 1246-1249)

Entry point:

```
def process_report(strategy: address) -> (uint256, uint256):
 self._enforce_role(msg.sender, Roles.REPORTING_MANAGER)
 return self._process_report(strategy)
```

(contracts/VaultV3.vy:1636-1646)

Impact:

- **High impact operational DoS**: inability to report leads to stale accounting (profits/losses/fees/refunds), and can block or complicate rebalancing/unwinding decisions.
- In emergencies, inability to report can prevent debt reductions (due to unrealised-loss gating) and prolong withdrawal servicing issues.

Recommendation:

Operationally constrain/reporting dependencies (only allow vetted accountant/strategy/factory implementations), and consider adding fallback mechanisms to proceed or bypass certain external calls when they are uncallable (e.g., disabling accountant, bypassing protocol fee config, or allowing limited report functionality).

LOW

Role setters allow potentially dangerous values (keeper/emergencyAdmin can be set to 0 or to the strategy itself)

Locations:

`lib/tokenized-strategy/src/TokenizedStrategy.sol:433-474`

`lib/tokenized-strategy/src/TokenizedStrategy.sol:1528-1546`

Description:

`keeper` and `emergencyAdmin` can be set to arbitrary addresses without validation. In particular:

- `initialize()` sets `S.keeper = _keeper;` with no checks.
- `setKeeper()` and `setEmergencyAdmin()` do not prevent setting these roles to `address(0)` or `address(this)`.

While `address(0)` may be intended to disable a role, setting privileged roles to `address(this)` is frequently an unsafe configuration because **self-calls** (`this.report()`, `this.tend()`, etc.) have `msg.sender == address(this)` and would then satisfy role checks if the role equals `address(this)`.

If a concrete strategy later introduces *any* external/user-triggerable path that performs such a self-call (or hook-driven internal logic), `report()` / `tend()` could become callable in unintended circumstances.

Evidence:

```
// initialize
S.keeper = _keeper;

// setters
function setKeeper(address _keeper) external onlyManagement {
 _strategyStorage().keeper = _keeper;
}

function setEmergencyAdmin(address _emergencyAdmin) external onlyManagement {
 _strategyStorage().emergencyAdmin = _emergencyAdmin;
}
```

Impact:

- Accidental misconfiguration can disable keeper/emergency operations (`address(0)`).
- Setting `keeper == address(this)` can create surprising authorization behavior where internal/self-call paths satisfy `onlyKeepers` checks, potentially widening who can effectively trigger `report()`/`tend()` depending on the strategy's additional code.

Recommendation:

Validate role addresses in `initialize()`/setters (at minimum disallow `address(this)`; consider whether `address(0)` should be allowed) to reduce configuration footguns that can weaken effective authorization.

LOW

TokenizedStrategy emergencyWithdraw can silently do nothing if strategy does not override BaseStrategy._emergencyWithdraw

Locations:

`lib/tokenized-strategy/src/TokenizedStrategy.sol:1337-1364`

`lib/tokenized-strategy/src/BaseStrategy.sol:355-440`

Description:

`TokenizedStrategy.emergencyWithdraw` is intended to allow management/emergencyAdmin to pull funds from the yield source after shutdown. However, the call chain ultimately invokes `BaseStrategy._emergencyWithdraw`, which has an empty default implementation.

If a strategist forgets to override `_emergencyWithdraw`, `emergencyWithdraw()` will succeed but perform no action, potentially giving a false sense of safety during incidents.

Evidence:

```
// TokenizedStrategy.emergencyWithdraw
require(_strategyStorage().shutdown, "not shutdown");
IBaseStrategy(address(this)).shutdownWithdraw(amount);
```

```
// BaseStrategy.shutdownWithdraw
function shutdownWithdraw(uint256 _amount) external virtual onlySelf {
 _emergencyWithdraw(_amount);
}

// BaseStrategy default hook
function _emergencyWithdraw(uint256 _amount) internal virtual {}
```

Impact:

Low to Medium depending on strategy complexity: in emergencies, inability to manually unwind may prolong or worsen fund lockups.

Recommendation:

Make `_emergencyWithdraw` abstract (required) for strategies that may need it, or emit an event / revert by default to avoid silent no-ops.

LOW

TokenizedStrategy withdraw allows receiver == strategy, causing accounting desync (transfer-to-self no-op)

Location:

`lib/tokenized-strategy/src/TokenizedStrategy.sol:989-1049`

Description:

`TokenizedStrategy._withdraw` allows `receiver == address(this)`. For standard ERC20s, transferring `asset` to self is a no-op while the function still decrements `S.totalAssets` and burns shares.

This can desynchronize `S.totalAssets` from the real on-chain balance until the next `report()` updates `totalAssets`.

Evidence:

```
// lib/tokenized-strategy/src/TokenizedStrategy.sol:997-1045
require(receiver != address(0), "ZERO ADDRESS");
...
S.totalAssets -= (assets + loss);
_burn(S, owner, shares);
_asset.safeTransfer(receiver, assets);
```

No check `receiver != address(this)`.

Impact:

Low. Primarily an accounting invariant issue; may confuse integrations relying on `totalAssets` between reports.

Recommendation:

Disallow `receiver == address(this)` (mirroring `_maxDeposit`'s self-receiver restriction), or use balance-delta accounting so no-op transfers don't change `totalAssets`.

LOW

Unchecked arithmetic in report() can overflow, corrupting fee amounts and profit-locking share math

Locations:

`lib/tokenized-strategy/src/TokenizedStrategy.sol:1119-1161`

`lib/tokenized-strategy/src/TokenizedStrategy.sol:1213-1233`

Description:

`TokenizedStrategy.report()` performs several economically-significant calculations inside `unchecked` blocks (fee amounts in assets, fee amounts in shares, protocol fee split, and weighted-average profit unlocking schedule).

Because there are no explicit bounds on `profit`, `sharesToLock`, `totalFeeShares`, or the post-report share supply, these unchecked multiplications/additions can overflow and wrap, producing nonsensical values (including values larger than the true profit).

While typical ERC-20 assets will not approach `uint256` limits, this contract is permissionless/generic and can be deployed with assets that have extremely large balances/supplies (or strategies that can legitimately report very large `newTotalAssets-`). At those edge values, unchecked wraparound can:

- mint an incorrect number of fee shares (diluting depositors or underpaying fees),
- cause `sharesToLock` to underflow/wrap and trigger incorrect profit-locking behavior,
- compute a bogus `newProfitLockingPeriod` from an overflowed `previously-LockedTime`, producing an incorrect unlocking rate/date.

Evidence:

Fee arithmetic is performed unchecked:

```
// TokenizedStrategy.sol
unchecked {
 totalFees = (profit * fee) / MAX_BPS;
 totalFeeShares = (sharesToLock * fee) / MAX_BPS;
}
...
unchecked {
 protocolFeeShares = (totalFeeShares * protocolFeeBps) / MAX_BPS;
 protocolFees = (totalFees * protocolFeeBps) / MAX_BPS;
}
...
unchecked {
 _mint(S, S.performanceFeeRecipient, totalFeeShares - protocolFeeShares);
}
```

Weighted-average time component is also unchecked:

```
unchecked {
 previouslyLockedTime = (_fullProfitUnlockDate - block.timestamp) *
 (totalLockedShares - sharesToLock);
}
uint256 newProfitLockingPeriod = (previouslyLockedTime +
 sharesToLock * _profitMaxUnlockTime) / totalLockedShares;
```

Impact:

At extreme values, overflows can silently produce wrong fee share minting and wrong profit-locking schedule calculations. This can cause severe share dilution / incorrect distribution of value between users and fee recipients, and/or break the intended profit-unlocking mechanism.

Recommendation:

Avoid unchecked arithmetic for these financial calculations, or enforce safe bounds that make overflow impossible. Prefer full-precision `Math.mulDiv` for `x * y / d` patterns and checked arithmetic for intermediate products (especially in fee and weighted-average computations).

LOW

Unguarded ERC20 share functions can be reentered during withdraw/redeem external calls, enabling griefing via share movement before burn

Locations:

`contracts/VaultV3.vy:715-909`

`contracts/VaultV3.vy:1872-1905`

Description:

`withdraw`/`redeem` are `@nonreentrant("lock")`, but the vault's ERC20 share functions (`transfer`, `transferFrom`, `approve`, `permit`) are **not**.

Because `withdraw`/`redeem` perform multiple external calls (e.g., to strategies via `redeem()` and to modules), an untrusted callback contract can re-enter the vault during these calls and invoke the unguarded ERC20 share methods.

While this does not directly bypass authorization, it can be used for **griefing/DoS** in situations where the callback contract has (or can obtain) allowance from `owner` (common when vault shares are used in other protocols). By moving `owner`'s shares away after the early balance check but before `_burn_shares`, the withdrawal will revert on underflow.

