

Tenbin Security Review

Auditors

Phaze, Lead Security Researcher

R0bert, Lead Security Researcher

Om Parikh, Security Researcher

RScodes, Associate Security Researcher

Report prepared by: Lucas Goiriz

February 2, 2026

Contents

1 About Spearbit **2**

2 Introduction **2**

3 Risk classification **2**
3.1 Impact . . . 2
3.2 Likelihood . . . 2
3.3 Action required for severity levels . . . 2

4 Executive Summary **3**
4.1 Scope . . . 3

5 Findings **5**
5.1 High Risk . . . 5
5.1.1 Redeem nonce mis-tracked enables replay . . . 5
5.1.2 Revenue accounting ignores losses . . . 5
5.1.3 Direct vault deposits incorrectly counted as revenue leading to liquidity drain . . . 7
5.2 Medium Risk . . . 8
5.2.1 Redemption price can stale while waiting for liquidity . . . 8
5.2.2 Revenue accounting drifts on outflows . . . 9
5.3 Low Risk . . . 9
5.3.1 Potential panic error in failed swap . . . 9
5.3.2 `LayerZeroMintBurnOFTAdapter` misreports `approvalRequired` . . . 10
5.3.3 Restricted operators can execute transferFrom on behalf of users . . . 10
5.3.4 Single oracle used for multiple stablecoin collateral types enables incorrect price validation . . . 11
5.3.5 New reward distributions extend vesting period and delay previously scheduled rewards . . . 12
5.3.6 Cooldown period changes do not affect pending cooldowns . . . 13
5.3.7 Slippage check assumes 1:1 price ratio between source and destination tokens . . . 14
5.3.8 Vesting period duration can't be changed when there is a current vesting . . . 15
5.3.9 Yield of last withdrawal may be lost . . . 16
5.3.10 Revenue withdrawal fails to redeem vault shares causing accounting mismatch . . . 16
5.3.11 Slippage check can underflow when `minReturnAmount > expectedAmountOut` . . . 17
5.3.12 1inch swap flags in payload are not validated . . . 18
5.3.13 DoS in `CollateralManager` deposit & withdraw due to market and capacity constraints . . . 19
5.3.14 Missing bad-debt protections in morpho vault-v2 deposit for metamorpho v1.1 adapters . . . 20
5.4 Gas Optimization . . . 21
5.4.1 Using `safeIncreaseAllowance` instead of forceApprove adds unnecessary checks and complexity . . . 21
5.5 Informational . . . 21
5.5.1 Chainlink price freshness unchecked . . . 21
5.5.2 EIP-712 typehash mismatch in deploy script . . . 22
5.5.3 Controller update relies on caller-supplied list . . . 22
5.5.4 Controller constructor skips ratio bound . . . 22
5.5.5 Slippage guard floors bps tolerance . . . 23
5.5.6 ERC4626 first-depositor inflation risk mitigation relies on OZ virtual shares . . . 23
5.5.7 SwapModule TODO is already implemented . . . 24
5.5.8 Missing nonce invalidation function prevents cancellation of accidental orders . . . 24
5.5.9 Systemic economic risks . . . 24
5.5.10 Oracle delta tolerance measured against order price allows extreme price deviations . . . 26
5.5.11 Unsafe downcasting in cooldown assets could lead to truncation . . . 27
5.5.12 Suggestion for a better way to calculate the new cooldown endtime . . . 27
5.5.13 Code quality improvements . . . 28
5.5.14 Whitelist 1inch executors in `SwapModule.swap1inch` . . . 29
5.5.15 Incorrect morpho integration setup in test-case . . . 29

1 About Spearbit

Spearbit is a decentralized network of expert security engineers offering reviews and other security related services to Web3 projects with the goal of creating a stronger ecosystem. Our network has experience on every part of the blockchain technology stack, including but not limited to protocol design, smart contracts and the Solidity compiler. Spearbit brings in untapped security talent by enabling expert freelance auditors seeking flexibility to work on interesting projects together.

Learn more about us at spearbit.com

2 Introduction

Disclaimer: This security review does not guarantee against a hack. It is a snapshot in time of Tenbin according to the specific commit. Any modifications to the code will require a new security review.

3 Risk classification

Severity level	**Impact: High**	**Impact: Medium**	**Impact: Low**
Likelihood: high	Critical	High	Medium
Likelihood: medium	High	Medium	Low
Likelihood: low	Medium	Low	Low

3.1 Impact

- High - leads to a loss of a significant portion (>10%) of assets in the protocol, or significant harm to a majority of users.
- Medium - global losses <10% or losses to only a subset of users, but still unacceptable.
- Low - losses will be annoying but bearable--applies to things like griefing attacks that can be easily repaired or even gas inefficiencies.

3.2 Likelihood

- High - almost certain to happen, easy to perform, or not easy but highly incentivized
- Medium - only conditionally possible or incentivized, but still relatively likely
- Low - requires stars to align, or little-to-no incentive

3.3 Action required for severity levels

- Critical - Must fix as soon as possible (if already deployed)
- High - Must fix (before deployment if not already deployed)
- Medium - Should fix
- Low - Could fix

4 Executive Summary

Over the course of 4 days in total, Tenbin engaged with Spearbit to review the tenbin-monorepo protocol. In this period of time a total of **35** issues were found.

Summary

Project Name	Tenbin
Repository	tenbin-monorepo
Commit	6cb95faf
Type of Project	RWA, Synthetics
Audit Timeline	Dec 8th to Dec 12th

Issues Found

Severity	Count	Fixed	Acknowledged
Critical Risk	0	0	0
High Risk	3	3	0
Medium Risk	2	0	2
Low Risk	14	8	6
Gas Optimizations	1	1	0
Informational	15	11	4
Total	**35**	**23**	**12**

4.1 Scope

The security review had the following components in scope for tenbin-monorepo on commit hash 6cb95faf:

```
packages/contracts/src
├── adapters
│   ├── LayerZeroMintBurnOFTAdapter.sol
│   ├── LayerZeroOFTAdapter.sol
│   └── LayerZeroOVaultComposer.sol
├── AssetSilo.sol
├── AssetToken.sol
├── CollateralManager.sol
├── Controller.sol
├── CustodianModule.sol
├── external
│   ├── 1inch
│   │   └── IAggregationRouterV6.sol
│   └── chainlink
│       └── AggregatorInterface.sol
├── interface
│   ├── ICollateralManager.sol
│   ├── IController.sol
```

```
│   ├── IMintableBurnable.sol
│   ├── IOracleAdapter.sol
│   ├── IRestrictedRegistry.sol
│   ├── IRevenueModule.sol
│   ├── IStakedAsset.sol
│   └── ISwapModule.sol
├── MultiCall.sol
├── OracleAdapter.sol
├── RevenueModule.sol
├── StakedAsset.sol
└── SwapModule.sol
```

5 Findings

5.1 High Risk

5.1.1 Redeem nonce mis-tracked enables replay

Severity: High Risk

Context: Controller.sol#L306-L307

Description: The redeem flow validates `_verifyNonce(order.payer, order.nonce)` but then records usage as `nonces[signer][order.nonce] = true`. When a payer delegates a signer, the signer and payer differ, so the payer's nonce is never consumed. A delegated redeem signature can be replayed endlessly, each time passing nonce checks and executing the transfer and burn. Example:

```
function redeem(Order calldata order, Signature calldata signature) external
↪  onlyRole(MINTER_ROLE) {
    (address signer,) = verifyOrder(order, signature); // checks payer nonce
    nonces[signer][order.nonce] = true;                // records signer nonce
    IERC20(order.collateral_token).safeTransferFrom(manager, order.recipient,
    ↪  order.collateral_amount);
    AssetToken(asset).burn(order.payer, order.asset_amount);
}
```

Impact: a malicious relayer or delegated signer can loop the same redeem order to drain collateral from `manager` and repeatedly burn the payer's assets until balances or allowances are exhausted.

Recommendation: Consume the nonce against the payer to match the verification key, mirroring `mint`, e.g. `nonces[order.payer][order.nonce] = true;` before external calls, or otherwise align nonce checks and storage so a redeem signature cannot be replayed.

Tenbin: Fixed in commit d04a30df.

Spearbit: Fix verified. That commit changes `redeem` to mark `nonces[order.payer][order.nonce] = true` in `Controller.sol` aligning storage with the payer-based nonce check.

5.1.2 Revenue accounting ignores losses

Severity: High Risk

Context: CollateralManager.sol#L374-L386

Description: The `CollateralManager` accounts for ”revenue” in a way that only ever accumulates positive yield and never offsets it when the underlying vault loses value. This allows `withdrawRevenue` to pay out amounts labeled as revenue that exceed the vault's net performance, withdrawing principal as ”revenue”.

