PUBLIC SAMPLE — REDACTED. A genuine Vari review with the client's identity and identifying scope details removed (shown as [redacted]). The redacted text is stripped from the file, not merely hidden. Findings, severities, locations, evidence and recommendations are unaltered.

Vari

FULL-STACK SECURITY REVIEW

Security Review

[redacted] · multi-collateral stablecoin · [redacted] + staking vault · LayerZero oVault cross-chain

CLIENT [redacted]	REPORT DATE 2026-08-28
STACK Solidity / EVM	TIER Full
VERSION REVIEWED [redacted]	

01 Executive Summary

1	3	7	13	3
CRITICAL	HIGH	MEDIUM	LOW	INFO

4 findings at High severity or above. The most serious is **PROTO-11 — A frozen address self-lifts its own restriction via renounceRole, disarming the entire compliance layer**. Any address the compliance layer freezes can unfreeze itself and move funds: a restricted holder blocked from transferring self-renounces and moves 5,000,000e18 [redacted] out. On [redacted]+ the self-lift also disarms the recovery lever, because `redistributeLockedAmount` reverts unless `_isBlacklisted(from)` holds -- a frozen staker can place their balance permanently beyond seizure. Since the freeze is the mechanism behind every sanction, blacklist and court-ordered recovery action the protocol can take, this reduces the compliance layer to a formality on all four token contracts.

In total 27 findings: 1 Critical · 3 High · 7 Medium · 13 Low · 3 Info. Each is stated with its location in the source, the conditions under which it is reachable, and a recommended fix. Findings that were driven to a working proof carry that proof beneath them.

02 Scope of Review

LAYER	STATUS
Smart contracts	Reviewed
Deployment and configuration	Reviewed
Cross-chain messaging	Reviewed

Version reviewed: Local working tree at the provided source snapshot (no git metadata present). Source root [redacted]/contracts/contracts, 19 .sol files (13 non-interface), aggregate md5 of all contract sources bf8a5bfa7c2a2d26a35359d3b22f2329. Compiled with solc 0.8.22+commit.4fc1097e, optimizer on, 20,000 runs; dependency set installed from [redacted]/contracts/package.json (@layerzerolabs/oft-evm, ovault-evm 0.0.6, oapp-evm, @openzeppelin/contracts + contracts-upgradeable v5).

03 Methodology

The review began with the architecture: every path along which value moves — minting, redemption, deposits, withdrawals, staking, bridging and settlement — was traced end to end, and every privileged role was mapped to the actions it can take and the key that holds it. Each module was then read in full against that map, looking for places where the code does something other than what its author intended, where a control is present but does not bind, or where value can be created or destroyed without a corresponding entry elsewhere.

That reading was checked against the defect classes this kind of system has failed on historically — drawn from the published record of audited protocols — so that a known failure mode is not missed because it did not occur to the reviewer on the day. Alongside it, a separate reading targeted the opposite risk: defects specific to this protocol's own logic, which by definition match no known pattern and which no checklist would surface.

Nothing was reported on suspicion. The contracts were compiled and deployed locally through their own initialization scripts, and each hypothesis was driven to a verdict by a test that either reproduces the problem or fails to. Where a finding below carries a proof, that test was executed and its output is quoted verbatim. Where a path could not be exercised in a local environment, it is stated as unverified rather than presented as sound. Findings were then ranked by what an attacker actually gains and what it costs them to try.

All work was performed against a local build. No transaction was sent, no production endpoint was contacted, and no client key was handled at any point during this review.

04 Findings

PROTO-11 · **Critical**

A frozen address self-lifts its own restriction via renounceRole, disarming the entire compliance layer

LOCATION

```
The restriction status IS an AccessControl role: [redacted].sol:968-970
`_isBlacklisted(a) => hasRole(FULL_RESTRICTED_ROLE,
```

The restriction flag is an AccessControl role rather than dedicated state: `[redacted]._isBlacklisted` ([redacted].sol:968-970) returns `hasRole(FULL_RESTRICTED_ROLE, account)`, and `[redacted]+._isBlacklisted` ([redacted]+.sol:709-711) does the same against `FULL_RESTRICTED_STAKER_ROLE`. Every token contract overrides `renounceRole` -- [redacted].sol:1634-1637, [redacted]+.sol:765-768, [redacted].sol:733-736, [redacted] OFT.sol:198-201 and [redacted]+ OFT.sol:173-176 -- but each override guards exactly one role: `if (role == DEFAULT_ADMIN_ROLE) revert CantRenounceOwnership();` before delegating to `super`. OpenZeppelin's `AccessControlUpgradeable.renounceRole` requires only `callerConfirmation == _msgSender()`; it never consults the role admin. A restricted address therefore calls `renounceRole(FULL_RESTRICTED_ROLE, self)` and clears its own freeze in one transaction, with no privileged key involved and no precondition beyond holding the role. Nothing in the tree overrides `_revokeRole` or `_grantRole` to reimpose the check, so the hole is identical on the hub tokens and on both spoke OFTs.

Impact. Any address the compliance layer freezes can unfreeze itself and move funds: a restricted holder blocked from transferring self-renounces and moves 5,000,000e18 [redacted] out. On [redacted]+ the self-lift also disarms the recovery lever, because `redistributeLockedAmount` reverts unless `_isBlacklisted(from)` holds -- a frozen staker can place their balance permanently beyond seizure. Since the freeze is the mechanism behind every sanction, blacklist and court-ordered recovery action the protocol can take, this reduces the compliance layer to a formality on all four token contracts.

Recommendation. Override `_revokeRole` in each of the four token contracts to revert when `role` is `FULL_RESTRICTED_ROLE` / `FULL_RESTRICTED_STAKER_ROLE` and the caller is not the role admin -- `renounceRole` routes through `_revokeRole`, so one override covers both entry points instead of patching the five `renounceRole` overrides separately. If you keep the per-function fix, widen the condition at [redacted].sol:1634 (and [redacted]+.sol:765, [redacted].sol:733, [redacted] OFT.sol:198, [redacted]+ OFT.sol:173) from `role == DEFAULT_ADMIN_ROLE` to cover the restriction roles as well. The structural alternative is to stop representing the freeze as an AccessControl role at all and hold it in a dedicated `mapping(address => bool)` written only by `addToBlacklist`/`removeFromBlacklist`; that removes the whole class rather than one instance of it, and is what I would take if you are willing to migrate the state.