Evidence:

Withdrawal burns shares only after external calls:

In `_redeem`, the vault performs strategy interactions and only later burns shares:

```
# ... external calls to strategies happen earlier in _redeem
self._burn_shares(shares, owner)  # contracts/VaultV3.vy:900-902
```

(see `_redeem`: `contracts/VaultV3.vy:715-909`)

Share transfer functions are externally callable without reentrancy lock:

```
@external
def transfer(receiver: address, amount: uint256) -> bool:
 self._transfer(msg.sender, receiver, amount)

@external
def transferFrom(sender: address, receiver: address, amount: uint256) -> bool:
 return self._transfer_from(sender, receiver, amount)
```

(see `contracts/VaultV3.vy:1882-1905`)

Impact:

A malicious or compromised callback contract (strategy/module/token) can:

- Re-enter during `withdraw`/`redeem` external calls and invoke `transferFrom(owner, ...)` (if it has allowance), reducing `owner`'s share balance.
- Cause `_burn_shares(shares, owner)` to revert, preventing withdrawals (griefing).

This is most relevant when vault shares are widely approved to third-party contracts.

Recommendation:

Consider applying the same `@nonreentrant("lock")` guard (or a separate read-lock pattern) to ERC20 share methods, or otherwise ensure that `withdraw`/`redeem` cannot be influenced by share movements during their external-call phase (e.g., by moving the burn earlier where safe, or snapshotting and enforcing balance invariants).

LOW

Vault uses `strategy.convertToAssets()` as a valuation oracle for reporting and unrealised-loss accounting (manipulable/stale valuations can mint/burn shares incorrectly)

Locations:

```
contracts/VaultV3.vy:667-694
contracts/VaultV3.vy:534-641
contracts/VaultV3.vy:714-908
contracts/VaultV3.vy:1114-1325
```

Description:

The vault treats `IStrategy(strategy).convertToAssets(IStrategy(strategy).balanceOf(vault))` as the authoritative valuation of its strategy position. This value is used as an **oracle** in multiple critical places:

- `process_report()` gain/loss computation (affects `total_debt`, strategy debts, profit-locking share mint/burn, and fee minting)
- `_assess_share_of_unrealised_losses()` (drives how much "unrealised loss" a withdrawing user is forced to realize)
- `maxWithdraw()` simulation / withdraw execution logic

ERC-4626 explicitly does **not** guarantee `convertToAssets` is a manipulation-resistant oracle. Many ERC-4626 implementations derive `totalAssets()` from `asset.balanceOf(strategy)` (donation/manipulable), use delayed pricing, or incorporate external oracles/pools. If `convertToAssets` can be manipulated up or down (e.g., via MEV-driven price manipulation of a strategy's valuation inputs), the vault can:

- report incorrect gains/losses,
- mint/burn the wrong number of shares,
- issue fees on "phantom profits",
- and/or force withdrawing users to take losses that are not economically real.

The contract even acknowledges this risk (comment at ~1146) but provides no mitigation.

Evidence:

Reporting relies on strategy valuation:

```
strategy_shares: uint256 = IStrategy(strategy).balanceOf(self)
total_assets = IStrategy(strategy).convertToAssets(strategy_shares)
...
if total_assets > current_debt:
 gain = total_assets - current_debt
else:
 loss = current_debt - total_assets
```

(see `_process_report` around lines 1145-1170)

Withdraw path uses the same valuation to compute “unrealised losses”:

```
vault_shares = IStrategy(strategy).balanceOf(self)
strategy_assets = IStrategy(strategy).convertToAssets(vault_shares)
...
users_share_of_loss = assets_needed - (assets_needed * strategy_assets / strategy_cur-
rent_debt)
```

(see `_assess_share_of_unrealised_losses` around lines 679-694)

Impact:

- **High-impact accounting corruption**: incorrect `gain/loss` changes `total_debt`/strategy debt and profit-locking shares, propagating into all share conversions (`deposit/mint/withdraw/redeem`).
- **Fee extraction on manipulated valuations**: if a report observes inflated `convertToAssets`, the vault can mint fee shares to the accountant/protocol even without real profit; later corrections manifest as losses borne by shareholders.
- **User harm / MEV griefing**: if `convertToAssets` is manipulable downwards, withdrawers can be forced to “realize” artificial losses, transferring value to remaining shareholders.

Recommendation:

Treat `convertToAssets` as untrusted external data:

- Restrict strategies to audited/approved implementations with manipulation-resistant accounting (or explicitly document and enforce requirements).
- Consider using realizable values derived from actual redemption outcomes (where possible) instead of pure view-based valuation.
- Add sanity bounds / circuit-breakers (e.g., max deviation vs last report, min/max rate-of-change) if strategies rely on external prices.

LOW

VaultFactory allows setting protocol fee recipient to non-withdrawable address (e.g., the Factory itself), permanently burning fee value and diluting users

Locations:

`contracts/VaultFactory.vy:266-287`

`contracts/VaultFactory.vy:235-263`

`contracts/VaultFactory.vy:289-310`

Description:

`VaultFactory` lets governance set `protocol_fee_recipient` to any non-zero address, including contracts that **cannot transfer/redeem** the fee shares they receive.

This is an asset-management issue because Yearn V3 protocol fees are paid by **minting vault/strategy shares** to the recipient. If the recipient is a contract with no method to move/redeem those shares (e.g., the `VaultFactory` contract itself, or an address that is not able to execute calls), the fee shares become **permanently stuck**. The protocol still "charges" fees (diluting vault/strategy share value), but the value is effectively **burned** rather than being receivable by anyone.

This violates the plumbing principle: fee value enters the recipient address, but has no outlet.

Evidence:

`set_protocol_fee_recipient` only checks non-zero:

```
@external
def set_protocol_fee_recipient(new_protocol_fee_recipient: address):
 assert msg.sender == self.governance, "not governance"
 assert new_protocol_fee_recipient != empty(address), "zero address"
 ...
 self.default_protocol_fee_data = self._pack_protocol_fee_data(
 new_protocol_fee_recipient,
 self._unpack_protocol_fee(default_fee_data),
 False
 )
```

No check prevents setting `new_protocol_fee_recipient == self` (the factory), nor does it ensure the recipient can ever move/redeem received fee shares.

The same lack of validation applies for fee-enabling operations (`set_protocol_fee_bps` / `set_custom_protocol_fee_bps`) that depend on a correct recipient configuration.

Impact:

- **Permanent value loss for vault/strategy users** if protocol fees are enabled while the recipient is a sink: shares are minted (dilution occurs) but can never be redeemed/used.
- **Protocol revenue blackhole**: fees are charged but irrecoverable.

Likelihood is **medium** (requires governance misconfiguration), but impact is **high** once enabled because every report that charges fees can permanently leak value.

Recommendation:

Prevent obviously sink recipients (at minimum `new_protocol_fee_recipient != self`), and consider adding stronger recipient validation (e.g., require a governance-controlled EOA/multisig or a recipient contract that can execute arbitrary calls).

LOW

VaultFactory.transferGovernance allows setting pendingGovernance to zero address, risking permanent governance lockout

Location:

`contracts/VaultFactory.vy:342-365`

Description:

`transferGovernance` does not validate `new_governance != empty(address)`. If governance mistakenly sets `pendingGovernance` to `empty(address)`, then `acceptGovernance` becomes uncallable (no account can be `msg.sender == 0x0`). If the current governance key is later lost, the factory becomes permanently unable to update protocol fee settings or shutdown.

Evidence:

```
# contracts/VaultFactory.vy:342-365
@external
def transferGovernance(new_governance: address):
 assert msg.sender == self.governance, "not governance"
 self.pendingGovernance = new_governance

@external
def acceptGovernance():
 assert msg.sender == self.pendingGovernance, "not pending governance"
 ...
```

Impact:

Low likelihood (requires privileged action) but realistic operational risk.

Recommendation:

Require `new_governance != empty(address)` (and optionally add a cancel/reset mechanism).

LOW

Withdraw/maxWithdraw execution cost depends on strategy/module implementation; expensive view logic can make core flows exceed gas limits

Locations:

```
contracts/VaultV3.vy:536-641
contracts/VaultV3.vy:696-713
contracts/VaultV3.vy:762-889
contracts/VaultV3.vy:560-568
```

Description:

Although loops are capped at `MAX_QUEUE = 10`, core user flows (`withdraw`/`redeem` via `_redeem()`, and `maxWithdraw`/`maxRedeem` via `_max_withdraw()`) perform **multiple external calls per queue entry**:

- `IStrategy.maxRedeem()` + `convertToAssets()`
- `_assess_share_of_unrealised_losses()` which calls `IStrategy.balanceOf()` + `convertToAssets()`
- `_withdraw_from_strategy()` which calls `previewWithdraw()` + `balanceOf()` + `redeem()`
- ERC20 `balanceOf()` checks around withdrawals
- Optional `withdraw_limit_module.available_withdraw_limit()` call

These external calls execute arbitrary strategy/module bytecode and can have **unbounded internal complexity** (loops over positions, accounting scans, etc.). If any configured strategy/module has view methods or redeem paths that are too expensive, withdrawals (and even `maxWithdraw` simulations used by on-chain integrators) can consistently run out of gas and become unexecutable.