Revenue-related state:

```
mapping(address => uint256) public pendingRevenue;
mapping(address => uint256) public lastTotalAssets;
```

New revenue is computed and realized as follows:

```
function _computeNewRevenue(address collateral, IERC4626 vault)
    internal
    view
    returns (uint256 revenue)
{
    uint256 totalAssets = vault.totalAssets();
    uint256 lastTotal = lastTotalAssets[collateral];
    if (totalAssets > lastTotal) {
        unchecked {
            revenue = totalAssets - lastTotal;
        }
```

```
    }
}

function _realizeRevenue(address collateral, IERC4626 vault) internal {
    uint256 newRevenue = _computeNewRevenue(collateral, vault);
    if (newRevenue > 0) pendingRevenue[collateral] += newRevenue;
}
```

When revenue is withdrawn:

```
function withdrawRevenue(address collateral, uint256 amount)
    external
    nonReentrant
    notPaused
    onlyRole(COLLECTOR_ROLE)
{
    IERC4626 vault = vaults[collateral];
    if (address(vault) == address(0)) revert CollateralNotSupported();

    _realizeRevenue(collateral, vault);
    uint256 totalRevenue = pendingRevenue[collateral];
    if (amount > totalRevenue) revert ExceedsPendingRevenue();

    IERC20(collateral).safeTransfer(msg.sender, amount);

    unchecked {
        pendingRevenue[collateral] = totalRevenue - amount;
    }
    lastTotalAssets[collateral] = vault.totalAssets();
    emit RevenueWithdraw(collateral, amount);
}
```

The key behavior is that `_computeNewRevenue` only returns a positive value when `totalAssets > lastTotal`. If `totalAssets < lastTotal` (the vault has suffered a loss), `_computeNewRevenue` returns 0 and `_realizeRevenue` makes no adjustment. As a result, `pendingRevenue` is never reduced when the strategy loses funds.

A concrete scenario shows the problem:

1. Initially, `lastTotalAssets[collateral] = 100` and `pendingRevenue[collateral] = 0`.
2. The underlying vault appreciates to `totalAssets = 120`. On the next management action, `_realizeRevenue` observes `120 > 100` and adds 20 to `pendingRevenue`. `lastTotalAssets` is updated to 120. At this point, net yield is +20, and `pendingRevenue = 20` is consistent.
3. Later, the strategy loses 15; `vault.totalAssets()` drops to 105 while `lastTotalAssets[collateral]` remains 120. `pendingRevenue[collateral]` is still 20; the previously booked revenue is not reduced, even though net yield is now only +5.
4. A call to `withdrawRevenue(collateral, 20)` triggers `_realizeRevenue`, which now sees `totalAssets = 105` and `lastTotal = 120`. Since `totalAssets` is lower, `_computeNewRevenue` returns 0 and `pendingRevenue` remains 20. The `amount <= pendingRevenue` check passes and the function transfers 20 units of collateral from the manager's balance to the collector, then zeroes out `pendingRevenue` and updates `lastTotalAssets` to 105.

In this sequence, the system has only generated net yield of 5, yet it pays out 20 as ”revenue”. The extra 15 must come from collateral that should economically be considered principal backing the asset tokens. This contradicts the intended separation of ”collateral backing” versus ”revenue” described in the documentation and can lead to over-distribution of yield, under-collateralizing the protocol from the perspective of asset holders.

In short, `pendingRevenue` tracks the sum of positive increments in `totalAssets` and ignores negative increments. After losses, it can reflect a number larger than the vault's net performance, enabling revenue withdrawals that eat into principal.

Recommendation: Change the revenue accounting so that `pendingRevenue` represents net, loss-adjusted yield rather than a sum of only positive jumps. When `totalAssets` decreases below `lastTotalAssets[collateral]`, the loss portion should first reduce `pendingRevenue[collateral]` before it is implicitly borne by principal.

One straightforward approach is to fold loss handling into `_realizeRevenue`, for example:

```
function _realizeRevenue(address collateral, IERC4626 vault) internal {
    uint256 totalAssets = vault.totalAssets();
    uint256 lastTotal = lastTotalAssets[collateral];

    if (totalAssets > lastTotal) {
        uint256 gain = totalAssets - lastTotal;
        pendingRevenue[collateral] += gain;
    } else if (totalAssets < lastTotal) {
        uint256 loss = lastTotal - totalAssets;
        uint256 rev = pendingRevenue[collateral];
        pendingRevenue[collateral] = loss >= rev ? 0 : rev - loss;
    }

    lastTotalAssets[collateral] = totalAssets;
}
```

The view function `getRevenue` should mirror this net logic when computing the current pending revenue without changing state. This ensures that, after a strategy loss, the revenue bucket is reduced (and possibly zeroed) before any further revenue withdrawals, preventing principal from being mislabeled and paid out as revenue.

Tenbin: Fixed in commit d04a30df.

Spearbit: Fix verified. This commit updates `_getRevenue` to subtract losses from `pendingRevenue` in `CollateralManager.sol`.

5.1.3 Direct vault deposits incorrectly counted as revenue leading to liquidity drain

Severity: High Risk

Context: CollateralManager.sol#L378-L386

Summary: The CollateralManager's revenue calculation mechanism incorrectly accounts for direct deposits to underlying vaults as protocol revenue. This occurs because the `_computeNewRevenue()` function calculates revenue by comparing the vault's total assets between two time periods, without distinguishing between yield generation and third-party deposits. This leads to overestimation of revenue and excessive withdrawal of assets from the protocol, potentially impacting liquidity.

Description: The `_computeNewRevenue()` function in the CollateralManager contract calculates revenue by comparing the current vault's total assets against the last recorded total assets:

```
function _computeNewRevenue(address collateral, IERC4626 vault) internal view returns (uint256
↪  revenue) {
    uint256 totalAssets = vault.totalAssets();
    uint256 lastTotal = lastTotalAssets[collateral];
    if (totalAssets > lastTotal) {
        unchecked {
            revenue = totalAssets - lastTotal;
        }
    }
}
```

This calculation assumes that any increase in the vault's total assets represents yield or revenue generated by the protocol. However, ERC4626 vaults typically allow direct deposits from any address. When external parties deposit assets directly to the vault, these deposits increase the vault's total assets and are consequently misclassified as protocol revenue.

The correct approach would be to track the protocol's share balance in the vault and only account for changes in the asset value of those shares as revenue, effectively isolating protocol-owned assets from third-party deposits.

Impact Explanation: The impact is medium to high as this issue leads to systematic overestimation of protocol revenue. This causes the protocol to withdraw more assets than it should as revenue, reducing the liquidity available for core protocol operations and user withdrawals. In scenarios with significant direct deposits, this could materially impact the protocol's ability to honor withdrawal requests.

Likelihood Explanation: The likelihood is medium to high. ERC4626 vaults inherently support direct deposits through their `deposit()` and `mint()` functions, which are callable by any address. While the frequency of direct deposits depends on whether users discover and utilize this capability, the technical possibility exists by default in standard vault implementations.

Recommendation: Consider tracking the protocol's share balance by querying the vault's `balanceOf()` function rather than relying on total assets. This approach isolates protocol-owned assets from third-party deposits:

```
  function _computeNewRevenue(address collateral, IERC4626 vault) internal view returns
  ↪  (uint256 revenue) {
-     uint256 totalAssets = vault.totalAssets();
+     uint256 shares = vault.balanceOf(address(this));
+     uint256 currentAssetValue = vault.previewRedeem(shares);
      uint256 lastTotal = lastTotalAssets[collateral];
-     if (totalAssets > lastTotal) {
+     if (currentAssetValue > lastTotal) {
          unchecked {
-             revenue = totalAssets - lastTotal;
+             revenue = currentAssetValue - lastTotal;
          }
      }
  }
```

This ensures that only changes in the value of protocol-owned shares are considered as revenue, preventing third-party deposits from affecting revenue calculations.

Tenbin: Fixed in commit d04a30df.

Spearbit: Fix verified.

5.2 Medium Risk

5.2.1 Redemption price can stale while waiting for liquidity

Severity: Medium Risk

Context: Controller.sol#L331-L380

Description: Redemption orders fix `collateral_amount` and `asset_amount` when signed, but execution is synchronous and requires the `CollateralManager` to have idle collateral. If funds are locked in vault queues, cooldowns or off-custody, curators must unwind first. That delay means the oracle price can move or the order can approach expiry while the caller waits. When liquidity finally arrives, the contract still executes at the originally signed price (subject only to the oracle tolerance at execution), so the executed price may be stale relative to the true exit cost of unwinding strategies or to the payer's original intent. If price moves beyond tolerance, the order simply reverts, creating liveness risk for large redemptions that need time to pull funds. If tolerance is widened to avoid reverts, the protocol can be forced to honor an outdated price after markets move. This mismatch between off-chain signing time and on-chain execution time under liquidity constraints leads either to stuck redemptions or execution at stale economics.

Recommendation: Permit redemptions to execute with positive slippage and partial fills to improve liveness when liquidity must be pulled. Allow a redeem to settle even if the oracle has moved in the user's favor (recipient accepts more collateral than originally quoted) while still blocking worse-than-quoted fills; this avoids reverts when price drifts during unwind. Add a partial-fill path that transfers the immediately available collateral, consumes the nonce

and requires a fresh order for the remainder once more liquidity is ready. Consider combining this with a minimum liquid buffer per collateral so routine redemptions clear without delay.

Tenbin: Acknowledged.

Spearbit: Acknowledged. The current implementation gives the payer a way to cancel and re-sign, but it doesn't change the `redeem` execution mechanics (fixed amounts, symmetric oracle delta, no partial fill/positive slippage), so the stale-price/liveness issue still exists just with a manual escape hatch.

5.2.2 Revenue accounting drifts on outflows

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

Description: The collateral manager models revenue as the cumulative change in an `ERC4626` vault's asset value since a stored snapshot and then subtracts that revenue from the total assets it reports. The view path uses this logic:

```
uint256 revenue = _getRevenue(collateral, vault);
assets = _totalAssets(vault) + IERC20(collateral).balanceOf(address(this)) - revenue;
```

However, pending revenue is only updated when the manager itself performs a `deposit`, `withdraw`, `withdrawRevenue` or `convertRevenue`. Collateral can leave the manager through redemption (the controller transfers from the manager), rebalancing, and swaps, and those flows do not refresh pending revenue or `lastTotalAssets`. Because collateral is fungible, these outflows can consume tokens that the accounting model still treats as revenue. This can make the computed revenue exceed the manager's actual holdings, which causes the assets calculation to underflow and revert, and can also cause revenue collection to fail after a prior read indicated revenue was available.