PROOF OF CONCEPT — EXECUTED

test/vari_verify/PriorFindingsCore.t.sol::test_USD01_BlacklistedSelfLiftsViaRenounceRole_[redacted],
::test_USD01_BlacklistedSelfLiftsViaRenounceRole_[redacted]+,
::test_USD01_AdminRoleIsTheOnlyRoleProtected;
test/vari_verify/PriorFindingXC01.t.sol::test_USD01_SpokeOFTsAlsoLetTheFrozenAddressSelfLift

```
[PASS] test_USD01_BlacklistedSelfLiftsViaRenounceRole_[redacted]() (gas: 63126)
  Logs:
    [redacted] escaped from frozen alice: 500000000000000000000000
[PASS] test_USD01_BlacklistedSelfLiftsViaRenounceRole_[redacted]+() (gas: 189704)
  Logs:
    seizure lever disarmed by the victim
[PASS] test_USD01_AdminRoleIsTheOnlyRoleProtected() (gas: 14595)
[PASS] test_USD01_SpokeOFTsAlsoLetTheFrozenAddressSelfLift() (gas: 71232)
  Logs:
    all four token contracts share the same defective override
```

PROTO-01 · High

Collateral without a wired Chainlink feed is credited at par (fail-OPEN, not fail-closed) — cross-asset depeg drain

LOCATION [redacted]/contracts/contracts/[redacted]/[redacted].sol:620 (_convertToMctAmount) and :636 (_convertToCollateralAmount)

Oracle pricing in [redacted] is opt-in per asset. `_convertToMctAmount` returns early with the plain decimal-normalised amount whenever `collateralFeed[asset] == address(0)`, i.e. an asset that is supported but has no feed is minted and redeemed at exactly $1.00 with no depeg protection. `addSupportedAsset` does not require a feed, and `setCollateralFeed` can only be called AFTER the asset is already supported, so every onboarding necessarily passes through a window in which the asset is live and priced at par. Three places in the code and the ops runbook state the opposite: the `_convertToMctAmount` NatSpec says 'Reverts if the asset has no feed', `_getValidatedPrice` is documented as 'Fail-closed: reverts if no feed is set', and `add-supported-asset.js:41` tells the operator 'after onboarding, the asset has NO price feed, so mint/redeem for it will revert (fail-closed) until an admin calls [redacted].setCollateralFeed'. That is factually wrong for the mint/redeem path — the early return in _convertToMctAmount is taken before _getValidatedPrice is ever reached. An operator following their own runbook will therefore believe a newly-onboarded asset is inert when it is in fact accepting deposits at par. Because [redacted] delegates all of its pricing, fee math and liquidity checks to these two [redacted] helpers, the mispricing propagates to the whole stablecoin.

Impact. Any supported collateral without a feed can be deposited at $1.00 regardless of its real market price and the resulting [redacted] redeemed against a healthy, oracle-priced leg. Executed PoC: 100,000 units of a depegged 6-decimal asset (real value $50,000) minted 100,000e18 [redacted] and immediately extracted 100,000 USDC from the vault — a $50,000 loss of honest collateral on a $100,000 deposit, unbounded except by the healthy leg's `collateralBalance` and `maxRedeemPerBlock`. The same primitive also lets a >$1 asset be under-credited, but the loss direction that matters is the drain.

Recommendation. Make the no-feed branch fail closed: revert in `_convertToMctAmount`/`_convertToCollateralAmount` when no feed is set, or require a feed at `addSupportedAsset` time (take feed+maxAge as parameters and validate atomically). If the 1:1 fallback must be kept for a migration window, gate it behind an explicit per-asset `allowUnpricedCollateral` flag

that governance must set, emit an event for it, and correct the three NatSpec blocks plus `add-supported-asset.js:41` so operators are not told the system is fail-closed when it is not.

PROOF OF CONCEPT — EXECUTED [redacted]/contracts/test/vari_redo/[redacted].t.sol::test_VR6_NoFeedAssetEnablesCrossAssetDepegDrain

```
[PASS] test_VR6_NoFeedAssetEnablesCrossAssetDepegDrain() (gas: 1124675)\nLogs:\n  [redacted] minted for 100,000 DPG (real value $50,000): 100000000000000000000000\n  USDC extracted: 100000000000\n\nControl (feed wired, same scenario):\n[PASS] test_VR6b_Refuted_DepegDrainWithFeedWired() (gas: 1369920)\nLogs:\n  [redacted] minted with feed wired: 50000000000000000000000
```

PROTO-N01 · High

[redacted]+.setVestingPeriod can push getUnvestedAmount() above the vault's asset balance, making totalAssets() revert and bricking the entire ERC4626 surface — and the StillVesting guard then locks governance out of undoing it

LOCATION [redacted]/contracts/contracts/[redacted]-plus/[redacted]+.sol:552 (setVestingPeriod), :584 (totalAssets), :592 (getUnvestedAmount)

CHANGES_POST_AUDIT.md states the design invariant explicitly: 'burnAssets(amount) (admin) burns vault-held [redacted], guarded by the unvested-reward amount and a 1e18 reserve floor (ReserveTooLowAfterBurn) so totalAssets() cannot underflow.' That invariant is enforced on exactly one path. totalAssets() is `balanceOf(this) - getUnvestedAmount()`, and getUnvestedAmount() is a pure function of three mutable state variables (vestingAmount, lastDistributionTimestamp, vestingPeriod). setVestingPeriod mutates the third one with only a `getUnvestedAmount() > 0 -> StillVesting` guard, which is structurally blind in the exact case that matters: while `vestingPeriod == 0` getUnvestedAmount() returns 0 unconditionally (line 594), so the guard cannot see the live vestingAmount/lastDistributionTimestamp pair it exists to protect. The whole flow uses only documented, supported operations: (1) governance sets vestingPeriod = 0 ('Setting period to 0 disables vesting (rewards vest instantly)' — the function's own NatSpec); (2) REWARDER_ROLE pushes rewards, which sets vestingAmount = R and lastDistributionTimestamp = now and makes them instantly claimable; (3) stakers legitimately withdraw those rewards, dropping the vault's [redacted] balance below R; (4) governance re-enables vesting with setVestingPeriod(P). Step 4 passes the guard, and getUnvestedAmount() jumps back to ~R, which now exceeds the vault's balance. totalAssets() underflows. A weaker but non-zero-period variant exists too: any setVestingPeriod(P_new) where P_new > (now - lastDistributionTimestamp) re-arms the stale vestingAmount. Crucially there is no way back: setVestingPeriod(0) now reverts StillVesting (its own guard, now un-blind), and transferInRewards reverts StillVesting as well, so neither of the two intended levers can restore the reserve.