This is a compute-bounds/stuck-state issue: the vault assumes per-strategy calls are cheap enough, but does not provide a fallback path that avoids calling these potentially expensive methods.

Evidence:

Per-iteration external calls in `_redeem()`:

```
max_withdraw = IStrategy(strategy).convertToAssets(IStrategy(strategy).maxRedeem(self))
unrealised_losses_share = self._assess_share_of_unrealised_losses(strategy, current_debt, as-
sets_to_withdraw)
self._withdraw_from_strategy(strategy, assets_to_withdraw)
```

(contracts/VaultV3.vy:794-846)

`_withdraw_from_strategy()` adds further external calls:

```
shares_to_redeem = min(IStrategy(strategy).previewWithdraw(assets_to_withdraw), IStrategy(-
strategy).balanceOf(self))
IStrategy(strategy).redeem(shares_to_redeem, self, self)
```

(contracts/VaultV3.vy:704-713)

`_max_withdraw()` similarly loops and calls into strategy code for simulation (contracts/VaultV3.vy:585-605).

An additional external dependency can be inserted via `withdraw_limit_module`:

```
IWithdrawLimitModule(withdraw_limit_module).available_withdraw_limit(owner, max_loss-
, strategies)
```

(contracts/VaultV3.vy:560-568, 744-747)

Impact:

- **High impact availability risk**: withdrawals/redeems can become uncallable under gas limits if any selected/default strategy or limit module is computationally heavy.
- If most assets are deployed into such strategies, user funds can be effectively locked until privileged roles reconfigure or rescue via off-path operations (which may themselves depend on the same expensive external code).

Recommendation:

Introduce stronger bounds/guardrails around external-call complexity (e.g., avoid repeated view calls in loops, cache results, allow skipping problematic strategies, or provide an emergency withdrawal path that does not depend on expensive view methods). Also constrain/validate strategy/module implementations operationally before activation.

LOW

`auto_allocate` lets any depositor trigger large strategy deposits, creating MEV/sandwich and griefing surface against strategy deposit execution

Locations:

`contracts/VaultV3.vy:643-666`

`contracts/VaultV3.vy:964-1111`

Description:

When `auto_allocate` is enabled, every user `deposit()`/`mint()` will automatically call `_update_debt(default_queue[0], max_uint, 0)`.

This makes **strategy allocation an attacker-triggerable action**:

- An attacker can deposit a tiny amount ("dust") to force the vault to deposit potentially **all available idle** into the first strategy.
- If that strategy's `deposit()` path is price-sensitive (swaps, LPing, lending rate spikes, etc.), attackers can sandwich/manipulate the strategy's execution timing to extract MEV or to deliberately cause slippage/loss.

This is a classic economic attack pattern: "use a permissionless user action to force a large trade/position adjustment at attacker-chosen time."

Evidence:

Auto-allocation triggered by any deposit:

```
def _deposit(recipient: address, assets: uint256, shares: uint256):
 ...
 if self.auto_allocate:
 self._update_debt(self.default_queue[0], max_value(uint256), 0)
```

The debt increase path can deposit substantial idle into the strategy:

```
# increasing debt
max_deposit: uint256 = IStrategy(strategy).maxDeposit(self)
...
assets_to_deposit: uint256 = new_debt - current_debt
...
self._erc20_safe_approve(_asset, strategy, assets_to_deposit)
IStrategy(strategy).deposit(assets_to_deposit, self)
...
self.total_idle -= assets_to_deposit
self.total_debt += assets_to_deposit
```

Impact:

- MEV extraction against strategies whose deposit execution is manipulable.
- Potential systematic loss for vault depositors if attackers repeatedly trigger allocations at unfavorable prices.
- Griefing/DoS if strategy deposits revert under certain conditions and attackers can force these paths repeatedly.

Recommendation:

If strategies can be MEV-sensitive, consider restricting auto-allocation triggers (e.g., keeper-only allocation, minimum deposit threshold for triggering allocation, or a separate permissioned allocation step) or only enabling `auto_allocate` when the first strategy's deposit path is robust against sandwiching.

LOW

`transfer_role_manager()` allows setting `future_role_manager` to zero address, permanently bricking role management

Location:

`contracts/VaultV3.vy:1571-1594`

Description:

`transfer_role_manager()` (step 1 of 2) does **not** validate that the new `role_manager` is non-zero:

```
@external
def transfer_role_manager(role_manager: address):
 assert msg.sender == self.role_manager
 self.future_role_manager = role_manager
```

If the current `role_manager` accidentally (or maliciously) sets `future_role_manager = empty(address)`, then `accept_role_manager()` becomes uncallable:

```
@external
def accept_role_manager():
 assert msg.sender == self.future_role_manager
```

Because `msg.sender` can never be the zero address, no one can accept, and the system is stuck with the old `role_manager` forever (and with `future_role_manager` pointing at `0x0`). There is also no “cancel transfer” function; the only way to recover is for the **current** `role_manager` to call `transfer_role_manager()` again—but if the current role manager key is lost or was intended to be rotated away, governance can be permanently broken.

Impact:

Permanent or long-lived **loss of administrative control** over:

- role assignment (`set_role`/`add_role`/`remove_role`)
- configuration setters and emergency controls (all gated via roles/role_manager)
- strategy management and reporting permissions

This is a realistic misconfiguration footgun and also enables a malicious role manager to irreversibly freeze governance without explicitly signaling an intentional renounce.

Recommendation:

Require `role_manager != empty(address)` in `transfer_role_manager()` (or implement an explicit, deliberate renounce flow if that behavior is desired).

LOW

auto_allocate can be enabled with empty default_queue, causing deposit/mint DoS via out-of-bounds access

Locations:

`contracts/VaultV3.vy:643-666`

`contracts/VaultV3.vy:1393-1406`

Description:

When `auto_allocate` is enabled, `_deposit()` unconditionally indexes `self.default_queue[0]`.

`set_auto_allocate()` explicitly warns that an empty `default_queue` will cause deposits to fail, but it does not enforce the precondition (non-empty queue) before enabling `auto_allocate`.

If `default_queue` is empty (the default initial state, or after misconfiguration), any `deposit()`/`mint()` call will revert due to an out-of-bounds array access, effectively pausing deposits.

Evidence:

```
# contracts/VaultV3.vy
@internal
def _deposit(recipient: address, assets: uint256, shares: uint256):
 ...
 if self.auto_allocate:
 self._update_debt(self.default_queue[0], max_value(uint256), 0)
```

```
@external
def set_auto_allocate(auto_allocate: bool):
 """
 ...
 NOTE: An empty `default_queue` will cause deposits to fail.
 """
 self._enforce_role(msg.sender, Roles.DEBT_MANAGER)
 self.auto_allocate = auto_allocate
```

Impact:

A `DEBT_MANAGER` (or an accidental operational mistake) can enable `auto_allocate` while `default_queue` is empty, causing all user deposits/mints to revert until governance/management intervenes to either:

- set a non-empty `default_queue`, or
- disable `auto_allocate`.

This is a core-function DoS (deposits) triggered by invalid configuration input.

Recommendation:
Add input/state validation when enabling `auto_allocate` (or before indexing the queue), e.g. require `len(default_queue) > 0` when setting `auto_allocate=True`, or skip auto-allocation when the queue is empty.

LOW

initialize() does not validate _asset != address(this); strategy can be initialized with itself as underlying asset

Location:

`lib/tokenized-strategy/src/TokenizedStrategy.sol:433-474`

Description:

`initialize()` accepts an `_asset` address and sets `S.asset = ERC20(_asset)` without validating that the asset is not the strategy contract itself.

If `_asset == address(this)`, the call to `ERC20(_asset).decimals()` will **succeed** (because `TokenizedStrategy` itself implements `decimals()`), returning the default/uninitialized `S.decimals` value during initialization. This allows the strategy to be initialized in a **self-referential** configuration where:

- the ERC-4626 “asset” token is the same contract as the strategy share token.

This violates core assumptions throughout the codebase (that `asset` is an external ERC-20 distinct from the share token), and can lead to nonsensical accounting, broken deposit/withdraw flows, and unexpected behavior in strategist-implemented `_deployFunds/_freeFunds/_harvestAndReport` which will now interact with the share token rather than a real underlying.