The result is a drift between accounting and reality that can break keeper automation and makes it easy for privileged operational flows to implicitly spend revenue that is intended to be separated from backing collateral. This creates a denial of service risk on accounting reads and revenue withdrawal flows and undermines the intended revenue separation model.

Recommendation: Decide whether revenue must be protected or can be spent, then enforce that policy on-chain. If revenue should be protected, route outbound flows through the manager or add a pre-transfer hook that refreshes revenue and enforces a cap such as `pendingRevenue <= totalAssets + onHand - amountLeaving`. If revenue can be spent, clamp revenue to available holdings in view logic to avoid underflow and to reflect the actual balance, and consider a sync function that explicitly snapshots revenue around large outflows.

Tenbin: Acknowledged.

Spearbit: Acknowledged. In commit 1ad0686 `getAssets` was replaced by `getVaultAssets` and now returns vault-only assets which removes the underflow risk, but does not address the revenue drift mentioned for `pendingRevenue` and `withdrawRevenue`. Outflows (redeem/rebalance/swap) can still consume balances that revenue accounting assumes are available, so revenue collection can still fail. Setting it as acknowledged for now unless the revenue drift issue is solved

5.3 Low Risk

5.3.1 Potential panic error in failed swap

Severity: Low Risk

Context: CollateralManager.sol#L338

Description: `swap` enforces caps on in/out amounts, then decrements `swapCap[dstToken]` by the actual amount received (`balanceAfter - balanceBefore`). If the received amount exceeds the stored cap, the subtraction underflows and triggers a panic revert after the swap call. The transaction reverts, but instead of a clear "cap exceeded" error, callers see an underflow panic, making positive-slippage swaps fail with a confusing error and DoS'ing swaps when caps are too low.

Recommendation: Before decrementing, explicitly check `received <= dstTokenCap` and revert with a clear cap error if exceeded.

Tenbin: Fixed in commit d04a30df.

Spearbit: Fix verified. The commit adds the explicit

```
if (actualOut > dstTokenCap) revert SwapCapExceededDst();
```

guard in `CollateralManager.sol`, preventing the underflow panic.

5.3.2 `LayerZeroMintBurnOFTAdapter` misreports `approvalRequired`

Severity: Low Risk

Context: LayerZeroMintBurnOFTAdapter.sol#L13

Description: `LayerZeroMintBurnOFTAdapter` inherits `MintBurnOFTAdapter`, whose `approvalRequired()` returns `false`, implying users don't need to approve the adapter. In this deployment `minterBurner` is `AssetToken` as defined in the test file(`test/integration/LayerZeroIntegration.t.sol`) and `_debit` calls `minterBurner.burn(_from, amountSentLD)`, which enforces `_spendAllowance(_from, msg.sender, _amount)`. Users who follow the flag and skip approvals will have cross-chain sends revert.

Recommendation: Consider setting `approvalRequired()` to `true` (override) as the underlying burn will require allowance. Consider also documenting the approval requirement in user-facing flows.

Tenbin: Fixed in commit d04a30df.

Spearbit: Fix verified. Issue was addressed by implementing the suggested recommendation.

5.3.3 Restricted operators can execute transferFrom on behalf of users

Severity: Low Risk

Context: StakedAsset.sol#L294-L302

Description: The `transferFrom()` function in the StakedAsset contract applies the `nonRestricted` modifier only to the `from` and `to` addresses, but not to the operator (`msg.sender`) executing the transfer:

```
function transferFrom(address from, address to, uint256 value)
    public
    override(IERC20, ERC20Upgradeable)
    nonRestricted(from)
    nonRestricted(to)
    returns (bool)
{
    return super.transferFrom(from, to, value);
}
```

This design allows a restricted operator who previously received approval from a user to continue moving that user's tokens through `transferFrom()`, even after being added to the restriction list. While the transfers themselves are still subject to the `from` and `to` restrictions, the operator executing the transfer is not checked.

In a scenario where malicious contracts obtain user approvals (such as through phishing campaigns) and are later identified and restricted, they could still drain approved user tokens by calling `transferFrom()` with unrestricted destination addresses.

Recommendation: Consider adding the `nonRestricted` modifier to check the operator (`msg.sender`) as well:

```
  function transferFrom(address from, address to, uint256 value)
      public
      override(IERC20, ERC20Upgradeable)
      nonRestricted(from)
      nonRestricted(to)
```

```
+     nonRestricted(msg.sender)
      returns (bool)
  {
      return super.transferFrom(from, to, value);
  }
```

This provides an additional layer of protection by preventing restricted operators from executing transfers on behalf of users, even when they have existing approvals. Users would need to be aware that restricting an address will prevent it from using any approvals it holds.

Tenbin: Fixed in commit d04a30df.

Spearbit: Fix verified.

5.3.4 Single oracle used for multiple stablecoin collateral types enables incorrect price validation

Severity: Low Risk

Context: Controller.sol#L363-L379

Description: The Controller contract uses a single oracle configuration for all collateral types, even though the protocol accepts multiple collateral assets. This design flaw prevents accurate price validation because different stablecoins can have different market prices, particularly during depeg events. Each stablecoin requires its own oracle feed to ensure proper price validation.

The Controller's order validation logic references a single oracle state variable for all price checks:

```
// Calculate price and revert if delta exceeds tolerance
Oracle memory oracleData = oracle;
if (oracle.adapter != address(0)) {
    uint256 oraclePrice = IOracleAdapter(oracle.adapter).getPrice();
    // normalize collateral amount to 18 decimals
    uint256 decimals = IERC20Metadata(order.collateral_token).decimals();
    uint256 collateralAmount;
    if (decimals == 18) collateralAmount = order.collateral_amount;
    else collateralAmount = order.collateral_amount * 10 ** (18 - decimals);
    // calculate price delta and revert if it exceeds the oracle tolerance
    uint256 price = Math.mulDiv(collateralAmount, 1e18, order.asset_amount);
    uint256 difference = price >= oraclePrice ? price - oraclePrice : oraclePrice - price;
    uint256 delta = Math.mulDiv(difference, 1e18, price);
    if (delta > oracleData.tolerance) revert ExceedsOracleDeltaTolerance();
}
```

While the protocol may intend to only accept stablecoin collateral, different stablecoins are not fungible and can trade at different prices:

- USDC might trade at $1.00.
- USDT might trade at $0.998.
- DAI might trade at $1.002.
- During depeg events, these differences can be much more pronounced.

Using a single oracle means that orders using different stablecoins will be validated against the same price, leading to:

1. USDT orders validated against USDC prices (or vice versa) producing incorrect delta calculations.
2. Valid orders being rejected when the actual collateral price differs from the configured oracle price.
3. Invalid orders being accepted when price deviations happen to fall within tolerance due to comparing against the wrong oracle.

This is particularly critical during depeg events where stablecoin prices can diverge significantly. For example, during the USDC depeg in March 2023, USDC traded as low as $0.88 while other stablecoins remained closer to $1.00.

Recommendation: Consider implementing a mapping that associates each collateral token with its specific oracle configuration:

```
- Oracle public oracle;
+ mapping(address => Oracle) public collateralOracles;

  // Calculate price and revert if delta exceeds tolerance
- Oracle memory oracleData = oracle;
+ Oracle memory oracleData = collateralOracles[order.collateral_token];
  if (oracleData.adapter != address(0)) {
-     uint256 oraclePrice = IOracleAdapter(oracle.adapter).getPrice();
+     uint256 oraclePrice = IOracleAdapter(oracleData.adapter).getPrice();
      // ... rest of validation logic
  }
```

Additionally, add a function to configure oracles per collateral:

```
function setCollateralOracle(
    address collateral,
    address adapter,
    uint256 tolerance
) external onlyRole(DEFAULT_ADMIN_ROLE) {
    collateralOracles[collateral] = Oracle({
        adapter: adapter,
        tolerance: tolerance
    });
    emit CollateralOracleSet(collateral, adapter, tolerance);
}
```

This ensures each stablecoin uses its appropriate price oracle for accurate validation, properly accounting for price deviations between different stablecoins.

Tenbin: Acknowledged.

Spearbit: Acknowledged.

5.3.5 New reward distributions extend vesting period and delay previously scheduled rewards

Severity: Low Risk

Context: StakedAsset.sol#L121-L131

Description: The `reward()` function in the StakedAsset contract extends the vesting end time whenever new rewards are added, which delays the distribution of previously scheduled rewards:

```
function reward(uint256 assets) external onlyRole(REWARDER_ROLE) {
    if (vesting.period > 0) {
        uint256 pending = _pendingRewards();
        vesting.assets = pending + assets;
        vesting.end = uint128(block.timestamp) + vesting.period;
        emit VestingStarted(pending + assets, uint128(block.timestamp) + vesting.period);
    }
    IERC20(asset()).safeTransferFrom(msg.sender, address(this), assets);
    emit RewardsReceived(assets);
}
```

The function combines pending (unvested) rewards with newly added rewards and resets the vesting end time to `block.timestamp + vesting.period`. This creates a scenario where rewards that were close to completing

their vesting schedule are effectively ”reset” to vest over a new full period.