Impact. totalAssets() reverting kills every ERC4626 code path on [redacted]+ at once: deposit, mint, withdraw, redeem, previewDeposit/Mint/Withdraw/Redeem, convertTo*, maxRedeem, maxWithdraw, cooldownAssets, cooldownShares and unstake (which calls previewRedeem). Only cancelCooldown and raw ERC20 transfers keep working. Governance cannot fix it with any setter — setVestingPeriod and transferInRewards are both self-blocked by StillVesting, and burnAssets reverts on its own unvested check. Recovery requires either waiting out up to MAX_VESTING_PERIOD (90 days) or a UUPS upgrade / an outright donation of [redacted] large enough to cover the phantom unvested amount. Staked principal is not stolen but is unwithdrawable for the duration.

Recommendation. Make the invariant hold at the state that defines it, not at one call site. In setVestingPeriod, compute the prospective unvested amount under the NEW period (not the current one) and reject any change that would make it exceed IERC20(asset()).balanceOf(address(this)); equivalently, zero out vestingAmount/lastDistributionTimestamp whenever vestingPeriod is set to 0, so a period of 0 cannot leave a latent schedule behind. Independently, make totalAssets() saturating (`bal > unvested ? bal - unvested : 0`) so no configuration can brick the vault, and add an explicit admin `resetVestingSchedule()` escape hatch.

PROOF OF CONCEPT — EXECUTED

[redacted]/contracts/test/vari_novel/NovelCore.t.sol::test_N01_VestingPeriodReenable_BricksTotalAsset

```
[PASS] test_N01_VestingPeriodReenable_BricksTotalAssets() (gas: 359787)
  vault [redacted] balance after full exit: 1
  getUnvestedAmount() after re-enable: 500000000000000000000
(test asserts: setVestingPeriod(8 hours) SUCCEEDS; totalAssets() then reverts; deposit() reverts;
maxWithdraw() reverts; setVestingPeriod(0) reverts StillVesting; transferInRewards reverts StillVesting)
```

PROTO-N02 · **High**

MAX_FEE_BPS is not the real redeem-fee ceiling: the uncapped minRedeemFeeAmount floor plus the M-2 clamp turns an over-fee into a silent 100% seizure of the redeemer's collateral instead of a revert

LOCATION [redacted]/contracts/contracts/[redacted]/[redacted].sol:853 (setMinRedeemFeeAmount), :1023 (_calculateRedeemFee), :1592 (M-2 clamp in _executeRedemption)

[redacted] advertises a hard fee ceiling in three places: `MAX_FEE_BPS = 1000` (10%), setRedeemFee/setInstantRedeemFee both enforce it, and setInstantRedeemFee's NatSpec reasons that 'the combined instant fee (redeemFeeBps + instantRedeemFeeBps) is at most 20%'. But _calculateRedeemFee returns `max(percentageFee, minRedeemFeeAmount)`, and minRedeemFeeAmount has no ceiling at all. Its only sanity cross-check — `minRedeemAmount > 0 && _minRedeemFeeAmount >= minRedeemAmount` — is skipped entirely while minRedeemAmount == 0, which is the value initialize() leaves it at and the value the deployed system runs with. The fee therefore has no upper bound in bps terms. The second half is what makes it dangerous: the M-2 audit fix (CHANGES_POST_AUDIT.md: '_executeRedemption clamps the collateral fee to the received collateral before subtracting, so a misconfigured minRedeemFeeAmount cannot underflow-revert and brick a queued redemption') converts what would have been a loud revert into `collateralAmount = 0`. redeem() has no min-out parameter and never consults previewRedeem (which does correctly return 0), so the caller's [redacted] is burned, 100% of the proceeds go to feeTreasury, and the call returns successfully. The mint side of the same contract handles the identical misconfiguration correctly — _mintWithCollateral reverts BelowMinimumAmount when the fee exceeds the deposit (fix #5) — so the two halves of the same fee model behave oppositely. This does not require a malicious admin: setting minRedeemFeeAmount to a routine gas-cost floor such as 1e18 ('$1 minimum fee') silently zeroes out every redemption below $1, because minRedeemAmount == 0 means there is no minimum redemption size to protect the user.

Impact. Total loss of the redeemed principal for the redeemer, with no revert and no on-chain signal other than the FeeCollected event. Under the benign configuration (a normal minimum-fee policy with the default minRedeemAmount of 0) this silently confiscates every small redemption. Under a hostile or compromised DEFAULT_ADMIN_ROLE it is a one-transaction, no-timelock, no-ceiling drain of the entire instant-redemption flow — including the cross-chain composer path — that the documented

MAX_FEE_BPS ceiling is specifically supposed to bound. It also applies to already-queued, grandfathered redemptions, which are re-priced at completion time.

Recommendation. (1) Bound minMintFeeAmount/minRedeemFeeAmount unconditionally, not only when the corresponding minimum amount is non-zero — e.g. require minRedeemAmount > 0 whenever minRedeemFeeAmount > 0, and require minRedeemFeeAmount < minRedeemAmount always. (2) Restore fail-closed behaviour on the redeem path: instead of clamping to 100%, revert BelowMinimumAmount when feeAmountCollateral >= receivedCollateral, mirroring _mintWithCollateral; the stranded-queue concern the clamp addressed is already covered by cancelRedeem(). (3) Give redeem() a minCollateralOut parameter so integrators and the composer can set slippage on the fee as well as on the price.

PROOF OF CONCEPT — EXECUTED

[redacted]/contracts/test/vari_novel/NovelCore.t.sol::test_N02_MinRedeemFee_Yields100PercentFee_Silen and ::test_N02b_OrdinaryMinFee_SilentlyZeroesSmallRedemptions

```
[PASS] test_N02_MinRedeemFee_Yields100PercentFee_Silently() (gas: 456580)
[PASS] test_N02b_OrdinaryMinFee_SilentlyZeroesSmallRedemptions() (gas: 403127)
Asserted: MAX_FEE_BPS()==1000 and minRedeemAmount()==0; setMinRedeemFeeAmount(1_000_000e18) accepted;
previewRedeem returns 0 while redeem() still succeeds with wasQueued==false, got==0,
usdc.balanceOf(alice) delta == 0 and usdc.balanceOf(treasury)==1_000e6 (100% of proceeds); the mint side
reverts BelowMinimumAmount under the mirror-image config. N02b: minRedeemFeeAmount=1e18 ->
redeem(0.5e18) returns 0, user balance unchanged, 0.5e18 [redacted] burned.
```

PROTO-02 · **Medium**

[redacted]+: a dust donation while totalSupply()==0 permanently bricks every deposit (MIN_SHARES floor)

LOCATION [redacted]/contracts/contracts/[redacted]-plus/[redacted]+.sol:662 (_checkMinShares), :670 (_deposit), :426 (unstake), :699 (_withdraw MIN_SHARES skip)