Evidence:

```
function initialize(
 address _asset,
 string memory _name,
 address _management,
 address _performanceFeeRecipient,
 address _keeper
) external {
 ...
 S.asset = ERC20(_asset);
 ...
 S.decimals = ERC20(_asset).decimals();
 ...
}
```

Impact:

- **Misconfiguration footgun**: A strategy can be deployed/initialized into an invalid state that does not represent a real ERC-4626 vault.
- Potential **accounting corruption / unexpected mint-burn dynamics** because deposits/withdrawals and strategist hooks will treat strategy shares as the underlying “asset”.

While typical deployments via `BaseStrategy` pass an external asset, permissionless deployments or CREATE2 setups could accidentally (or intentionally) set `_asset` to the strategy’s own address.

Recommendation:

In `initialize()`, add explicit validation that `_asset != address(this)` (and typically also ensure `_asset.code.length > 0` and/or is an expected ERC-20) to prevent self-referential and non-token assets.

LOW

maxWithdraw/maxRedeem can misrepresent withdrawable value when losses are allowed (returns requestable amount, not receivable assets)

Locations:

`contracts/VaultV3.vy:534-641`

`contracts/VaultV3.vy:2042-2083`

`contracts/VaultV3.vy:780-908`

Description:

The `maxWithdraw` / `maxRedeem` logic is implemented via `_max_withdraw`, which simulates how much the user can *request* in a withdrawal without reverting given liquidity and `max_loss`. However, the vault's actual withdrawal flow (`_redeem`) can reduce the amount actually paid out (`requested_assets`) to account for unrealised and realised strategy losses.

As a result, when `max_loss` permits losses (e.g. `MAX_BPS`), `maxWithdraw` may return a value that is **not** the maximum assets the user can *receive*, but rather the maximum assets the user can *ask for* (and then receive less). This is a semantic mismatch with the ERC-4626 meaning of `maxWithdraw` ("maximum amount of assets that can be withdrawn") and can break integrators that treat the return value as receivable assets.

Evidence:

`_max_withdraw` explicitly treats unrealised loss as part of the "withdrawable" request:

```
# _max_withdraw doc
# i.e. If we have 100 debt and 10 of unrealised loss, the max we can get
# out is 90, but a user of the vault will need to call withdraw with 100
# in order to get the full 90 out.
```

(contracts/VaultV3.vy:552-555)

In the loop, `have` is increased by `to_withdraw` (which can include an unrealised loss component) and ultimately assigned back to `max_assets`:

```
have += to_withdraw
...
max_assets = have
```

(contracts/VaultV3.vy:627-640)

Actual payout is reduced inside `_redeem`:

Unrealised losses reduce the eventual payout (`requested_assets`) even if the user "requests" more:

```
requested_assets: uint256 = assets
...
requested_assets -= unrealised_losses_share
...
requested_assets -= loss
...
self._erc20_safe_transfer(_asset, receiver, requested_assets)
```

(contracts/VaultV3.vy:753-905)

Public API claims ERC-4626 compliance:

```
@external
def maxWithdraw(...):
 """
 @notice Get the maximum amount of assets that can be withdrawn.
 @dev Complies to normal 4626 interface and takes custom params.
 """
 return self._max_withdraw(owner, max_loss, strategies)
```

(contracts/VaultV3.vy:2042-2060)

Impact:

- Integrations using `maxWithdraw(owner, MAX_BPS, ...)` to determine *receivable* assets can overestimate what users will actually receive.
- Downstream accounting/UX issues: UIs may show "you can withdraw X" but the receiver gets less.
- For protocols that rely on `maxWithdraw` as a safety check (e.g., slippage limits), this can create incorrect assumptions.

Recommendation:

Clarify semantics in documentation (distinguish "requestable" vs "receivable" assets), or change `_max_withdraw` to return the maximum receivable assets under the given `max_loss` and strategy limits (i.e., simulate the loss reductions applied in `_redeem`).

LOW

max_loss parameter not range-checked in maxWithdraw/maxRedeem and update_debt (values > 100% accepted)

Locations:

`contracts/VaultV3.vy:2044-2083`

`contracts/VaultV3.vy:1746-1762`

`contracts/VaultV3.vy:966-1035`

Description:

Several entrypoints accept a `max_loss` basis-points parameter but do not validate it is within `[0, MAX_BPS]` (0–10_000):

- `maxWithdraw(owner, max_loss, strategies)`
- `maxRedeem(owner, max_loss, strategies)`
- `update_debt(strategy, target_debt, max_loss)`

In contrast, the actual withdrawal path (`_redeem`) does enforce `max_loss <= MAX_BPS`.

Allowing `max_loss > MAX_BPS` can lead to inconsistent semantics and can also propagate unexpected values into external modules (e.g., a withdraw-limit module may assume the 0..10_000 range and revert).

Evidence:

No bound check in view functions:

```
def maxWithdraw(owner: address, max_loss: uint256 = 0, strategies: DynArray[address, MAX_QUEUE]
= []) -> uint256:
 return self._max_withdraw(owner, max_loss, strategies)

def maxRedeem(owner: address, max_loss: uint256 = MAX_BPS, strategies: DynArray[address,
MAX_QUEUE] = []) -> uint256:
 return min(self._convert_to_shares(self._max_withdraw(owner, max_loss, strategies),
Rounding.ROUND_DOWN), self.balance_of[owner])
```

No bound check in `update_debt`:

```
def update_debt(strategy: address, target_debt: uint256, max_loss: uint256 = MAX_BPS) ->
uint256:
 return self._update_debt(strategy, target_debt, max_loss)
```

Impact:

- Off-chain callers/integrations that pass user-supplied `max_loss` can experience unexpected reverts or incorrect simulations.
- The contract accepts nonsensical values (>100%) which weakens the intended “guardrail” semantics of a loss bound.

Recommendation:

Add `assert max_loss <= MAX_BPS` (and potentially document/normalize behavior) in these entrypoints to match `_redeem`’s validation.

LOW

shutdown_vault() permanently escalates EMERGENCY_MANAGER to DEBT_MANAGER (role separation bypass)

Location:

`contracts/VaultV3.vy:1764-1787`

Description:

Calling `shutdown_vault()` (requires `EMERGENCY_MANAGER`) permanently ORs the caller's roles with `DEBT_MANAGER`.

This is a **privilege escalation path**: any address that is granted `EMERGENCY_MANAGER` (often intended to be a limited "guardian" role) can grant itself full debt-management powers after shutdown, even if the role-manager did not intend that separation.

Because shutdown is irreversible, this escalation is also irreversible unless the (separate) `role_manager` later revokes it.

Code Snippet:

```
def shutdown_vault():
 self._enforce_role(msg.sender, Roles.EMERGENCY_MANAGER)
 assert self.shutdown == False
 self.shutdown = True
 ...
 new_roles: Roles = self.roles[msg.sender] | Roles.DEBT_MANAGER
 self.roles[msg.sender] = new_roles
 log RoleSet(msg.sender, new_roles)
```

Impact:

If the system's intended security model is role separation (guardian can only pause/shutdown; debt managers can move funds), this violates that model. Post-shutdown, the emergency guardian can:

- call `update_debt()` to move funds between vault and strategies (potentially realizing losses or interfering with orderly unwinds),
- influence fund availability and withdrawal outcomes.

Recommendation:
Avoid implicit role escalation. If debt-management powers are required during emergencies, either:

- require the caller already has both roles, or
- introduce a separate emergency-only unwind function set with narrowly scoped capabilities, or
- grant `DEBT_MANAGER` to a preconfigured emergency-unwind address at deployment (explicitly), rather than whichever guardian happens to call shutdown.

INFO

BaseStrategy access-control modifiers can be bypassed if strategy defines shadowing require* functions (selector collision)

Locations:

`lib/tokenized-strategy/src/BaseStrategy.sol:51-75`

`lib/tokenized-strategy/src/TokenizedStrategy.sol:297-339`

Description:

`BaseStrategy`'s access-control modifiers (`onlyManagement`, `onlyKeepers`, `onlyEmergencyAuthorized`) enforce authorization by making an **external call to `address(this)`** via the `TokenizedStrategy` interface:

```
modifier onlyManagement() {
 TokenizedStrategy.requireManagement(msg.sender);
 _;
}
```

Because `TokenizedStrategy` is set to `ITokenizedStrategy(address(this))`, the call is routed through the strategy's normal function selector dispatch. If a derived strategy contract **defines its own** `requireManagement(address)`, `requireKeeperOrManagement(address)`, or `requireEmergencyAuthorized(address)` function (intentionally or accidentally via inheritance), that local implementation will be executed **instead of** the delegated `TokenizedStrategy` implementation.

This can silently disable/alter authorization checks for any strategy functions that rely on these modifiers, potentially making privileged operations callable by anyone.