Consider this example with a 100-day vesting period:

- T = 0: 100 assets added, expected to fully vest at t = 100.
- T = 50: 1 new asset added via `reward()`.
  - Pending rewards: 50 assets (half vested).
  - New vesting.assets: 51 (50 pending + 1 new).
  - New vesting.end: 150 (50 + 100).
- T = 100 (original expected completion):
  - Vested amount: ~34 assets (51 × 50/100).
  - Expected amount: 100 assets.
  - Shortfall: ~66 assets delayed until t = 150.

This means that calling `reward()` frequently can indefinitely delay full reward distribution, as each call extends the vesting period by the full duration rather than only vesting the new rewards. The rewarder is also able to extend the period while adding 0 assets.

Recommendation: Consider one of the following approaches:

- Option 1: If this behavior is intentional, document it clearly in the code to ensure operators understand that adding new rewards will extend the vesting timeline for all pending rewards:

```
/// @notice Adds new rewards to the contract and extends vesting period
/// @dev WARNING: This resets the vesting end time to block.timestamp + vesting.period,
///      which can delay distribution of previously pending rewards
/// @param assets Amount of reward assets to add
function reward(uint256 assets) external onlyRole(REWARDER_ROLE) {
    // ... existing implementation
}
```

  Additionally, a check could be added enforcing that the previous vesting has ended.

- Option 2: Adjust the vesting end time proportionally to avoid resetting the schedule for existing rewards.
- Option 3: Maintain separate vesting schedules for each reward batch, though this adds implementation complexity and requires tracking multiple schedules.

Tenbin: Acknowledged.

Spearbit: Acknowledged.

5.3.6 Cooldown period changes do not affect pending cooldowns

Severity: Low Risk

Context: StakedAsset.sol#L179-L186

Description: The `setCooldownPeriod()` function in the StakedAsset contract allows administrators to update the cooldown period:

```
function setCooldownPeriod(uint256 newCooldownPeriod) external onlyRole(ADMIN_ROLE) {
    if (newCooldownPeriod > MAX_COOLDOWN_PERIOD) revert ExceedsMaxCooldownPeriod();
    cooldownPeriod = newCooldownPeriod;
    emit CooldownPeriodUpdated(newCooldownPeriod);
}
```

However, this change only affects new cooldowns initiated after the update. Users who have already started their cooldown process are not impacted because the contract stores the cooldown end time rather than the start time. Once a cooldown is initiated, the end time is calculated as `block.timestamp + cooldownPeriod` and stored, making it immutable regardless of subsequent changes to the cooldown period.

This creates issues during crisis scenarios such as security incidents or bank runs. If the protocol detects a vulnerability or faces a liquidity crisis and wants to allow users to exit faster by reducing the cooldown period, users who already started their cooldown remain locked into the longer original period. This creates:

- Unfairness: Early responders who initiated cooldown before the reduction must wait longer than those who start after.
- Panic dynamics: Users may race to cancel and restart their cooldown to take advantage of the shorter period, adding complexity during already stressful situations.
- Reduced effectiveness: Emergency measures to expedite withdrawals don't apply to potentially the largest group of users (those already in cooldown).

Similarly, if the cooldown period is increased for security reasons after a hack is detected, users with pending cooldowns can complete their withdrawals earlier than intended, potentially allowing attackers to exit before the increased restrictions take effect.

Recommendation: If the protocol intends for cooldown period changes to apply to all pending cooldowns, consider storing the cooldown start time instead of the end time and calculating the end time dynamically:

```
- mapping(address => uint256) public cooldownEnd;
+ mapping(address => uint256) public cooldownStart;

  function startCooldown() external {
      // ... existing checks
-     cooldownEnd[msg.sender] = block.timestamp + cooldownPeriod;
+     cooldownStart[msg.sender] = block.timestamp;
      emit CooldownStarted(msg.sender);
  }

  function _cooldownComplete(address account) internal view returns (bool) {
-     return block.timestamp >= cooldownEnd[account];
+     return cooldownStart[account] > 0 &&
+            block.timestamp >= cooldownStart[account] + cooldownPeriod;
  }
```

This allows cooldown period changes to apply uniformly to all users, ensuring emergency measures and policy updates affect everyone consistently. Alternatively, if the current behavior is intentional (pending cooldowns should not be affected by parameter changes), consider documenting this explicitly to set clear expectations during crisis management scenarios.

Tenbin: Acknowledged. The desired behavior is to change cooldown time only for new stakers.

Spearbit: Acknowledged.

5.3.7 Slippage check assumes 1:1 price ratio between source and destination tokens

Severity: Low Risk

Context: CollateralManager.sol#L347-L353

Description: The `CollateralManager`'s swap slippage validation assumes that source and destination tokens have equal value, comparing normalized token amounts directly without accounting for potential price differences:

```
// normalize decimal amounts to compare slippage
uint256 expectedAmountOut = _normalizeTo18(params.amount,
↪  IERC20Metadata(params.srcToken).decimals());
uint256 minAmountOut = _normalizeTo18(params.minReturnAmount,
↪  IERC20Metadata(params.dstToken).decimals());
```

```
// Calculate bps and verify slippage is within tolerance
uint256 slippageBps = (expectedAmountOut - minAmountOut) * BASIS_PRECISION / expectedAmountOut;
if (slippageBps > swapTolerance[params.srcToken][params.dstToken]) revert InvalidSlippage();
```

This calculation treats `expectedAmountOut` (the input amount) and `minAmountOut` (the output amount) as if they represent equivalent values. While this assumption may be reasonable when swapping between closely-pegged stablecoins under normal market conditions, it breaks down during depeg events or when stablecoins trade at different prices.

Consider a scenario during a USDC depeg event where USDC trades at $0.95 and USDT remains at $1.00:

- Swapping 100 USDC for 95 USDT represents fair market value ($95 for $95).
- The current implementation calculates: slippageBps = (100 - 95) × 10000 / 100 = 500 (5%).
- This 5% "slippage" is actually the market price difference, not true slippage.
- The swap might be incorrectly rejected if the tolerance is below 5%.

Conversely, a swap of 100 USDC for 90 USDT would show 10% slippage, but the actual value slippage is approximately 5% when accounting for USDC's $0.95 price.

Recommendation: Consider integrating price oracles to account for actual token values when calculating slippage:

```
// Normalize amounts to 18 decimals
uint256 normalizedSrcAmount = _normalizeTo18(params.amount,
↪  IERC20Metadata(params.srcToken).decimals());
uint256 normalizedDstAmount = _normalizeTo18(params.minReturnAmount,
↪  IERC20Metadata(params.dstToken).decimals());

// Get oracle prices for both tokens
uint256 srcPrice = IOracleAdapter(collateralOracles[params.srcToken].adapter).getPrice();
uint256 dstPrice = IOracleAdapter(collateralOracles[params.dstToken].adapter).getPrice();

// Calculate expected output amount accounting for price difference
uint256 expectedAmountOut = normalizedSrcAmount * srcPrice / dstPrice;

// Calculate slippage based on actual values
uint256 slippageBps = (expectedAmountOut - normalizedDstAmount) * BASIS_PRECISION /
↪  expectedAmountOut;
if (slippageBps > swapTolerance[params.srcToken][params.dstToken]) revert InvalidSlippage();
```

This approach ensures that slippage validation reflects true value loss rather than nominal token amount differences, providing more accurate protection during depeg events while maintaining appropriate checks under normal conditions. If price oracles are not available, consider documenting the 1:1 assumption and its limitations during non-standard market conditions.

Tenbin: Fixed in commit d04a30df. We decided to use actual amounts instead of this slippage check.

Spearbit: Fix verified.

5.3.8 Vesting period duration can't be changed when there is a current vesting

Severity: Low Risk

Context: StakedAsset.sol#L170-L177

Description:

```
if (vesting.end > block.timestamp) revert VestingNotCompleted();
```

When there is a current vesting, the admin is not able to change the vesting period duration.

Recommendation: Allow admin the freedom to change the vesting period duration.

Tenbin: Fixed in commit d04a30df.

Spearbit: Fix verified.

5.3.9 Yield of last withdrawal may be lost

Severity: Low Risk

Context: StakedAsset.sol#L146-L165

Description:

```
function cooldownAssets(uint256 assets) external nonRestricted(msg.sender) returns (uint256
↪  shares) {
    if (assets > maxWithdraw(msg.sender)) revert CooldownExceededMaxWithdraw();
    shares = previewWithdraw(assets);
    // forge-lint: disable-next-line(unsafe-typecast)
    cooldowns[msg.sender].assets += uint160(assets);
    // forge-lint: disable-next-line(unsafe-typecast)
    cooldowns[msg.sender].end = uint96(block.timestamp + cooldownPeriod);
    _withdraw(msg.sender, address(silo), msg.sender, assets, shares);
    emit CooldownStarted(msg.sender, assets);
}

/// @notice Unstake shares that are in cooldown
/// @param to Account to transfer assets to
function unstake(address to) external nonRestricted(msg.sender) nonZeroAddress(to) {
    Cooldown memory cooldown = cooldowns[msg.sender];
    if (cooldown.end > block.timestamp) revert CooldownInProgress();
    delete cooldowns[msg.sender];
    silo.withdraw(to, cooldown.assets);
    emit Unstake(msg.sender, to, cooldown.assets);
}
```

Yield from silo are given through `reward` which are distributed to depositors who have not cooldown their shares yet. That would usually work as expected but there will be a small problem when the last withdrawer cools down their shares. The yield is then just lost and left in the contract when it is distributed using reward. Also `rescueToken` cant pull out the yield tokens as it can only be used for tokens other than `asset()`.