`_deposit` enforces `_checkMinShares()` AFTER minting: any deposit that leaves `0 < totalSupply() < MIN_SHARES (1e18)` reverts with MinSharesViolation. Shares are quoted by the standard OZ ERC4626 formula `assets * (totalSupply + 1) / (totalAssets + 1)`, and `totalAssets()` is `balanceOf(asset) - unvested`, so a plain ERC20 transfer of [redacted] into the vault counts immediately. While `totalSupply() == 0` an attacker can therefore donate an arbitrarily small amount X and force every subsequent depositor to bring at least `1e18 * X` wei of assets to clear the floor. Nothing can undo it: `rescueTokens` explicitly refuses `asset()`, and `burnAssets` reverts unless `balance - amount >= 1 ether`, so with a sub-1-ether balance no burn amount is legal. Only a UUPS upgrade recovers the vault. The zero-supply state is reachable three ways: (a) at deployment — `deploy/01-FullSystem.ts` never seeds the vault; (b) after the staker base exits; (c) after an admin `redistributeLockedAmount(sole staker, address(0))` compliance burn, which also strands the seized assets. Separately, `unstake` deliberately skips `_checkMinShares` (the `_unstaking` flag at :448/:699), so a single holder can drive totalSupply into (0, 1e18) from a live vault and then donate, reaching the same state without waiting for a full exit.

Impact. Permanent, unrecoverable-without-upgrade denial of service on the staking product: no new [redacted]+ can ever be minted, and every cross-chain stake routed through [redacted]+ Composer refunds. Existing stakers can still exit. Attacker cost is ~1e-6 [redacted] plus gas. No funds are stolen — but in the compliance-burn variant the seized assets (10,000e18 in the executed PoC) are permanently stranded in the vault.

Recommendation. Seed the vault atomically at deployment by minting MIN_SHARES of dead shares to address(0)/the protocol (this is the standard fix and removes case (a) and (b) entirely), and either raise `_decimalsOffset()` so donations cannot move the rate materially, or replace the hard MIN_SHARES revert with a floor that is only enforced when totalSupply was already non-zero. Also make `unstake` respect the floor or, better, allow deposits whenever totalSupply()==0 regardless of the asset balance.

PROOF OF CONCEPT — EXECUTED

[redacted]/contracts/test/vari_redo/PlusMinSharesDoS.t.sol::test_VR1_DustDonationBricksAllDeposits, ::test_VR2_UnstakeBypassesMinSharesThenDonationBricks, [redacted]/contracts/test/vari_redo/PlusVaultAndRoles.t.sol::test_VR14b_ComplianceBurnOfSoleStakerBri

```
[PASS] test_VR1_DustDonationBricksAllDeposits() (gas: 220197)\nLogs:\n  shares a 1,000,000 [redacted] deposit would mint: 999999999999\n  [redacted] (wei) needed to mint MIN_SHARES: 1000000000001000000000000000000\n\n[PASS] test_VR2_UnstakeBypassesMinSharesThenDonationBricks() (gas: 349751)\nLogs:\n  totalSupply after unstake: 1\n  shares for a 1,000,000 [redacted] deposit: 199999999960000\n\n[PASS] test_VR14b_ComplianceBurnOfSoleStakerBricksTheVault() (gas: 298873)\nLogs:\n  assets left with zero shares: 100000000000000000000\n  shares for a 1,000,000 [redacted] deposit: 99
```

PROTO-12 · **Medium**

[redacted] Composer does not validate the cross-chain beneficiary, paying out to a hub-blacklisted address

LOCATION

Confirmed, and the asymmetry with the sibling contract is the proof. [redacted] Composer.lzCompose ([redacted] Composer.sol:495-

`[redacted] Composer.lzCompose` ([redacted] Composer.sol:495-549) validates the LayerZero endpoint, the compose sender and the whitelisted-OFT set, then dispatches to the deposit or the redeem leg. The only compliance checks in the contract run against the message originator -- `hasValidCredentials(originalSender)` and `isBlacklisted(originalSender)` at :341-346 for deposit and :417-422 for redeem, where `originalSender` is decoded from `composeFrom`. The destination beneficiary carried in `sendParam.to` is never inspected on either leg; on the redeem leg it is used raw at :449-455 as the recipient of the collateral transfer. The sibling contract does check it: `[redacted]+ Composer.lzCompose` ([redacted]+ Composer.sol:99-108) decodes the beneficiary and reverts `BeneficiaryRestricted(beneficiary)` before dispatch. Reaching the gap requires only a well-formed compose message from an originator in good standing that names a restricted address as beneficiary -- no privileged role, no malformed payload.

Impact. A sanctioned or blacklisted address that cannot hold or receive [redacted] on the hub is still paid out through the composer: a redeem compose from a clean originator delivers 100,000e6 USDC directly to an address blacklisted on the hub [redacted], while the identical message shape is rejected by the [redacted]+ composer. What leaves the system is raw collateral, which carries no protocol-level restriction of its own, so the funds are unrecoverable once transferred. The originator still supplies the value, so this is a compliance-evasion channel rather than a theft of protocol funds -- but it defeats the restriction on precisely the address the protocol intended to cut off.

Recommendation. Port the `[redacted]+ Composer.lzCompose` guard verbatim into `[redacted] Composer.lzCompose`: decode `sendParam.to` before dispatch and revert `BeneficiaryRestricted(beneficiary)` when the beneficiary is blacklisted or lacks valid credentials, on both the deposit and the redeem leg. On the redeem leg place it ahead of the collateral transfer at [redacted] Composer.sol:449-455, since that is where raw USDC leaves the system carrying no protocol-level

restriction of its own. Fail into the existing de-whitelist refund path rather than reverting the compose outright, so a bad beneficiary returns the funds to the originator instead of stranding the message.

PROOF OF CONCEPT — EXECUTED

test/vari_verify/PriorFindingXC01.t.sol::test_XC01_[redacted] Composer_PaysOutToBlacklistedBeneficiar
::test_XC01_PlusComposer_RejectsBlacklistedBeneficiary,
::test_XC01_NoBeneficiaryCredentialCheckAnywhereIn[redacted] Composer

```
[PASS] test_XC01_[redacted] Composer_PaysOutToBlacklistedBeneficiary() (gas: 287076)
  Logs:
    USDC delivered to the BLACKLISTED beneficiary: 100000000000
[PASS] test_XC01_NoBeneficiaryCredentialCheckAnywhereIn[redacted] Composer() (gas: 250094)
  Logs:
    originator check present, beneficiary check absent
[PASS] test_XC01_PlusComposer_RejectsBlacklistedBeneficiary() (gas: 74018)
  Logs:
    [redacted]+ composer has the guard; [redacted] composer does not
```

PROTO-13 · **Medium**

Raising the vesting period re-locks already-vested rewards and drops the share price, irreversibly