Evidence:

BaseStrategy modifiers rely on external self-calls that can be shadowed:

```
// lib/tokenized-strategy/src/BaseStrategy.sol
modifier onlyManagement() {
 TokenizedStrategy.requireManagement(msg.sender);
 _;
}

modifier onlyKeepers() {
 TokenizedStrategy.requireKeeperOrManagement(msg.sender);
 _;
}

modifier onlyEmergencyAuthorized() {
 TokenizedStrategy.requireEmergencyAuthorized(msg.sender);
 _;
}
```

The intended checks live in the delegated implementation and use the passed `_sender`:

```
// lib/tokenized-strategy/src/TokenizedStrategy.sol
function requireManagement(address _sender) public view {
 require(_sender == _strategyStorage().management, "!management");
}
```

Impact:

If a strategy contract accidentally introduces a colliding function selector (e.g., by importing another base contract with a similarly named `requireManagement(address)`), then:

- Strategy functions guarded by `onlyManagement`/`onlyKeepers`/`onlyEmergencyAuthorized` may become callable by unauthorized users.
- Unauthorized callers could change critical strategy configuration (keeper/emergency admin/fees) or trigger emergency controls, depending on what the strategy exposes behind those modifiers.

Recommendation:

Avoid external self-calls for authorization checks that can be shadowed by derived contracts. Use internal checks in `BaseStrategy` that cannot be overridden by selector collisions, or ensure checks are performed via an unambiguous call path that cannot be intercepted by strategy-defined functions.

INFO

MEV sandwich opportunity around setProfitMaxUnlockTime(0) (instant burn of locked shares causes immediate PPS jump, flash-loanable arbitrage)

Locations:

`contracts/VaultV3.vy:1488-1519`

`contracts/VaultV3.vy:423-485`

`contracts/VaultV3.vy:1794-1870`

Description:

Calling `setProfitMaxUnlockTime(0)` burns **all vault-held shares** (`balance_of[self]`) and resets the unlocking schedule. Burning these shares reduces `total_supply` without reducing `total_assets`, causing an **instant increase in PPS**.

Because this parameter change is executed in a normal transaction, a searcher can MEV-sandwich it:

- Front-run: deposit at the pre-burn (lower) PPS.
- Back-run: redeem/withdraw immediately after the burn at the higher PPS.

This can be done with a flash loan (no need to hold the position over time) as long as the vault can service the post-burn withdrawal.

Evidence:

Instant burn in setter:

```
# contracts/VaultV3.vy
if (new_profit_max_unlock_time == 0):
 share_balance: uint256 = self.balance_of[self]
 if share_balance > 0:
 self._burn_shares(share_balance, self)
 self.profit_unlocking_rate = 0
 self.full_profit_unlock_date = 0
```

Share/asset conversions depend on `total_supply` and `_total_assets`:

```
amount = shares * self._total_assets() / self._total_supply()
shares = assets * self._total_supply() / self._total_assets()
```

Exploit sketch:

1) Vault currently has locked profit shares in `balance_of[self]`. 2) Governance/manager sends `setProfitMaxUnlockTime(0)` in the public mempool. 3) Attacker front-runs with a large `deposit()`. 4) Manager tx burns locked shares ' PPS increases instantly. 5) Attacker back-runs with `redeem()` to extract the proportional PPS jump and repay a flash loan.

Impact:

- Transfers a portion of the previously locked profit from existing holders to the sandwich attacker.
- Can be large if many shares are locked.
- Particularly dangerous because it is immediate, riskless, and potentially flash-loanable.

Recommendation:

Execute such PPS-shifting governance actions via private transactions/timelocks, or redesign the unlock/lock mechanics so parameter flips cannot create an instantaneous, sandwichable PPS discontinuity.

INFO

TokenizedStrategy strategy name is unbounded/unsanitized (initialize + setName), enabling storage bloat and phishing/confusable metadata

Locations:

`lib/tokenized-strategy/src/TokenizedStrategy.sol:433-474`

`lib/tokenized-strategy/src/TokenizedStrategy.sol:1620-1626`

Description:

`TokenizedStrategy.initialize` stores a user-supplied `_name` string in storage, and `setName(string)` allows management to replace it. Neither path enforces any maximum length or sanitizes characters.

This is an input-validation gap for user-visible metadata:

- Extremely large strings can cause **storage bloat** and make `name()` calls expensive (and potentially fail under low gas stipends/offchain tooling limits).
- Arbitrary bytes / confusable Unicode / control characters can be used for **phishing** and UI deception.

Evidence:

Initialization stores `_name` directly:

```
S.name = _name;
```

Management setter stores arbitrary-length calldata string:

```
function setName(string calldata _name) external onlyManagement {
 _strategyStorage().name = _name;
}
```

Impact:

Low:

- Primarily an operational / UX / ecosystem safety issue.
- Can degrade offchain infrastructure and enable deceptive token metadata.

Recommendation:

Enforce a reasonable maximum length (and optionally restrict to safe character sets) for strategy names in both `initialize` and `setName`.

INFO

VaultFactory constructor does not validate critical addresses (governance/vault_original can be zero)

Location:

`contracts/VaultFactory.vy:102-107`

Description:

The factory constructor accepts `vault_original` and `governance` without validating that they are non-zero addresses.

- `vault_original == empty(address)` would make `deploy_new_vault` deploy unusable proxies.
- `governance == empty(address)` permanently disables all governance-only functions.

Evidence:

```
# contracts/VaultFactory.vy:102-107
@external
def __init__(name: String[64], vault_original: address, governance: address):
 self.name = name
 VAULT_ORIGINAL = vault_original
 self.governance = governance
```

Impact:

Low. Misconfiguration risk at deployment.

Recommendation:

Add `assert vault_original != empty(address)` and `assert governance != empty(address)`.

INFO

VaultFactory custom fee setters accept zero/invalid vault addresses (misconfiguration foot-gun)

Location:

`contracts/VaultFactory.vy:289-325`

Description:

`set_custom_protocol_fee_bps()` and `remove_custom_protocol_fee()` do not validate the `vault` address parameter (e.g., non-zero, expected contract type, deployed vault/strategy).

While only governance can call these, the lack of validation makes it easy to accidentally configure fees for `empty(address)` or the wrong address, silently resulting in incorrect protocol-fee behavior.

Evidence:

```
@external
def set_custom_protocol_fee_bps(vault: address, new_custom_protocol_fee: uint16):
 ...
 self.custom_protocol_fee_data[vault] = ...

@external
def remove_custom_protocol_fee(vault: address):
 self.custom_protocol_fee_data[vault] = ...
```

Impact:

Low:

- Incorrect or missing protocol fee assessment for intended vaults/strategies due to typos.
- Potential operational confusion (fees appear set on-chain but not used by the intended reporter).

Recommendation:

Validate `vault != empty(address)` and (optionally) that it is a contract / expected reporter.

INFO

BaseStrategy `onlySelf` hook guard is bypassable via any external self-call entrypoint (not strictly “post-delegatecall” as documented)

Locations:

`lib/tokenized-strategy/src/BaseStrategy.sol:42-82`

`lib/tokenized-strategy/src/BaseStrategy.sol:375-440`

Description:

`BaseStrategy` documents `onlySelf` as ensuring hook functions are called “post a delegateCall from this address to the TokenizedStrategy”. In reality, the guard only enforces `msg.sender == address(this)`.

That condition is also satisfied by **any** external call that the strategy makes to itself (e.g., `address(this).call(...)` / `this.someHook(...)`). If a derived strategy (or an added helper) exposes *any* externally callable method that triggers a self-call to one of these hook functions, an unprivileged user can execute logic that was intended to be reachable only through TokenizedStrategy-controlled flows.

This is an access-control footgun: the correctness of the guard depends on the derived strategy never providing user-triggerable self-calls into these hooks.

Evidence:

Guard only checks caller is self:

```
function _onlySelf() internal view {
 require(msg.sender == address(this), "!self");
}
```

Hook functions protected only by `onlySelf`:

```
function harvestAndReport() external virtual onlySelf returns (uint256) { ... }
function shutdownWithdraw(uint256 _amount) external virtual onlySelf { ... }
```

Impact:

If a derived strategy accidentally (or intentionally) exposes a user-callable path that self-calls these hooks, users may be able to:

- trigger emergency-withdraw logic (`shutdownWithdraw`) without being emergency authorized,
- trigger reporting/tending internals out of band,
- generally bypass the intended “TokenizedStrategy is the sole orchestrator” assumption.

While this is not exploitable in BaseStrategy alone, it materially increases the risk of access-control mistakes in derived strategies and contradicts the stated guarantee.

Recommendation:

Document this limitation prominently and/or strengthen the guard to bind hook execution to TokenizedStrategy-controlled entrypoints (e.g., additional state/selector context checks), so accidental self-call exposure cannot bypass it.