Recommendation: Consider taking care of this edge case by allowing `rescueToken` to work with yield tokens when there are no shares left(last withdrawal), so that atleast the admin can manually give the yield tokens (especially if its significant) to the last withdrawer.

Tenbin: Acknowledged.

Spearbit: Acknowledged.

5.3.10 Revenue withdrawal fails to redeem vault shares causing accounting mismatch

Severity: Low Risk

Context: CollateralManager.sol#L275-L292

Summary: The `withdrawRevenue()` function in CollateralManager calculates revenue based on increases in vault total assets but attempts to transfer tokens directly without redeeming the corresponding vault shares. This causes the function to either revert due to insufficient balance or use tokens from other protocol operations.

Description: The CollateralManager tracks revenue by monitoring changes in the vault's total assets through `_computeNewRevenue()`. However, when revenue is withdrawn via `withdrawRevenue()`, the function only updates accounting and transfers tokens without actually redeeming vault shares:

```
function withdrawRevenue(address collateral, uint256 amount) external
↪  onlyRole(REVENUE_MANAGER_ROLE) {
    if (amount == 0) revert ZeroAmount();
```

```
    if (amount > pendingRevenue[collateral]) revert InsufficientRevenue();
    pendingRevenue[collateral] -= amount;
    IERC20(collateral).safeTransfer(revenueModule, amount);
    emit RevenueWithdrawn(collateral, amount);
}
```

This creates a mismatch between accounting and actual token movements:

1. Revenue is calculated based on vault.totalAssets() increases.
2. The actual tokens corresponding to this revenue remain locked in the vault as shares.
3. When `withdrawRevenue()` tries to transfer tokens, it can only succeed if tokens happen to be available in the CollateralManager from other operations.

The typical flow would be:

- Vault earns yield, vault.totalAssets() increases from 100 to 110.
- `pendingRevenue[collateral]` is credited with 10 tokens.
- `withdrawRevenue(collateral, 10)` is called.
- Function attempts `safeTransfer(revenueModule, 10)` from CollateralManager's balance.
- Issue: The 10 tokens of value exist as vault shares, not as liquid tokens in CollateralManager.

Impact Explanation: The impact is medium to high as this breaks the revenue withdrawal mechanism. The protocol cannot systematically collect revenue without either having sufficient liquidity from other operations or implementing workarounds. This results in an accounting issue where tracked revenue cannot be properly extracted from vaults.

Likelihood Explanation: The likelihood is high. Revenue withdrawal will consistently fail or behave incorrectly in normal protocol operation unless there happens to be sufficient token balance from unrelated operations.

Recommendation: Consider redeeming the appropriate vault shares before transferring revenue:

```
  function withdrawRevenue(address collateral, uint256 amount) external
  ↪  onlyRole(REVENUE_MANAGER_ROLE) {
      if (amount == 0) revert ZeroAmount();
      if (amount > pendingRevenue[collateral]) revert InsufficientRevenue();
      pendingRevenue[collateral] -= amount;
+
+     // Redeem vault shares to obtain the revenue tokens
+     IERC4626 vault = collateralVault[collateral];
+     vault.withdraw(amount, address(this), address(this));
+
      IERC20(collateral).safeTransfer(revenueModule, amount);
      emit RevenueWithdrawn(collateral, amount);
  }
```

Note on rounding: The `previewWithdraw()` function rounds shares up per ERC4626 specification, meaning the vault will redeem slightly more shares than needed to obtain exactly `amount` assets. This causes a small discrepancy where the protocol's vault shares decrease by more than the asset-equivalent of the withdrawn revenue. While these are typically dust amounts, this behavior should be documented if a more complex solution that tracks share-based revenue accounting is not desired.

Tenbin: Acknowledged. We decided to keep the current behavior as it is.

Spearbit: Acknowledged.

5.3.11 Slippage check can underflow when `minReturnAmount > expectedAmountOut`

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

Description: In `CollateralManager._verifySlippage()` the contract normalizes:

- `expectedAmountOut` from `params.amount`.
- `minAmountOut` from `params.minReturnAmount`.

and then computes slippage in bps:

```
uint256 slippageBps = (expectedAmountOut - minAmountOut) * BASIS_PRECISION / expectedAmountOut;
```

this formula assumes `expectedAmountOut >= minAmountOut`. If `minAmountOut > expectedAmountOut`, the subtraction underflows and the swap reverts before execution.

Stablecoins frequently trade slightly above/below $1 and can flip relative value (e.g., USDT > USDC or vice versa). If `minReturnAmount` is taken directly from an aggregator quote (e.g., 1inch api / batch dutch-auction based routes), it may legitimately imply more output units (`dst` token) than input is worth. For example:

- USDC = $0.9998.
- USDT = $1.0005.
- Swap: USDT (src) → USDC (dst).
- Aggregator quote sets `minReturnAmount` tightly basis pricing USDT at 1.0001$ notional to allow some deviation from current price (1.0005$) just enough to not leave MEV.
- Result: after normalization, minAmountOut > expectedAmountOut → underflow revert in `_verifySlippage()` and also implying swap will need to executed at worse price than 1:1 in future.

This is true irrespective of tokens are priced using relative oracle or at 1:1 ratio, as oracles would price some stablecoins at upper bound of 1$ or have market lag compared to AMM / auction price.

Recommendation: Handle ”positive slippage” safely:

- If `minAmountOut >= expectedAmountOut`, treat slippage as 0 bps (favorable quote) and do not revert.
- Consider sanitizing the payload on backend server such that there is no positive slippage basis swap params, but this would miss opportunity to execute trades at more favourable price.

Tenbin: Fixed in commit d04a30df.

Spearbit: Fix verified.

5.3.12 1inch swap flags in payload are not validated

Severity: Low Risk

Context: SwapModule.sol#L72

Description: `SwapModule.swap1Inch()` forwards curator-supplied `IAggregationRouterV6.SwapDescription.flags` to the 1inch router without validation and ignores `spentAmount` returned by the router.

```
uint256 private constant _NO_PARTIAL_FILLS_FLAG = 1 << 255;
uint256 private constant _ALLOW_MULTIPLE_FILLS_FLAG = 1 << 254;
uint256 private constant _PRE_INTERACTION_CALL_FLAG = 1 << 252;
uint256 private constant _POST_INTERACTION_CALL_FLAG = 1 << 251;
uint256 private constant _NEED_CHECK_EPOCH_MANAGER_FLAG = 1 << 250;
uint256 private constant _HAS_EXTENSION_FLAG = 1 << 249;
uint256 private constant _USE_PERMIT2_FLAG = 1 << 248;
uint256 private constant _UNWRAP_WETH_FLAG = 1 << 247;
```

> **Note:** These might differ on different chains and across aggregator version, the above-mentioned flags are from ethereum router v6 deployment and not documented anywhere officially by 1inch.

If wrong flags are passed, it can have negative impact such as:

- When `_NO_PARTIAL_FILLS_FLAG` is false, some quantity `src` token might not be used and forever stuck on swap module as there is no way to retrieve it.
- Nonce and time-based expiry on quote might not be enforceable.
- Might allow interacting with arbitrary contracts pre and post swap, which may enable read-only re-entrancy on certain functions.

Recommendation: Consider validating swap flags, at least critical subset of it.

Tenbin: Fixed in commit d04a30df.

Spearbit: Fix verified. The partial fills flag is now checked along with the return values. Other suggested flags (e.g. multiple fills) are not checked.

5.3.13 DoS in `CollateralManager` deposit & withdraw due to market and capacity constraints

Severity: Low Risk

Context: CollateralManager.sol#L258, CollateralManager.sol#L269

Description: The `CollateralManager` uses Morpho `vault-v2` which allocate assets through adapters. Both deposits and withdrawals can be DoS'd.

DoS Scenarios by Configuration:

Configuration	Deposit	Withdraw
Liquidity adapter set	Auto-allocates on deposit; reverts if adapter's absolute/relative caps reached.	Auto-deallocates on withdraw; reverts if underlying liquidity borrowed. **Single adapter**: blocked if that adapter illiquid. **Multiple adapters**: total withdrawable = Σ available liquidity across all; partial DoS if some illiquid.
Liquidity adapter not set	No DoS risk; assets remain idle in vault. **Single/multiple adapters**: deposits unaffected since no auto-allocation occurs regardless of adapter count.	reverts if no idle balance Single adapter: reverts if that adapter illiquid. Multiple adapters: can deallocate from whichever has liquidity, if at-least one is liquid. needs external `forceDeallocate` or should be de-allocated by allocator role.

DoS Scenarios by Adapter Type:

Underlying Adapter Type	Deposit	Withdraw
MorphoVaultV1Adapter	Caps on adapter id limit deposits. **Single adapter**: one nested MetaMorpho's caps apply. **Multiple VaultV1Adapters**: each wraps different MetaMorpho; caps checked independently per adapter id.	Nested illiquidity: vault-v2 → VaultV1Adapter → MetaMorpho → Morpho Blue markets. **Single adapter**: if any Blue market in MetaMorpho's queue utilized → revert cascades up. **Multiple VaultV1Adapters**: each has independent MetaMorpho; illiquidity in one doesn't block others

Morpho Blue Market Adapter	Caps on market id limit deposits. **Single adapter**: direct exposure to one market's cap. **Multiple Market Adapters (same market)**: share same id/cap. **Multiple Market Adapters (different markets)**: independent caps per market id.	Direct market exposure. **Single adapter**: if that market highly utilized → immediate revert. **Multiple Market Adapters (same market)**: all blocked together since same liquidity source. **Multiple Market Adapters (different markets)**: independent liquidity; can deallocate from liquid markets.