LOCATION `[redacted]+.setVestingPeriod ([redacted]+.sol:551-561) guards with `if (getUnvestedAmount() > 0) revert StillVesting();``

`[redacted]+.setVestingPeriod` ([redacted]+.sol:552-564) refuses to change the period mid-schedule with `if (getUnvestedAmount() > 0) revert StillVesting();`, but that guard is evaluated against the old `vestingPeriod`, and neither `vestingAmount` nor `lastDistributionTimestamp` is ever cleared when a schedule completes. `getUnvestedAmount()` (:592-614) recomputes purely from those two residual values against whatever `vestingPeriod` currently holds, returning 0 once `block.timestamp - lastDistributionTimestamp >= vestingPeriod`. So once an 8-hour window has elapsed the guard reads zero and passes, and writing `vestingPeriod = 90 days` immediately re-amortises the same, already-distributed `vestingAmount` over the new window, which `totalAssets()` (:584-586) then subtracts a second time. The resulting state cannot be undone in place: lowering the period again reverts `StillVesting`, and `transferInRewards` reverts too because `_updateVestingAmount` (:742-746) carries the same guard. The call is `onlyRole(DEFAULT_ADMIN_ROLE)`.

Impact. One admin transaction, moving no tokens, re-locks rewards that have already vested and drops the share price for every holder: with 100,000e18 of fully-vested rewards the price falls from 1.099999999999999999 to 1.000370383230452674, 99,629e18 is re-locked, and `totalAssets` drops from 1,100,000e18 to 1,000,370e18. The vault cannot be restored -- the period cannot be lowered and no new rewards can be pushed until the full 90 days elapse -- so anyone who redeems in the interim realises the loss permanently. The trigger requires `DEFAULT_ADMIN_ROLE`, so this is an unrecoverable operator footgun rather than an externally reachable exploit; the price drop is nonetheless visible in the public mempool, and holders who exit ahead of it leave the loss with those who do not.

Recommendation. Settle the finished schedule instead of leaving it resident: once `block.timestamp >= lastDistributionTimestamp + vestingPeriod`, zero `vestingAmount` (and stamp `lastDistributionTimestamp`) so `getUnvestedAmount()` and `totalAssets()` cannot re-amortise a distribution that has already been paid out. The minimal patch is to clear both fields inside `setVestingPeriod` ([redacted]+.sol:551-561) after the `StillVesting` check passes and before the new

period is written; the structural fix is to do the clearing in `_updateVestingAmount` / a shared `_settleVesting()` called by every entry point, so no caller can ever observe the stale pair. Take the second -- the first leaves the same residue readable by `totalAssets` between a window ending and the next admin call.

PROOF OF CONCEPT — EXECUTED

test/vari_verify/PriorFindingsCore.t.sol::test_NSP01_IncreasingVestingPeriodRelocksVestedRewards
::test_NSP01_ReLockAlsoBricksRewardsAndCannotBeUndone

```
[PASS] test_NSP01_IncreasingVestingPeriodRelocksVestedRewards() (gas: 329879)
  Logs:
    share price BEFORE setVestingPeriod: 1099999999999999999
    RE-LOCKED (was vested, now unvested): 99629616769547325102880
    share price AFTER  setVestingPeriod: 1000370383230452674
    totalAssets before: 110000000000000000000000
    totalAssets after : 100037038323045267489712 0
[PASS] test_NSP01_ReLockAlsoBricksRewardsAndCannotBeUndone() (gas: 338643)
  Logs:
    vault stuck: no reward pushes and no period change until 90d elapse
```

PROTO-N03 · **Medium**

[redacted]+ Composer's M-3/rewards-composer-1 restriction checks live only on lzCompose; the inherited same-chain depositAndSend lets a FULL_RESTRICTED_STAKER_ROLE address stake and route [redacted]+ to a fresh address — the exact scenario the fix names

LOCATION [redacted]/contracts/contracts/[redacted]-plus/[redacted]+ Composer.sol:77 (lzCompose override) vs the un-overridden VaultComposerSync.depositAndSend/redeemAndSend inherited into the same contract

[redacted]+ Composer overrides lzCompose specifically to add two restriction checks, and the NatSpec states the threat model verbatim: 'the bare base composer checks neither originator nor (effectively) destination for the staker, so a blacklisted user could stake cross-chain and route the resulting [redacted]+ to a fresh address, escaping FULL_RESTRICTED_STAKER_ROLE' (M-3), plus 'also reject a hub-blacklisted destination BENEFICIARY' (rewards-composer-1). The contract inherits, and does not override or disable, VaultComposerSync.depositAndSend(uint256,SendParam,address) and redeemAndSend(uint256,SendParam,address) — same-chain entrypoints that reach the identical internal _depositAndSend/_redeemAndSend with no originator check and no beneficiary check. [redacted]+'s own _deposit blacklist check cannot help: it sees caller == receiver == the composer, never the user. Because [redacted] and [redacted]+ keep separate restriction lists on separate contracts, an address on the [redacted]+ list is free to hold and move [redacted], call composer.depositAndSend on the hub, and have the minted [redacted]+ delivered to any address on any chain. The sibling composer does not have this hole: [redacted] Composer's local entrypoint depositCollateralAndSend routes through _depositCollateralAndSend, which checks hasValidCredentials and isBlacklisted on msg.sender. So the same protection is present on one composer's local path and absent on the other's.

Impact. Complete bypass of the [redacted]+ compliance/restriction list for the deposit direction, and of the beneficiary check in both directions, from the hub chain. A sanctioned or frozen staker can re-enter the vault at an unrestricted address at will; the interim operational mitigation the runbook recommends

(blacklist offenders on the spoke [redacted] OFT) does not cover the hub-local path either. No funds are stolen, but the control the M-3 remediation was deployed for does not hold, and the composers are non-upgradeable — closing it requires another composer redeploy plus a Stargate/compose-target re-point.

Recommendation. Override depositAndSend and redeemAndSend on [redacted]+ Composer to run the same originator (msg.sender) and beneficiary (sendParam.to) restriction checks lzCompose performs — or, better, hoist the checks into overrides of the internal _depositAndSend/_redeemAndSend so every present and future entrypoint inherits them. Enumerate the composer's full inherited ABI and explicitly account for every entry before redeploy.