INFO

Deposit-limit hook parameter semantics mismatch: TokenizedStrategy passes `receiver` but BaseStrategy docs imply depositor/caller

Locations:

`lib/tokenized-strategy/src/TokenizedStrategy.sol:869-892`

`lib/tokenized-strategy/src/BaseStrategy.sol:283-307`

Description:

`TokenizedStrategy` enforces deposit limits via `IBaseStrategy.availableDepositLimit(...)`, but passes the **ERC-4626 `receiver`** (the address receiving shares), not the `msg.sender` depositor.

While this matches ERC-4626's `maxDeposit(receiver)` semantics, the NatSpec in `BaseStrategy.availableDepositLimit` describes the parameter as "the address that is depositing into the strategy", which many developers interpret as the depositor/caller.

A strategist implementing a whitelist or per-user cap based on the perceived depositor can accidentally enforce limits on the wrong party. For example, a non-whitelisted depositor could deposit **to a whitelisted receiver** and bypass a "caller whitelist" that was mistakenly implemented using `availableDepositLimit`.

Evidence:

`TokenizedStrategy` forwards `receiver`:

```
return IBaseStrategy(address(this)).availableDepositLimit(receiver);
```

(`TokenizedStrategy.sol:870-878`)

But BaseStrategy docs describe the parameter as the depositor:

```
* @param . The address that is depositing into the strategy.
* @return . The available amount the `_owner` can deposit in terms of `asset`
function availableDepositLimit(address /*_owner*/) ...
```

(`BaseStrategy.sol:301-307`)

Impact:

Mis-implemented deposit limits/whitelists can be bypassed by depositing on behalf of a permitted receiver.

Recommendation:

Clarify the NatSpec to explicitly state that `availableDepositLimit` is queried with the ERC-4626 `receiver` address (share recipient), not necessarily the transaction caller.

INFO

ERC20 `totalSupply()` / `balanceOf(address(this))` are time-dependent due to virtual profit unlocking (no Transfer events) and comments are misleading

Locations:

```
lib/tokenized-strategy/src/TokenizedStrategy.sol:655-667
lib/tokenized-strategy/src/TokenizedStrategy.sol:829-835
lib/tokenized-strategy/src/TokenizedStrategy.sol:1660-1677
lib/tokenized-strategy/src/TokenizedStrategy.sol:1275-1289
```

Description:

The contract implements profit-locking by treating an increasing portion of the strategy's own share balance as "unlocked" over time. This is implemented **purely in view functions** by subtracting `_unlockedShares()` from:

- `totalSupply()` (via `_totalSupply()`)
- `balanceOf(address(this))`

As time passes, `totalSupply()` and `balanceOf(address(this))` change **without any state change and without emitting `Transfer` events**. This deviates from many ERC-20 integration assumptions (e.g., that balances only change when transfers/mints/burns occur and events are emitted).

Additionally, the `totalSupply()` NatSpec is internally inconsistent with the implementation:

- Comment says locked shares are "not counted … until they are unlocked", but the implementation subtracts only the **unlocked** portion, meaning locked shares are counted and supply decreases as they unlock.

Evidence:

`totalSupply()` documentation:

```
// Locked shares issued to the strategy from profits are not
// counted towards the full supply until they are unlocked.
//
// As more shares slowly unlock the totalSupply will decrease
// causing the PPS of the strategy to increase.
```

(`TokenizedStrategy.sol:655-662`)

Implementation subtracts unlocked shares in views:

```
function _totalSupply(StrategyData storage S) internal view returns (uint256) {
 return S.totalSupply - _unlockedShares(S);
}
```

(`TokenizedStrategy.sol:829-834`)

And for strategy's own balance:

```
if (account == address(this)) {
 return S.balances[account] - _unlockedShares(S);
}
```

(`TokenizedStrategy.sol:1674-1676`)

`_unlockedShares()` depends on time since `lastReport`:

```
unlocked = (S.profitUnlockingRate * (block.timestamp - S.lastReport)) / MAX_BPS_EXTENDED;
```

(`TokenizedStrategy.sol:1279-1284`)

Impact:

This can break or confuse:

- indexers and accounting systems that rely on ERC-20 `Transfer` events to track balance/supply changes
- integrations that assume `totalSupply()` is a static state value and not time-dependent
- reviewers/strategists relying on the NatSpec for correct mental models

Recommendation:

Clarify the NatSpec to match the actual mechanism, and consider documenting prominently that `totalSupply()` and `balanceOf(address(this))` are time-dependent view projections (virtual burn/unlock) rather than literal ERC-20 state.

INFO

IBaseStrategy section header claims 'IMMUTABLE FUNCTIONS' but includes mutable/external-effect hooks (comment/semantics mismatch)

Location:

`lib/tokenized-strategy/src/interfaces/IBaseStrategy.sol:7-29`

Description:

`IBaseStrategy` labels a section as `IMMUTABLE FUNCTIONS`, but the listed functions include state-changing hooks (`deployFunds`, `freeFunds`, `harvestAndReport`, `tendThis`, `shutdownWithdraw`) that are neither immutable nor view/pure.

This comment-to-ABI mismatch can mislead implementers/integrators doing semantic reviews or generating docs from the interface.

Evidence:

```
/*////////////////////////////////////////////////////////////////
 IMMUTABLE FUNCTIONS
////////////////////////////////////////////////////////////////*/

function deployFunds(uint256 _assets) external;
function freeFunds(uint256 _amount) external;
function harvestAndReport() external returns (uint256);
function tendThis(uint256 _totalIdle) external;
function shutdownWithdraw(uint256 _amount) external;
```

Impact:

Documentation/semantic confusion only.

Recommendation:

Rename the section header to something accurate (e.g., "strategy hooks" / "callback hooks"), or separate view vs state-changing functions into distinct sections.

INFO

Misleading/outdated comments can cause operator and integrator mistakes

Locations:

`contracts/VaultV3.vy:486-502`

`contracts/VaultV3.vy:669-690`

`contracts/VaultV3.vy:1409-1418`

Description:

Multiple comments describe semantics that do not match the current implementation, increasing the risk of operator/integrator misunderstanding.

Evidence:

Safe ERC20 helper comments contradict actual usage:

The helpers are documented as only for "tokens that are not the type managed by this Vault", but they are used for the managed `asset` (e.g., `_deposit`, `buy_debt`, refunds).

```
# Used only to approve/transfer tokens that are not the type managed by this Vault.
# Used to handle non-compliant tokens like USDT
```

(contracts/VaultV3.vy:486-502)

Unrealised loss formula comment is incorrect/misleading:

```
# Users will withdraw assets_needed divided by loss ratio (strategy_assets / strategy_current_debt - 1).
```

The implemented math is `assets_needed - (assets_needed * strategy_assets / strategy_current_debt)` (i.e., `assets_needed * (1 - strategy_assets/strategy_current_debt)`), not a division by `(ratio - 1)`. (contracts/VaultV3.vy:686-689)

set_deposit_limit natspec contradicts behavior during shutdown:

Docstring says the limit "can not be changed … unless … or if shutdown", but the code forbids calling when shutdown:

```
@external
def set_deposit_limit(...):
 ...
 assert self.shutdown == False
```

(contracts/VaultV3.vy:1411-1418)

Impact:

- Operators may misconfigure modules/roles based on incorrect expectations.
- Integrators may misunderstand loss math and accounting flows.

Recommendation:

Update comments/natspec to reflect actual behavior and clarify any intentional deviations.

INFO

No rescue/sweep mechanism for accidental ETH/ERC20 transfers in Factory/Vault/TokenizedStrategy (assets can become stuck forever)

Locations:

```
contracts/VaultFactory.vy:1-365
```

```
contracts/VaultV3.vy:1-2198
```

```
lib/tokenized-strategy/src/TokenizedStrategy.sol:74-2046
```

Description:

Multiple core contracts in scope have no explicit mechanism to recover assets mistakenly sent to them (either ERC20s that are not part of their normal flows, or ETH received via `SELFDESTRUCT`).

This creates one-way asset flows (black holes): tokens/ETH can enter, but cannot leave.

Evidence:

- `VaultFactory.vy` has no `payable`/`default` method and no governance-controlled token/ETH recovery function.
- `VaultV3.vy` only transfers out the configured `asset` during withdrawals and strategy-share tokens during `buy_debt`; it has no generic `sweep(token)`.
- `TokenizedStrategy.sol` similarly lacks any `sweep/recover` function for non-`asset` tokens held by the strategy.

Impact:

- Any ERC20s mistakenly transferred to these contracts (reward tokens, airdrops, user mistakes) are **stuck permanently**.
- ETH forcibly sent via `SELFDESTRUCT` is also stuck.

Recommendation:

Add an explicitly scoped recovery mechanism (e.g., governance-only sweep of non-`asset` tokens, and ETH withdrawal), carefully excluding the primary `asset` to avoid rug risk.