Due to this:

- Rebalance will revert, which may cause loss of peg for synthetic RWA assets.
- Withdrawal of revenue and distribution of yield will not be possible.

The adapters which have fixed or stable rate interest distribution are more likely to DOS than ones with adaptive curve IRM because as interest spikes very high on max utilization, it incentivizes repayments.

It is possible for adversarial actors to sandwich rebalance txn by deposit large chunk to underlying vaults/markets and then instantly withdrawing almost risk free to block rebalance.

Recommendation: To solve for deposits:

- Instead of reverting in above-mentioned deposit cases cases, keep it idle within tenbin and re-deploy to yield sources when space is available.

To solve for withdraw:

- It might not be always possible to withdraw collateral from offchain sources (CEX) and make it available on-chain as it will increase leverage beyond maintenance bounds.
- So, a certain percentage of TVL or on-chain deployed collateral should be kept idle within system to serve withdrawals.

Tenbin: Acknowledged.

Spearbit: Acknowledged.

5.3.14 Missing bad-debt protections in morpho vault-v2 deposit for metamorpho v1.1 adapters

Severity: Low Risk

Context: CollateralManager.sol#L258

Description: `CollateralManager` performs ERC4626 deposit with no slippage or bad-debt check bounds. In case of bad-debt, when a `MorphoVaultV1Adapter` is used with v1.1 version of the metamorpho vault, it is not realized and share price is non-decreasing. And, hence `MorphoVaultV1Adapter.realAssets` will not reduce assets, implying loss will not be reflected in vault-v2's share price. Reference to no bad-debt realization mechanism:

- MorphoVaultV1Adapter.sol#L17-L18
- MetaMorphoV1_1.sol#L919-L948

Due to this, if metamorpho v1.1 ever accrues bad-debt or shortfall and `CollateralManager` keeps depositing, it will be providing exit liquidity for other LPs on that specific metamorpho vault, and will be minting share at much higher price.

Recommendation:

- Avoid depositing if for any of the attached `MorphoVaultV1Adapter`, `total assets - real total assets > threshold`. To calculate real total assets, for each of the metamorpho adapter, factor in lost assets
- Avoid depositing if for any of the attached `MorphoVaultV1Adapter` (with v1.1) vault's `lostAssets` are greater than some notional threshold (generally very low, so that dust amount of losses cannot DoS the deposit).

Tenbin: Fixed in commit d04a30df. We add a slippage guard to share price when performing a deposit or withdraw. The curator can calculate slippage off-chain and ensure the share price does not decrease below that (or above that).

Spearbit: Fix verified.

5.4 Gas Optimization

5.4.1 Using `safeIncreaseAllowance` instead of forceApprove adds unnecessary checks and complexity

Severity: Gas Optimization

Context: CollateralManager.sol#L135-L136

Description: The `CollateralManager` contract uses `safeIncreaseAllowance()` to set token approvals:

```
IERC20(collateral).safeIncreaseAllowance(controller, type(uint256).max);
IERC20(collateral).safeIncreaseAllowance(vault, type(uint256).max);
```

The `safeIncreaseAllowance()` and `safeDecreaseAllowance()` functions perform an additional allowance check before setting the new value, which adds unnecessary gas overhead compared to `forceApprove()`. Additionally, using relative allowance changes (`safeIncreaseAllowance`/`safeDecreaseAllowance`) is less clear than absolute values when the protocol consistently sets either maximum allowance or zero.

The `forceApprove()` function from OpenZeppelin's SafeERC20 library provides a safer and more efficient alternative by:

- Setting allowances to absolute values without unnecessary checks.
- Handling both standard and non-standard ERC20 tokens (like USDT that require setting to 0 first).
- Making the code's intent clearer when setting max or zero allowances.

Recommendation: Consider using `forceApprove()` instead of `safeIncreaseAllowance()`:

```
- IERC20(collateral).safeIncreaseAllowance(controller, type(uint256).max);
- IERC20(collateral).safeIncreaseAllowance(vault, type(uint256).max);
+ IERC20(collateral).forceApprove(controller, type(uint256).max);
+ IERC20(collateral).forceApprove(vault, type(uint256).max);
```

This simplifies the code, reduces gas costs, and makes the intent clearer when setting absolute allowance values.

Tenbin: Fixed in commit d04a30df.

Spearbit: Fix verified.

5.5 Informational

5.5.1 Chainlink price freshness unchecked

Severity: Informational

Context: OracleAdapter.sol#L24

Description: `OracleAdapter`'s `getPrice` returns `oracle.latestAnswer()` scaled by `1e10` without verifying round freshness, `answeredInRound`, or non-zero/positive values. A paused or stuck feed can keep serving an outdated answer that `Controller.verifyOrder` will treat as valid for tolerance checks, allowing mint or redeem flows to proceed using stale prices and misprice collateralization, or keep using an old quote after updates stop entirely.

Recommendation: Fetch round data via `latestRoundData`, require `answeredInRound` to match the returned round, reject non-positive answers and enforce a heartbeat on `updatedAt`. Revert when any condition fails so stale or invalid Chainlink responses are not accepted. Finally, consider validating in the constructor that the Chainlink's Price Feed decimals are equal to 8 so the decimal adjustment in the `getPrice` function is always correct.

Tenbin: Fixed in commit d04a30df.

Spearbit: Fix verified. The commit updates `OracleAdapter.getPrice()` to use `latestRoundData`, checks `answeredInRound`, staleness, non-positive answers and enforces decimals in the constructor.

5.5.2 EIP-712 typehash mismatch in deploy script

Severity: Informational

Context: DeployTestnet.s.sol#L239-L243

Description: The on-chain controller defines `ORDER_TYPEHASH` as

```
keccak256("Order(uint8 order_type,uint256 nonce,uint256 expiry,address payer,address
↪  recipient,address collateral_token,uint256 collateral_amount,uint256 asset_amount)")
```

but the testnet deployment script logs a different type string:

```
"OrderType order_type,uint256 nonce,uint256 expiry,address payer,address recipient,address
↪  collateral_token,uint256 collateral_amount,uint256 asset_amount"
```

If off-chain signing code copies the script's string, it will compute a domain-typed data hash that doesn't match the contract's type hash, causing all order signatures to fail verification.

Recommendation: Standardize on the contract's exact type string for all off-chain signing and logging. Update the deployment script to emit

```
Order(uint8 order_type,uint256 nonce,uint256 expiry,address payer,address recipient,address
↪  collateral_token,uint256 collateral_amount,uint256 asset_amount)
```

so tooling uses the same type hash as the controller.

Tenbin: Fixed in commit d04a30df.

Spearbit: Fix verified.

5.5.3 Controller update relies on caller-supplied list

Severity: Informational

Context: CollateralManager.sol#L171

Description: `updateController` depends on the admin passing every supported collateral to revoke the old controller's allowance and grant the new one. If a collateral is omitted, its allowance to the old controller remains and the new controller is not approved for that token, causing operational reverts and residual permissions. The function does not track collaterals on-chain, so completeness is not enforced at the contract level.

Recommendation: Consider tracking supported collaterals on-chain and iterate them inside `updateController` to revoke the old controller's allowance and grant the new one for every collateral automatically.

Tenbin: Fixed in commit d04a30df.

Spearbit: Fix verified. `updateController` now iterates the on-chain collaterals set and revokes/approves for every supported collateral automatically, removing reliance on a caller-supplied list.

5.5.4 Controller constructor skips ratio bound

Severity: Informational

Context: Controller.sol#L144

Description: The `Controller` constructor stores the initial custodian split `ratio_` without validation, while `setRatio` later enforces `ratio <= 1e18`. A deployment that supplies a ratio above `RATIO_PRECISION` will

be accepted and used to compute custodian/manager splits, leading to over-allocation and reverts if downstream logic assumes a 0-1e18 range. This is a one-time parameter and should be guarded the same way as the setter.

Recommendation: Apply the same bound in the constructor:

```
if (ratio_ > RATIO_PRECISION) revert InvalidRatio();
```

before assigning `ratio`.

Tenbin: Fixed in commit d04a30df.

Spearbit: Fix verified. The constructor now enforces `if (ratio_ > RATIO_PRECISION) revert InvalidRatio();` in the `Controller` contract.

5.5.5 Slippage guard floors bps tolerance

Severity: Informational

Context: CollateralManager.sol#L344-L354

Description: The slippage check computes basis points with integer division:

```
slippageBps = (expectedAmountOut - minAmountOut) * BASIS_PRECISION / expectedAmountOut;
```

Because the division floors, the computed bps can understate the real slippage by nearly 1 bps (and proportionally more for tiny trades). A swap that is slightly worse than the configured tolerance still passes because the truncated bps stays below the threshold. Impact: swaps can execute with marginally worse rates than intended.

Recommendation: Calculate slippage with rounding up so any extra loss is caught, for example

```
Math.mulDiv(diff, BASIS_PRECISION, expectedAmountOut, Math.Rounding.Up)
```

or by adding the denominator minus one before dividing. This enforces the configured tolerance exactly rather than allowing the extra window created by flooring.

Tenbin: Fixed in commit d04a30df.

Spearbit: Fix verified. The commit switched the bps calc to `Math.mulDiv(..., Math.Rounding.Ceil)`. Since then, the slippage-bps guard was removed entirely in favour of a per-pair `minSwapPrice`, so the flooring issue no longer exists.