> PROOF OF CONCEPT — EXECUTED
>
> [redacted]/contracts/test/vari_novel/NovelComposerBypass.t.sol::test_N09_PlusComposer_DepositAndSend_
>
> (and ::test_N09b for the sibling contrast)

```
[PASS] test_N09_PlusComposer_DepositAndSend_BypassesRestrictionList() (gas: 325603)
  [redacted]+ delivered to fresh address: 500000000000000000
Sequence asserted in one test: [redacted]Plus.addToBlacklist(alice) -> [redacted]Plus.deposit(500e18, fresh) from
alice REVERTS; a well-formed lzCompose from the endpoint with composeFrom==alice REVERTS
(OriginatorRestricted); [redacted]PlusComposer.depositAndSend(500e18, sendParam{to: fresh}, alice) SUCCEEDS
and mints 500e18 [redacted]+ to the fresh address.
[PASS] test_N09b_[redacted] Composer_LocalEntrypoint_DoesEnforce() — the sibling composer's local entrypoint
reverts UnauthorizedSender for a blacklisted caller, and its inherited depositAndSend/redeemAndSend are
inert.
```

PROTO-N04 · **Medium**

[redacted].cancelRedeem is whenNotPaused, trapping queued-redemption escrow during a pause — while the same codebase deliberately exempted [redacted]+.cancelCooldown for exactly this reason and completionsAllowedWhilePaused lets the solver keep settling through the same pause

LOCATION

```
[redacted]/contracts/contracts/[redacted]/[redacted].sol:666 (cancelRedeem), :616
(updateRedemptionRequest), :727 (setCompletionsAllowedWhilePaused) vs
[redacted]/contracts/contracts/[redacted]-plus/[redacted]+.sol:512 (cancelCooldown)
```

The two escrow silos in this system are structurally identical — a user commits tokens to a silo, and a single function returns them 1:1 with no price read and no protocol asset outflow. [redacted]+.cancelCooldown carries a 14-line justification for being pause-exempt ('An emergency pause must block NEW risk … but not a user retrieving their OWN already-committed shares … C4 2023-10-ethena #511 ruled trapping committed cooldown/silo funds a real vulnerability'). [redacted].cancelRedeem does exactly the same thing for the redeem silo and is whenNotPaused. Its own NatSpec contradicts the modifier: 'This function always works regardless of asset support status, providing an escape hatch for users' and 'Users can cancel at any time. This is by design.' updateRedemptionRequest (the partial-decrease path) is likewise pause-gated. The asymmetry is compounded by the V2 escape hatch in the opposite direction: setCompletionsAllowedWhilePaused(true) lets COLLATERAL_MANAGER_ROLE keep completing queued redemptions during the same pause, in whatever order it chooses (the contract itself documents that completion order is 'discretionary by the COLLATERAL_MANAGER_ROLE (solver)', not FIFO). So a pause can be configured such that the protocol keeps settling selected users at live prices while no user can withdraw their escrow or reduce their exposure.

Impact. During any pause, [redacted] escrowed in the redeem silo is unrecoverable by its owner for the pause's duration, and the position cannot even be resized. Because completions remain enabled at the

solver's discretion, the pause becomes a one-sided window: users are locked into a redemption at whatever price/fee configuration exists when the solver chooses to act (fees, minRedeemFeeAmount and instantRedeemFeeBps are all admin-settable during the pause), with no exit. This is the precise failure mode the project cites C4 2023-10-ethena #511 for and fixed on the other silo.

Recommendation. Remove whenNotPaused from cancelRedeem (keeping nonReentrant and the blacklist check, which is what actually blocks a restricted user via the silo transfer) so it matches cancelCooldown and matches its own NatSpec; at minimum, gate it on the inverse of completionsAllowedWhilePaused so users can always exit whenever the protocol can still settle. Consider the same treatment for the decrease branch of updateRedemptionRequest.

PROOF OF CONCEPT — EXECUTED

[redacted]/contracts/test/vari_novel/NovelCore.t.sol::test_N06_PauseTrapsRedeemEscrow_ButNotCooldownE

```
[PASS] test_N06_PauseTrapsRedeemEscrow_ButNotCooldownEscrow() (gas: 618689)
Asserted in one test: after [redacted].pause() + [redacted]Plus.pause(), [redacted].cancelRedeem() REVERTS
(EnforcedPause) and [redacted].updateRedemptionRequest(500e18) REVERTS, while
setCompletionsAllowedWhilePaused(true) + completeRedeem(alice) SUCCEEDS with a non-zero payout, and
[redacted]Plus.cancelCooldown() SUCCEEDS and returns the escrowed shares.
```

PROTO-N05 · **Medium**

Both silos are the only Ownable contracts in the system without the renounceOwnership() lockout override the changelog claims covers them; renouncing permanently kills setVault/setToken, lockWiring and the UUPS upgrade authorisation on the contracts that escrow user funds

LOCATION

[redacted]/contracts/contracts/[redacted]/[redacted]RedeemSilo.sol:15 and
[redacted]/contracts/contracts/[redacted]-plus/[redacted]+Silo.sol:15
(Ownable2StepUpgradeable inherited, renounceOwnership not overridden)

CHANGES_POST_AUDIT.md lists the ownership safeguard under 'Files: [redacted], [redacted], [redacted]+, all OFTs and adapters, both silos' and states 'renounceOwnership is disabled on the Ownable OFTs/adapters ... Prevents an accidental last-admin lockout.' [redacted] OFT, [redacted]+ OFT, [redacted]OFTAdapter, [redacted] OFTAdapter, [redacted]+ OFTAdapter and StakingRewardsDistributor all carry `function renounceOwnership() public view override onlyOwner { revert CantRenounceOwnership(); }`. The two silos — the only remaining Ownable contracts, and the only ones that actually hold escrowed user tokens — do not. Both expose a live renounceOwnership() in their deployed ABI. On the silos the consequence is strictly worse than on the OFTs, because the owner is simultaneously the wiring authority (setVault/setNaraUsd, setStakingVault/setToken), the finalizer (lockWiring), and the UUPS upgrade authority (_authorizeUpgrade is onlyOwner): renouncing removes all four at once with no recovery path of any kind. The deploy runbook makes this reachable — lockWiring() is an explicit post-upgrade step still marked as pending ('it is unlocked by default after the upgrade; the fix is only armed once called'), so the window in which the wiring is not yet final and the owner can be lost is the live window. Related observation from the same read: lockWiring's promise that wiring 'can no longer be repointed ... One-way' is only true against the setters; while the owner exists it is one _authorizeUpgrade away from being undone, so the flag is a convention rather than an enforcement.

Impact. If ownership is renounced (accidentally, by a script that mirrors the 'clean up deployer privileges' step used on other contracts, or by a compromised key acting destructively) the silo is frozen forever:

wiring cannot be corrected, lockWiring cannot be armed, and the contract can never be upgraded. Every future [redacted]/[redacted]+ implementation is permanently bound to the silo address recorded at that moment; if the vault side ever has to be re-pointed, all escrowed queued-redemption [redacted] and all escrowed cooldown [redacted]+ become unreachable. The silos are the escrow contracts, so this is the worst place in the system to lose governance.