INFO

Protocol fee parameters are governance-controlled without timelock (centralization / sudden fee-change risk)

Location:

`contracts/VaultFactory.vy:235-286`

Description:

`VaultFactory` governance can change the default protocol fee bps and recipient immediately via `set_protocol_fee_bps` and `set_protocol_fee_recipient`, and can set per-address custom protocol fee bps via `set_custom_protocol_fee_bps`.

There is no timelock / delay mechanism, so fees can be increased up to `MAX_FEE_BPS` (50%) and applied instantly on the next report.

Impact:

Centralization risk: users cannot reliably react/exit before fee configuration changes take effect. This is especially relevant for permissionless deployments where users may assume stable fee policies.

Recommendation:

Consider adding an optional timelock/delay for fee changes (especially increases), or emitting scheduled-change events with an enforced activation time, so depositors can exit before the change applies.

INFO

TokenizedStrategy API_VERSION constant (3.0.3) mismatches Vault/VaultFactory API_VERSION (3.0.4), risking incorrect compatibility checks

Locations:

`lib/tokenized-strategy/src/TokenizedStrategy.sol:345-347`

`contracts/VaultFactory.vy:71-73`

`contracts/VaultV3.vy:178-180`

Description:

The repository's vault and factory identify themselves as API version `3.0.4`, but the bundled `TokenizedStrategy` implementation reports `API_VERSION = "3.0.3"`.

Because `BaseStrategy` delegates to the shared `TokenizedStrategy` implementation, off-chain systems (or integrators performing version gating) may treat strategies as a different API version than the vault/factory they are deployed alongside.

Evidence:

- `TokenizedStrategy.sol`:

```
string internal constant API_VERSION = "3.0.3";
```

- `VaultFactory.vy`:

```
API_VERSION: constant(String[28]) = "3.0.4"
```

- `VaultV3.vy`:

```
API_VERSION: constant(String[28]) = "3.0.4"
```

Impact:

Potential ecosystem-level incompatibility:

- indexers / frontends / scripts may refuse to interact with strategies based on a perceived version mismatch,

- monitoring that assumes strategy+vault share a common API version may misclassify deployments.

Recommendation:
Align version constants across components (or clearly document that vault and tokenized-strategy versions differ and should not be compared directly).

INFO

TokenizedStrategy ERC20 approve/permit inherits the known nonzero-to-nonzero allowance race condition

Locations:

`lib/tokenized-strategy/src/TokenizedStrategy.sol:1748-1751`

`lib/tokenized-strategy/src/TokenizedStrategy.sol:1887-1898`

`lib/tokenized-strategy/src/TokenizedStrategy.sol:1965-2009`

Description:

`TokenizedStrategy` implements ERC20-style `approve()` / `_approve()` by directly setting the allowance mapping to `amount`. This inherits the well-known ERC-20 allowance race: when changing an allowance from one non-zero value to another non-zero value, a spender can front-run and spend the old allowance before the new allowance is applied, potentially resulting in both allowances being usable.

The contract comments warn about this risk, but it is not mitigated in code (e.g., via `increaseAllowance`/`decreaseAllowance` patterns or enforcing zero-first updates).

`permit()` also calls `_approve()` directly, so the same race/UX hazard applies when using signatures.

Evidence:

- `approve()` calls `_approve(...)` directly:

```
function approve(address spender, uint256 amount) external returns (bool) {
 _approve(_strategyStorage(), msg.sender, spender, amount);
 return true;
}
```

- `_approve()` overwrites the allowance unconditionally:

```
S.allowances[owner][spender] = amount;
```

- `permit()` ultimately calls `_approve(...)`:

```
_approve(_strategyStorage(), recoveredAddress, spender, value);
```

Impact:

This is primarily an integration/user-safety issue: users and UIs that “change” allowances without first setting them to zero can be exposed to front-running/double-spend of allowances.

Recommendation:

Document this prominently for integrators/UIs and/or add `increaseAllowance` / `decreaseAllowance` helpers (or require zero-first changes) to reduce the practical risk.

INFO

VaultFactory CREATE2 salt excludes initial role_manager and profit_max_unlock_time (deployment collisions for alternative initial configs)

Location:

`contracts/VaultFactory.vy:108-144`

Description:

`deploy_new_vault` derives the CREATE2 salt from `(msg.sender, asset, name, symbol)` but does not include `role_manager` or `profit_max_unlock_time`. Two deployments differing only in those omitted parameters will collide to the same deterministic address, causing the later deployment to revert.

This is likely intentional (both parameters are mutable post-deploy), but it can surprise integrators expecting those parameters to influence the deterministic address.

Evidence:

```
# contracts/VaultFactory.vy:128-133
salt=keccak256(_abi_encode(msg.sender, asset, name, symbol))
```

Impact:

Informational / functional. Can cause unexpected deployment DoS for alternate initial configs with same (deployer, asset, name, symbol).

Recommendation:

Either include the omitted parameters in the salt, or document explicitly that deterministic address is independent of them.

INFO

VaultFactory protocol fee packing comment claims an 8-bit custom flag, but implementation uses only the lowest bit

Locations:

`contracts/VaultFactory.vy:94-96`

`contracts/VaultFactory.vy:219-234`

Description:

The factory documents the packed `protocol_fee_data` layout as including an **8-bit custom flag**, but the implementation packs a `bool` and checks only the **lowest bit**.

This comment-code mismatch can cause external implementers (off-chain decoders, other contracts, migrations) to pack data incorrectly (e.g., setting the flag to `2` expecting it to be truthy), which the factory will treat as `False`.

Evidence:

Comment:

```
# 72 bits Empty | 160 bits fee recipient | 16 bits fee bps | 8 bits custom flag
```

Implementation:

```
def _pack_protocol_fee_data(recipient: address, fee: uint16, custom: bool) -> uint256:
 return shift(convert(recipient, uint256), 24) | shift(convert(fee, uint256), 8) |
convert(custom, uint256)

def _unpack_custom_flag(config_data: uint256) -> bool:
 return config_data & 1 == 1
```

Impact:

Primarily an integration/correctness hazard: third parties relying on the documented layout may mis-handle the flag and misinterpret whether a custom fee is set.

Recommendation:

Update comments to reflect the actual encoding (1-bit flag, 7 reserved bits), or update `_unpack_custom_flag` to treat the whole low byte as the flag if that was the intent.

INFO

VaultV3 privileged operations execute immediately with no timelock (governance/role-manager centralization risk)

Location:

`contracts/VaultV3.vy:1327-1785`

Description:

VaultV3 includes many privileged operations (strategy add/revoke, debt updates, limit module changes, shutdown, role changes) that take effect immediately when called by authorized roles/role-manager. There is no built-in timelock/delay mechanism.

This is a governance/operational risk: if privileged keys are compromised, harmful configuration changes can be executed instantly, leaving users limited time to react.

Evidence:

Examples of immediate privileged functions include:

- `add_strategy`, `revoke_strategy`, `force_revoke_strategy`
- `update_debt`, `update_max_debt_for_strategy`
- `set_deposit_limit(_module)`, `set_withdraw_limit_module`
- `shutdown_vault`
- role assignments via `set_role`, `add_role`, `remove_role`

Impact:

Informational (design/trust model). Users must treat privileged roles as fully trusted.

Recommendation:

Consider adding a timelock layer (off-chain governance timelock, on-chain delay module) or explicitly documenting the trust assumptions for privileged roles.

INFO

`maxWithdraw`/`maxRedeem` NatSpec incorrectly says `owner` corresponds to `msg.sender` of `redeem`

Location:

`lib/tokenized-strategy/src/TokenizedStrategy.sol:770-805`

Description:

The NatSpec for `maxWithdraw(owner)` and `maxRedeem(owner)` states that `owner` "corresponds to the msg.sender of a {redeem} call". In ERC-4626, `owner` is the share owner whose balance is being redeemed/withdrawn, while `msg.sender` may be an approved third-party spender.

This is a semantic mismatch between documentation and ERC-4626 meaning and can mislead integrators/strategists.

Evidence:

```
/**
 * @notice Total number of underlying assets that can be
 * withdrawn from the strategy by `owner`, where `owner`
 * corresponds to the msg.sender of a {redeem} call.
 */
function maxWithdraw(address owner) external view returns (uint256) { ... }
...
/**
 * @notice Total number of strategy shares that can be
 * redeemed from the strategy by `owner`, where `owner`
 * corresponds to the msg.sender of a {redeem} call.
 */
function maxRedeem(address owner) external view returns (uint256) { ... }
```

(`TokenizedStrategy.sol:770-805`)

Impact:

Documentation mismatch; may lead to incorrect usage patterns and wrong mental models about allowance/spender vs owner.

Recommendation:

Update NatSpec to reflect ERC-4626 semantics: `owner` is the share owner, independent of `msg.sender`.