5.5.6 ERC4626 first-depositor inflation risk mitigation relies on OZ virtual shares

Severity: Informational

Context: StakedAsset.sol#L42

Description: `StakedAsset` inherits OZ `ERC4626Upgradeable`. Classic ERC4626 vaults can suffer a first-depositor donation/inflation attack if unseeded: an attacker donates to skew share price and front-runs deposits. OZ v5 introduces ”virtual shares/assets” (decimals offset) to make this attack uneconomic by default, but it does not fully eliminate slippage sensitivity in near-empty vaults. The contract's NatSpec warns to seed a minimum stake at deployment; without seeding, early deposits still face slippage risk, though the virtual shares capture most of the would-be gain for the attacker. This is informational only; if you seed with 1e18 at deployment as planned, the practical risk is negligible.

Recommendation: Ensure that the vault is seeded with the planned 1e18 (or a non-trivial) initial deposit during/just after initialization to stabilize the share price.

Tenbin: Acknowledged.

Spearbit: Acknowledged.

5.5.7 SwapModule TODO is already implemented

Severity: Informational

Context: SwapModule.sol#L61

Description: `swap1Inch` includes a TODO comment ("params.minReturnAmount <= swapData.minReturnAmount (ENG-237)"), but the very next line enforces the check:

```
if (params.minReturnAmount > swapData.minReturnAmount) revert InvalidMinReturnAmount();
```

The TODO comment is, therefore, already implemented.

Recommendation: Remove the outdated TODO comment to reflect that the validation is already present.

Tenbin: Fixed in commit d04a30df.

Spearbit: Fix verified.

5.5.8 Missing nonce invalidation function prevents cancellation of accidental orders

Severity: Informational

Context: Controller.sol#L101-L103

Description: The Controller contract tracks used nonces through a mapping but does not provide a function for users to manually invalidate nonces:

```
/// @notice Keeps track of which nonces a payer has used
mapping(address => mapping(uint256 => bool)) nonces;
```

Without a nonce invalidation mechanism, users who have signed orders off-chain cannot cancel them if they were created accidentally or contain errors. The only way to prevent such orders from being executed is to wait for them to be filled or to ensure the nonce is used for a different transaction, which may not always be practical or timely.

This limitation can be problematic in scenarios where:

- Users sign incorrect or unintended orders.
- Market conditions change and users want to cancel pending orders.
- Users want to revoke orders that haven't been submitted on-chain yet.

Recommendation: Consider adding a function that allows users to invalidate specific nonces or ranges of nonces:

```
/// @notice Allows a user to invalidate specific nonces to cancel pending orders
/// @param nonce The nonce to invalidate
function invalidateNonce(uint256 nonce) external {
    nonces[msg.sender][nonce] = true;
    emit NonceInvalidated(msg.sender, nonce);
}
```

This would provide users with the flexibility to cancel pending orders and improve the overall user experience when managing signed orders.

Tenbin: Fixed in commit d04a30df.

Spearbit: Fix verified.

5.5.9 Systemic economic risks

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: The current Tenbin design mints ERC20 `AssetTokens` whose value is intended to track a reference asset (for example, gold) using an off-chain delta-one futures hedge and on-chain collateral that is deployed into yield strategies. While this architecture can provide good price tracking in normal conditions, the design introduces a large set of systemic economic risks.

The core mechanism is that each `AssetToken` represents an off-chain obligation: the protocol ”owes” holders the mark-to-market value of the reference asset. The hedge is implemented by a long futures or perpetual futures position sized to match the notional exposure. The on-chain collateral acts as margin backing plus a source of yield to cover funding costs and fees. In practice, this creates a three-way dependency: on-chain collateral value and yield, off-chain hedge performance and funding, and user behavior (mints, redeems and secondary market trading).

First, there is significant hedge-side risk that is not constrained on-chain. The protocol relies on futures being ”delta-one” with the reference asset (i.e. a 1:1 sensitivity), but in reality they can deviate due to several effects. ”Basis” is the persistent price difference between the futures contract and spot; if the futures trade at a premium or discount, the hedge does not perfectly track the underlying. ”Roll slippage” arises when expiring futures must be rolled into new contracts and the new contract is priced less favorably than the old one, generating a loss when the hedge is rolled. ”Temporary dislocations” are short-lived periods where the futures market diverges from spot due to liquidity imbalances, exchange outages or stressed conditions. If dated futures are used, rolling the hedge can lock in those relative losses versus spot; if perpetuals are used, high or spiking funding rates can persist for long periods. These effects can generate an ongoing net economic cost of maintaining the hedge that must be funded from on-chain yield or protocol surplus. The contracts do not appear to model or cap this basis and funding exposure, which means the system can drift into a persistent negative PnL state without any on-chain circuit breaker, even while the token continues to appear fully backed by a simple collateral ratio.

Second, there is margin and liquidation risk at the venue level. The off-chain hedge requires sufficient margin to absorb mark-to-market losses, meaning the unrealized losses that occur when the futures price moves against the hedge as the underlying asset changes in value. If the collateral allocated to the custodian is insufficient or cannot be topped up quickly enough, the futures venue can partially or fully liquidate the hedge at unfavorable prices. While such liquidations are unlikely in calm markets with conservative margining, they become materially more likely during fast, large price moves, gaps around news events, or operational incidents that delay margin transfers. After a liquidation event, the protocol is left with the full on-chain liability (the outstanding `AssetToken` supply) but no effective hedge, turning the system into an unintended speculative position on the reference asset. The on-chain logic that selects the ratio of collateral sent to the custodian versus retained in the `CollateralManager` does not encode any explicit margin-of-safety rules tied to observed or configured volatility, so it is possible to choose parameters that look safe statically but are inadequate in a stress scenario.

Third, there are structural risks in the on-chain collateral and yield strategies. To pay for funding and fees, the protocol must chase yield in various on-chain instruments such as lending markets, LSTs, or yield-bearing stable assets. These instruments each introduce their own risk of depeg, smart contract loss, governance attack, or liquidity crisis. The current design allows arbitrary vaults or strategies to be whitelisted and used by the `CollateralManager` based on governance decisions. However, there is no explicit on-chain risk accounting that ties the protocol's liability (outstanding `AssetToken` value) to the risk profile, diversification or liquidity of the chosen yield strategies. As a result, governance could allocate a large share of backing into higher-yield but materially riskier vaults to keep headline APYs attractive, while silently increasing the probability that a single depeg or vault exploit causes a shortfall against `AssetToken` holders.

Fourth, there is a liquidity and maturity mismatch between the redeemability of `AssetToken` and the underlying positions. `AssetToken` can be minted and redeemed on demand at or near the oracle price, but collateral is often invested in strategies that have withdrawal queues, cooldowns or limited liquidity. The `CollateralManager` does have caps and tolerances around rebalances and swaps, but these are static configuration values and not derived from a liquidity stress model. In a redemption wave, the protocol might be forced to unwind yield positions at a discount, pay early exit penalties, or temporarily pause redemptions, creating a first-mover advantage where early redeemers exit at close to par and later redeemers face discounts or delays. None of this is directly visible to token holders today because the contracts do not distinguish between immediately withdrawable collateral and collateral that is locked or slow to exit, nor do they expose an on-chain notion of ”available redemption capacity” over different time horizons.

Fifth, there is significant centralization and counterparty risk in both off-chain operations and on-chain governance.

Off-chain exchanges, brokers, and custodians can fail, be hacked, or become inaccessible due to regulatory action, freezing margin and PnL that the on-chain logic assumes is available. On-chain privileged roles (admins, curators, cap managers) can concentrate risk by selecting aggressive strategies, loosening swap tolerances, or prioritizing revenue extraction over prudence. Because the contracts do not enforce strong, hard-coded risk limits around these actions, users are effectively trusting a small set of operators and governance participants to manage the hedge and collateral prudently.

Sixth, the fee and PnL flows associated with funding, trading fees, gas, and yield are not modeled from the perspective of long-term solvency. The product relies on an informal assumption that on-chain yield will be sufficient to cover funding and costs over time, but funding rates in perpetual markets can remain elevated for months while on-chain yields compress. There is no on-chain budget mechanism that caps the size of the asset book relative to expected funding costs, nor any parametrized rule that halts minting when the long-term net carry (the expected difference between yield earned on collateral and all costs of maintaining the hedge, including funding, fees and slippage) is negative. That means the protocol can continue to grow TVL and supply while quietly running a structurally negative economic spread until a shock makes the shortfall obvious.

Seventh, the design is sensitive to path-dependence and tail events that are not captured by simple collateral ratios. Sudden spikes in volatility, exchange outages, oracle lags, or liquidity gaps can break the tight coupling between on-chain state transitions and off-chain hedge maintenance. If large mints or redeems occur during a market move and the hedge cannot be updated at the same time, the protocol can end up over- or under-hedged in ways that create large unaccounted PnL. Because there is no on-chain representation of the current hedge delta or real-time solvency, these episodes would be invisible until losses are realized.

Finally, from a user's perspective, the `AssetToken` is presented as a token that ”represents” the reference asset, but economically it is a claim on a complex, leveraged balance sheet that spans multiple venues and strategies. There is no single on-chain function or invariant that verifies that total assets (on-chain collateral plus off-chain hedge PnL) exceed total liabilities at any point in time. There is also no formal mechanism to express and enforce risk limits such as maximum share of collateral in any one strategy, maximum exposure to a single exchange, or minimum buffer between projected funding costs and projected yield. This opacity makes it difficult for integrators or holders to price risk correctly and increases the chance that the token trades at a discount or that losses are socialized in an ad-hoc manner after an incident.