Recommendation. Add the same `renounceOwnership() public view override onlyOwner { revert ... }` to [redacted]RedeemSilo and [redacted]+Silo — they are UUPS-upgradeable, so this ships in the already-planned silo upgrade with no redeploy. Separately, either make lockWiring also freeze upgrades (or move the silos behind a timelocked proxy admin) or soften the 'permanently / one-way' wording so operators do not treat wiringLocked as a hard guarantee.

PROOF OF CONCEPT — EXECUTED

[redacted]/contracts/test/vari_novel/NovelCore.t.sol::test_N03_SilosCanRenounceOwnership_UnlikeEveryS

```
[PASS] test_N03_SilosCanRenounceOwnership_UnlikeEverySibling() (gas: 1243573)
Asserted: wiringLocked()==false; redeemSilo.renounceOwnership() SUCCEEDS -> owner()==address(0);
setVault, lockWiring and upgradeToAndCall all then REVERT; identical result for plusSilo
(renounceOwnership, then setStakingVault and lockWiring revert). ABI enumeration via forge inspect
confirms renounceOwnership() present on both silos and absent from the five OFTs/adapters' reachable
surface (reverting override).
```

PROTO-P0095 · **Medium**

[redacted].cancelRedeem / updateRedemptionRequest are pause-gated, so a GATEKEEPER pause traps [redacted] the user has already escrowed in the redeem silo

LOCATION

[redacted]/contracts/contracts/[redacted]/[redacted].sol:666 (cancelRedeem, `external nonReentrant whenNotPaused`) and :616 (updateRedemptionRequest); contrast [redacted]/contracts/contracts/[redacted]-plus/[redacted]+.sol:512 (cancelCooldown, deliberately NOT whenNotPaused, rationale at :500-511)

A queued redemption escrows the user's [redacted] 1:1 in [redacted]RedeemSilo. cancelRedeem() is the only unconditional way back out, and it carries whenNotPaused. The team already reasoned this exact question on the sibling contract: [redacted]+.cancelCooldown carries a 13-line justification ([redacted]+.sol:499-511, '[redacted]-plus-2') for deliberately NOT gating it -- 'This path moves no protocol assets and reads no price -- it returns escrowed [redacted]+ 1:1 ... so gating it only traps the user's own funds', citing C4 2023-10-ethena #511 that trapping committed cooldown/silo funds is a real vulnerability. [redacted].cancelRedeem is the identical shape (returns escrowed [redacted] 1:1 from a silo, moves no collateral, reads no oracle) and was left pausable. The completionsAllowedWhilePaused escape hatch does not close the gap: it only lets the COLLATERAL_MANAGER solver push completions, which additionally require live collateral liquidity and pass through the fee and per-block caps -- it is not a user-controlled exit. [redacted] transfers themselves are NOT pause-gated ([redacted].sol:1614 _update has no _requireNotPaused), so the pause is not even protecting a frozen ledger: everyone who did NOT queue keeps full mobility while the queued users alone are frozen.

Impact. For the whole duration of a pause -- which for this system is the documented response to a sustained collateral depeg, i.e. exactly when a user most wants their [redacted] back -- every user with a queued redemption is unable to recover their escrowed principal, while unqueued holders can transfer freely. The trap also compounds two other admin-config paths: lowering maxRedeemPerBlock below a live

request, or removing the requested collateral from [redacted], both make completion impossible and leave cancelRedeem as the only exit -- which the pause closes.

Recommendation. Drop whenNotPaused from cancelRedeem (keep nonReentrant and the blacklist check), matching [redacted]+.cancelCooldown and its written rationale. If a pause must be able to freeze the queue, add a separate, narrower flag rather than reusing the global pause, and document the divergence from the sibling contract either way.

PROOF OF CONCEPT — EXECUTED

[redacted]/contracts/test/vari_sweep/SweepTriage.t.sol::SweepTriage::test_PAT0095_cancelRedeem_isTrap

```
[PASS] test_PAT0095_cancelRedeem_isTrappedByPause() (gas: 399343)
```

PROTO-03 · Low

[redacted].redeem has no zero-output guard: sub-unit redemptions burn [redacted] and [redacted] for zero collateral

LOCATION [redacted]/contracts/contracts/[redacted]/[redacted].sol:371-397 (redeem); compare :226 (mint's ZeroMintAmount guard)

`[redacted].mint` explicitly rejects a deposit that would credit backing while minting zero [redacted] (`if ([redacted]Amount == 0) revert ZeroMintAmount()` — [redacted]-[redacted]-1). The redeem direction has no symmetric check. `_convertToCollateralAmount` rounds down into native decimals, so for 6-decimal collateral any value below 1e12 wei of [redacted] converts to 0 units; the balance check `collateralBalance < 0` passes trivially, `_burn(msg.sender, [redacted]Amount)` executes, and `safeTransfer(beneficiary, 0)` moves nothing. [redacted]'s `_instantRedeem` takes this path (the liquidity precondition `collateralBalance >= 0` is always satisfied), so the user's [redacted] is burned too. `minRedeemAmount` would prevent it but defaults to 0.

Impact. Value destruction with no counterpart payout. It is protocol-favourable (backing ratio improves, remaining holders gain) so it is self-grief rather than theft, but it is an unguarded asymmetry on a value-moving path and it silently breaks the caller's expectation that burning shares returns assets. Griefing a specific user is not possible; the loss is always the caller's own.

Recommendation. Mirror the mint-side guard: `if (collateralAmount == 0) revert InvalidAmount();` in `[redacted].redeem`, and consider a non-zero `minRedeemAmount` default in [redacted].

PROOF OF CONCEPT — EXECUTED [redacted]/contracts/test/vari_redo/[redacted].t.sol::test_VR5_DustRedeemBurnsWithoutPayout

```
[PASS] test_VR5_DustRedeemBurnsWithoutPayout() (gas: 344571)\nLogs:\n [redacted] destroyed for zero payout (wei): 999999999999\nassertions held: got == 0, wasQueued == false (instant path), user USDC balance unchanged, [redacted] supply -999999999999, [redacted] supply -999999999999, collateralBalance unchanged
```

PROTO-04 · Low

[redacted].burn retires unbacked [redacted] without crediting [redacted].unbackedMinted — the 'net outstanding' cap degrades into a lifetime cap

LOCATION [redacted]/contracts/contracts/[redacted]/[redacted].sol:987-997 (burn -> [redacted].burn) vs [redacted]/contracts/contracts/[redacted]/[redacted].sol:275 (burnUnbacked)

[redacted] documents `maxUnbackedMint` as a net-outstanding ceiling and provides `burnUnbacked(amount)` to free budget when unbacked supply is retired. But the only route by which unbacked [redacted] is actually destroyed in production — `[redacted].burn(amount)`, which calls the plain ERC20Burnable `[redacted].burn(amount)` on [redacted]'s own [redacted] balance — never touches `unbackedMinted`. `burnUnbacked` also requires MINTER_ROLE on [redacted] and burns from the caller, so it cannot be used to reconcile a burn that already happened inside [redacted]. Conversely `burnUnbacked` floors the counter at 0, so a MINTER burning collateral-backed [redacted] through it would free budget it never consumed.