INFO

`tend()` NatSpec claims tendTrigger must be implemented, but `tend()` does not enforce it (can execute `_tend` even when `tendTrigger()` returns false)

Location:

`lib/tokenized-strategy/src/TokenizedStrategy.sol:1295-1319`

Description:

The NatSpec for `tend()` says it is “for a keeper to tend the strategy if a custom tendTrigger() is implemented” and that both `tendTrigger` and `_tend` must be overridden for it to be used.

However, `tend()` does **not** check `tendTrigger()` at all; it always callbacks into `BaseStrategy.tendThis(_totalIdle)`, which will execute `_tend(_totalIdle)` if the strategist has overridden `_tend`.

This is a semantic/documentation mismatch: a strategy can have meaningful `_tend` logic even if `tendTrigger()` remains the default `false`.

Evidence:

NatSpec:

```
/**
 * @notice For a 'keeper' to 'tend' the strategy if a custom
 * tendTrigger() is implemented.
 *
 * @dev Both 'tendTrigger' and '_tend' will need to be overridden
 * for this to be used.
 */
function tend() external nonReentrant onlyKeepers {
 IBaseStrategy(address(this)).tendThis(asset.balanceOf(address(this)));
}
```

(`TokenizedStrategy.sol:1295-1319`)

Impact:

Operational/automation confusion: off-chain keepers may rely on `tendTrigger()` and never call `tend()`, even though `_tend` would do useful work. Conversely, an authorized keeper/management can call `tend()` regardless of trigger state.

Recommendation:
Clarify NatSpec to state that `tendTrigger()` is only an advisory helper for off-chain automation and is not enforced on-chain.

INFO

maxLoss check can revert due to multiplication overflow for extremely large withdrawals

Location:

`lib/tokenized-strategy/src/TokenizedStrategy.sol:1026-1032`

Description:

The max-loss guard computes the acceptable loss with `(assets * maxLoss) / MAX_BPS` using checked arithmetic:

```
require(loss <= (assets * maxLoss) / MAX_BPS, "too much loss");
```

If `assets` is extremely large (near `type(uint256).max`) this multiplication can overflow and revert, making withdrawals fail even if the actual loss is within bounds.

Impact:

Informational: likely unreachable for typical ERC-20 supplies, but it is a numerical edge case that can cause unexpected reverts for extreme-value assets/strategies.

Recommendation:

Use `Math.mulDiv(assets, maxLoss, MAX_BPS)` to avoid intermediate overflow.

INFO

maxLoss parameter units (basis points) are undocumented in ITokenizedStrategy, increasing risk of incorrect integration usage

Locations:

`lib/tokenized-strategy/src/interfaces/ITokenizedStrategy.sol:60-73`

`lib/tokenized-strategy/src/TokenizedStrategy.sol:556-565`

`lib/tokenized-strategy/src/TokenizedStrategy.sol:605-615`

`lib/tokenized-strategy/src/TokenizedStrategy.sol:996-1035`

Description:

`ITokenizedStrategy` introduces non-standard ERC-4626 overloads `withdraw(..., maxLoss)` and `redeem(..., maxLoss)` but the interface provides **no unit/scale documentation** for `maxLoss`.

In the implementation, `maxLoss` is interpreted as **basis points (BPS)** (0–10_000) of the *requested* asset amount:

- `maxLoss <= 10_000` is required.
- The withdraw/redeem succeeds with a shortfall only if `loss <= requestedAssets * maxLoss / 10_000`.

Evidence:

Implementation explicitly treats `maxLoss` as BPS:

```
require(maxLoss <= MAX_BPS, "exceeds MAX_BPS"); // MAX_BPS = 10_000
...
require(loss <= (assets * maxLoss) / MAX_BPS, "too much loss");
```

But the interface only exposes:

```
function withdraw(uint256 assets, address receiver, address owner, uint256 maxLoss) external
returns (uint256);
function redeem(uint256 shares, address receiver, address owner, uint256 maxLoss) external
returns (uint256);
```

with no unit description.

Impact:

Integrators can easily pass `maxLoss` in the wrong units (e.g., a raw asset amount, a 1e18 fixed-point percentage, etc.), leading to:

- unexpected reverts (`maxLoss` > 10_000), or
- unintended permissiveness/strictness (e.g., `maxLoss=1` interpreted as 0.01% not 1%).

Recommendation:

Document `maxLoss` units explicitly in the interface (e.g., “basis points where 10_000 = 100% of requested assets”) and, ideally, include named constants in public docs for common values.

INFO

pricePerShare() may overflow/revert or return wrapped value for high-decimal assets (10**decimals)

Location:

`contracts/VaultV3.vy:1617-1625`

Description:

`pricePerShare()` computes a scaling factor as `10 ** decimals` and passes it to `_convert_to_assets`.

For non-standard ERC-20s with very large `decimals` (the interface returns `uint8`, so up to 255), `10 ** decimals` can exceed `2**256-1`.

Depending on Vyper 0.3.7 exponentiation semantics, this can:

- revert due to overflow checks, or
- silently wrap modulo `2**256`.

Either behavior makes `pricePerShare()` unusable and numerically incorrect for such assets.

Evidence:

```
return self._convert_to_assets(10 ** convert(self.decimals, uint256), Rounding.ROUND_DOWN)
```

(contracts/VaultV3.vy:1624)

Impact:

- `pricePerShare()` can revert or return nonsense for high-decimal assets, breaking integrations that rely on it.

Recommendation:

Clamp `decimals` to a safe maximum for `10**decimals` (or compute PPS using a safe exponentiation / precomputed scaling), or document supported decimal ranges.

INFO

profitUnlockingRate calculation can revert from checked overflow with extreme share balances (totalLockedShares * 1e12)

Location:

`lib/tokenized-strategy/src/TokenizedStrategy.sol:1230-1233`

Description:

`report()` computes `profitUnlockingRate` using a checked multiplication:

```
S.profitUnlockingRate = (totalLockedShares * MAX_BPS_EXTENDED) / newProfitLockingPeriod;
```

For extremely large `totalLockedShares`, `totalLockedShares * 1e12` can overflow and revert, DoSing `report()`.

While this is unlikely for typical ERC-20 magnitudes, the contract does not enforce any upper bound on share supply.

Impact:

Informational edge case: potential `report()` DoS for extreme-value deployments.

Recommendation:

Use `Math.mulDiv(totalLockedShares, MAX_BPS_EXTENDED, newProfitLockingPeriod)` or add explicit bounds to prevent overflow.

INFO

symbol() makes an unbounded external call to the underlying token's symbol(), which may revert or be gas-prohibitive

Location:

`lib/tokenized-strategy/src/TokenizedStrategy.sol:1645-1648`

Description:

`symbol()` calls the underlying asset's `symbol()` externally:

```
string(abi.encodePacked("ys", _strategyStorage().asset.symbol()))
```

`symbol()` is not guaranteed to be cheap or even present for all ERC-20-like tokens (some tokens revert, return very large strings, or implement nonstandard behavior). This can make the strategy's `symbol()` revert or be gas-prohibitive to call on-chain.

Impact:

- DoS/compatibility issue for on-chain integrations that query `symbol()`.
- Operational issues for indexers/UIs if metadata calls revert.

Recommendation:

Consider caching the asset symbol at initialization or providing a management-settable symbol override to avoid relying on external token metadata at runtime.

Advisory to Clients

Cecuro highly recommends scheduling a follow-up security assessment within six months of this report or immediately after any significant modifications to the codebase, whichever occurs first. This practice is essential for preserving the project's security posture and identifying potential vulnerabilities that may arise from code changes or updates.

Disclaimer

The smart contracts submitted for analysis have been reviewed using Cecuro.ai's AI security analysis system, applying industry-standard vulnerability detection patterns and best practices at the time of this report. The findings disclosed in this report reflect the AI system's assessment of the submitted source code.

This report contains no statements or warranties on the identification of all vulnerabilities or the overall security of the code. Any security review methodology may not detect all potential issues, particularly novel attack vectors, complex business logic flaws, or economic exploits. The report covers the code submitted at the specified version and may not be relevant after any modifications.

Do not consider this report as a final and sufficient assessment regarding the security, utility, or bug-free status of the code. We recommend complementing this analysis with additional independent audits and a public bug bounty program to strengthen the security posture of your smart contracts.

Smart contracts are deployed and executed on blockchain platforms. The platform, its programming language, compiler, and other related software may contain vulnerabilities beyond the scope of code-level analysis. Cecuro.ai cannot guarantee the explicit security of the analyzed smart contracts.

This report does not constitute financial, legal, or investment advice. It is not a recommendation to buy, sell, or invest in any token or project, nor an endorsement of any kind. Users should conduct their own independent due diligence.

cecuro

Agentic Smart Contract Auditing