Tenbin: Acknowledged.

Spearbit: Acknowledged.

5.5.10 Oracle delta tolerance measured against order price allows extreme price deviations

Severity: Informational

Context: Controller.sol#L363-L379

Description: The Controller contract's oracle price validation is intended as a security measure to prevent orders from executing at prices significantly away from spot. The implementation calculates the price delta using the order price as the denominator:

```
uint256 price = Math.mulDiv(collateralAmount, 1e18, order.asset_amount);
uint256 difference = price >= oraclePrice ? price - oraclePrice : oraclePrice - price;
uint256 delta = Math.mulDiv(difference, 1e18, price);
if (delta > oracleData.tolerance) revert ExceedsOracleDeltaTolerance();
```

This formula creates asymmetric bounds around the oracle price:

- For underpriced orders (P < O): The lower bound is strictly enforced. With 100% tolerance, prices must be at least 50% of the oracle price.
- For overpriced orders (P ≥ O): The upper bound weakens dramatically as tolerance approaches 100%. At exactly 100% tolerance (the maximum allowed by `MAX_ORACLE_TOLERANCE = 1e18`), the check `|P - O| / P ≤ 100%` is always true for any P ≥ O, effectively removing the upper bound entirely.

At maximum tolerance, orders at 10x, 100x, or any multiple above the oracle price will pass validation, as the calculated delta asymptotically approaches but never exceeds 100%. During periods of high volatility, operators

might temporarily raise tolerance to accommodate legitimate price movements without fully understanding that 100% tolerance removes upper price protection.

Recommendation: Consider calculating the delta relative to the oracle price to provide symmetric bounds:

```
  uint256 price = Math.mulDiv(collateralAmount, 1e18, order.asset_amount);
  uint256 difference = price >= oraclePrice ? price - oraclePrice : oraclePrice - price;
- uint256 delta = Math.mulDiv(difference, 1e18, price);
+ uint256 delta = Math.mulDiv(difference, 1e18, oraclePrice);
  if (delta > oracleData.tolerance) revert ExceedsOracleDeltaTolerance();
```

This provides symmetric interpretation where tolerance of 10% means prices in range [0.9×O, 1.1×O] and tolerance of 100% means prices in range [0, 2×O], ensuring an upper bound exists even at maximum tolerance settings.

Tenbin: Fixed in commit d04a30df.

Spearbit: Fix verified.

5.5.11 Unsafe downcasting in cooldown assets could lead to truncation

Severity: Informational

Context: StakedAsset.sol#L150

Description: The StakedAsset contract performs an unsafe downcast from `uint256` to `uint160` when recording cooldown assets:

```
cooldowns[msg.sender].assets += uint160(assets);
```

While the risk of truncation is low in practice (since `uint160` can represent values up to ~1.46e48, which is far beyond realistic token amounts), using unchecked type conversions leaves the contract vulnerable to unexpected behavior if extremely large values are somehow passed.

The downcast could silently truncate values without reverting, potentially causing users to lose their cooldown state or recording incorrect asset amounts.

Recommendation: Consider using OpenZeppelin's SafeCast library to ensure type conversions revert on overflow:

```
+ import {SafeCast} from "@openzeppelin/contracts/utils/math/SafeCast.sol";

  contract StakedAsset {
+     using SafeCast for uint256;

-     cooldowns[msg.sender].assets += uint160(assets);
+     cooldowns[msg.sender].assets += assets.toUint160();
  }
```

This provides an additional safety guarantee without meaningful gas overhead and follows best practices for type conversions in production contracts.

Tenbin: Fixed in commit d04a30df.

Spearbit: Fix verified.

5.5.12 Suggestion for a better way to calculate the new cooldown endtime

Severity: Informational

Context: StakedAsset.sol#L138-L140

Description: In `StakedAsset.sol`:

```
cooldowns[msg.sender].assets += uint160(assets);
// forge-lint: disable-next-line(unsafe-typecast)
cooldowns[msg.sender].end = uint96(block.timestamp + cooldownPeriod);
```

When an user queues new shares for cooldown, the `endtime` extends for all shares in cooldown.

If the user currently has alot of assets queued that are already near the end of its cooldown then it would be very wasted if the user would to newly cool down a significantly lesser amount of shares.

Maybe the contract can consider a weighted version of cooldown end time. (refer to recommendation section).

Recommendation: Set the new endtime to the weighted average of the newly calculated endtime wif the old one based on the amounts of new shares queued and the old amounts of shares already queued.(for the weighted average).

Tenbin: Acknowledged.

Spearbit: Acknowledged.

5.5.13 Code quality improvements

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description and Recommendations:

1. Validate `srcReceiver` in 1inch swap to prevent misconfiguration.

   The 1inch swap adapter validates the destination receiver but not the source receiver. A keeper could accidentally craft calldata that attempts to pull tokens from an address that hasn't approved the router, causing swaps to revert with unclear errors.

   ```
       function swap1Inch(SwapParameters memory params, bytes calldata data) internal {
           (IAggregationExecutor executor, IAggregationRouterV6.SwapDescription memory
           ↪  swapData, bytes memory route) =
               abi.decode(data, (IAggregationExecutor,
               ↪  IAggregationRouterV6.SwapDescription, bytes));

           // sanitize inputs
           if (params.router != router) revert InvalidRouter();
           if (params.srcToken != address(swapData.srcToken)) revert InvalidSrcToken();
           if (params.dstToken != address(swapData.dstToken)) revert InvalidDstToken();
           if (params.amount != swapData.amount) revert InvalidAmount();
           if (params.minReturnAmount > swapData.minReturnAmount) revert
           ↪  InvalidMinReturnAmount();
           if (swapData.dstReceiver != manager) revert InvalidReceiver();
   +       if (swapData.srcReceiver != address(this)) revert InvalidSrcReceiver();
   ```

2. Remove deprecated `MintRedeemPauseStatusChanged` event from interface.

   The `MintRedeemPauseStatusChanged` event is declared in the IController interface but is deprecated in favor of the new pause system that emits `PauseStatusChanged`. The unused event declaration should be removed to avoid confusion.

   ```
   - event MintRedeemPauseStatusChanged(bool paused);
   ```

3. Add array length validation in multicall.

   The `multicall()` function iterates over `data.length` but indexes `targets[i]` without validating that both arrays have the same length. Malformed input causes an array panic rather than a clean revert.

   ```
       function multicall(address[] calldata targets, bytes[] calldata data) external {
   +       if (targets.length != data.length) revert ArrayLengthMismatch();
   ```

```
        for (uint256 i = 0; i < data.length; i++) {
            // ... call logic
        }
    }
```

4. Validate oracle feed decimals in constructor.

   The OracleAdapter assumes all Chainlink feeds use 8 decimals by hard-coding a `* 1e10` conversion. While this is currently true for all Chainlink feeds, validating this assumption once in the constructor provides an explicit guarantee and makes the contract more robust.

```
    constructor(address oracle_) {
       oracle = AggregatorInterface(oracle_);
+      if (oracle.decimals() != 8) revert UnsupportedFeedDecimals();
    }
```

5. Add explicit check to prevent swap cap underflow.

   The destination swap cap enforcement subtracts the actual output without bounds checking. When positive slippage causes actual output to exceed the remaining cap, the subtraction causes an underflow revert with an unclear error message.

```
    uint256 balanceAfter = IERC20(params.dstToken).balanceOf(address(this));
    if (balanceAfter - balanceBefore < params.minReturnAmount) revert
    ↪  InsufficientAmountReceived();

+   uint256 actualOut = balanceAfter - balanceBefore;
+   if (actualOut > dstTokenCap) revert SwapCapExceeded();
+
    swapCap[params.srcToken] = srcTokenCap - params.amount;
-   swapCap[params.dstToken] = dstTokenCap - (balanceAfter - balanceBefore);
+   swapCap[params.dstToken] = dstTokenCap - actualOut;
```

Tenbin: Fixed in commit d04a30df.

Spearbit: Fix verified.

5.5.14 Whitelist 1inch executors in `SwapModule.swap1inch`

Severity: Informational

Context: SwapModule.sol#L72

Description: 1inch doesn't enforce any check on `executor` used for the swap call, the worst case is just leading to consuming up-to slippage threshold which may be reasonable. However, it can do some unwanted & unintended things in intermediate context (for e.g calling tornado-swap) to get `SwapModule` labelled as bad actor on certain lists.

Recommendation:

- Consider maintaining whitelist mapping for `executor`.
- Consider validating `executor` at the backend server, however it increases trust assumptions for EOA.

Tenbin: Acknowledged. We are assuming 1Inch executors are acting faithfully. Because we enforce strict slippage checks, it is extremely low risk.

Spearbit: Acknowledged.

5.5.15 Incorrect morpho integration setup in test-case

Severity: Informational

Context: MorphoIntegration.t.sol#L33-L35

Description: morpho vault-v2, morpho adapter and morpho vault-v1.1 (metamorpho v1.1) are deployed directly, while they should be deployed from relevant factories. Especially, `MorphoVaultV1Adapter` must be strictly used with only morpho v1 and v1.1 vault (from respective factories), if used with generic & arbitrary ERC4626s then bunch of stuff breaks.

Recommendation: Use specific factories in test-cases:

- MetaMorphoFactory.sol
- MorphoVaultV1AdapterFactory.sol
- VaultV2Factory.sol

And in production, strictly use `MorphoVaultV1Adapter` only with morpho v1 / 1.1 vaults.

Tenbin: Fixed in commit d04a30df.

Spearbit: Fix verified.