Impact. The unbacked-mint budget is consumed permanently even when the corresponding supply is retired, so the timelocked cap tightens monotonically and governance is eventually forced into a 2-day `proposeRaiseMaxUnbackedMint` cycle it should not have needed. Safety-control drift, not a fund loss. The counter can also over-report outstanding unbacked supply, making any monitor built on it wrong.

Recommendation. Have `[redacted].burn` call a dedicated [redacted] entry point that both burns and decrements `unbackedMinted` (bounded by the outstanding amount), or track backed vs unbacked [redacted] explicitly so the counter cannot drift in either direction.

PROOF OF CONCEPT — EXECUTED [redacted]/contracts/test/vari_redo/[redacted].t.sol::test_VR10_UnbackedCounterDesyncOnBurn

```
[PASS] test_VR10_UnbackedCounterDesyncOnBurn() (gas: 110326)\nLogs:\n  unbacked budget permanently consumed despite the supply being retired\nassertions held: 1,000e18 unbacked [redacted] destroyed by [redacted].burn; [redacted].unbackedMinted() unchanged; the next mintWithoutCollateral(bob, 1) reverts ExceedsUnbackedMintCap
```

PROTO-05 · Low

[redacted]+ OFT._update omits the blacklisted-operator (msg.sender) check present on every other [redacted] token

LOCATION [redacted]/contracts/contracts/[redacted]-plus/[redacted]+ OFT.sol:109-119 (_update)

[redacted] (_update, [redacted].sol:1605), [redacted]+ (_update, [redacted]+.sol:744) and [redacted] OFT (_update, [redacted] OFT.sol:125) all reject a transfer whose `msg.sender` holds the restriction role, specifically to stop a blacklisted operator from moving an approved holder's tokens via `transferFrom`. [redacted]+ OFT — the spoke-side [redacted]+ token — checks only `from` and `to`. A restricted address that already holds an allowance from a non-restricted holder can therefore still execute `transferFrom(holder, freshAddress, amount)` on the spoke.

Impact. A sanctioned/blacklisted party retains the ability to act as an operator over other users' spoke [redacted]+ and route it to an address that is not on the list. It does not let them receive tokens directly (the `to` check still applies) and does not move their own frozen balance, so this is a compliance-control gap rather than a theft primitive.

Recommendation. Add `if (from != msg.sender && hasRole(FULL_RESTRICTED_STAKER_ROLE, msg.sender)) revert OperationNotAllowed();` to `[redacted]+ OFT._update`, matching the other three tokens.

PROOF OF CONCEPT — EXECUTED

[redacted]/contracts/test/vari_redo/CrossChainAttacks.t.sol::test_CC10_PlusSpokeOftMissesBlacklistedO

```
[PASS] test_CC10_PlusSpokeOftMissesBlacklistedOperatorCheck() (gas: 1719328)\nLogs:\n  [redacted]+ OFT._update omits the msg.sender restriction check\nassertions held: [redacted]Oft.transferFrom by the blacklisted operator REVERTS; [redacted]PlusOft.transferFrom by the same operator SUCCEEDS and bob receives 1e18
```

PROTO-06 · Low

burnQuarantine breaks the hub-locked == aggregate-spoke-supply invariant with no on-chain reconciliation

LOCATION [redacted]/contracts/contracts/[redacted]/[redacted] OFT.sol:186 and [redacted]/contracts/contracts/[redacted]-plus/[redacted]+ OFT.sol:161 (burnQuarantine)

The lockbox model requires `[redacted].balanceOf([redacted] OFTAdapter) == sum of spoke OFT totalSupply`. `burnQuarantine` destroys spoke supply while the matching [redacted] stays locked in the hub adapter. The contract comment acknowledges this and declares reconciliation an 'operational requirement', but there is no hub-side function that unlocks or burns the orphaned escrow, and no on-chain link between the two actions. Release-vs-burn are correctly mutually exclusive (both zero `quarantinedBalance` before acting) and the quarantine sink accounting itself is exact.

Impact. After any seizure the hub lockbox holds [redacted] that no spoke share can ever claim. It is not a solvency hole in the user-facing direction (the excess is on the safe side), but any monitor comparing hub escrow to aggregate spoke supply diverges permanently, and the stranded [redacted] is unrecoverable without an upgrade to the adapter. Admin-only trigger.

Recommendation. Pair the seizure with an on-chain hub action — e.g. a `seizeLocked(uint256)` on the adapter callable only by the same governance address, or emit a structured event that a hub keeper consumes — and add a public `reconciliationDeficit()` view so monitoring can assert the invariant instead of relying on a runbook.

PROOF OF CONCEPT — EXECUTED

[redacted]/contracts/test/vari_redo/CrossChainAttacks.t.sol::test_CC5_BurnQuarantineDesyncsHubEscrow

```
[PASS] test_CC5_BurnQuarantineDesyncsHubEscrow() (gas: 812612)\nLogs:\n  orphaned [redacted] stranded in the hub lockbox: 10000000000000000000000\nassertions held: before the burn [redacted].balanceOf(adapter) == [redacted]Oft.totalSupply(); after burnQuarantine spoke totalSupply == 0 while the adapter still holds 10,000e18; a later releaseQuarantine reverts NothingQuarantined (mutual exclusion intact)
```

PROTO-07 · Low

Per-block mint cap under-counts when a collateral trades above $1 (fee taken at the real price, credited at the clamped price)

LOCATION [redacted]/contracts/contracts/[redacted]/[redacted].sol:1076-1081 and :1707-1717 (_mintWithCollateral)

`_mintWithCollateral` computes `[redacted]Amount` with [redacted]'s min(price,$1) mint clamp, derives `expectedMintAmount = [redacted]Amount - feeAmount18`, checks that against `maxMintPerBlock` and adds exactly that to `mintedPerBlock`. It then converts the fee to collateral units at the REAL price (`[redacted]ToCollateral`, i.e. fee/price), transfers that to the treasury, and re-values the remainder through the clamp. When price > $1 the fee removed in collateral is worth less than `feeAmount18` at the clamped

valuation, so `[redacted].mint` returns more than `expectedMintAmount` and more [redacted] enters circulation than was counted against the cap. The over-shoot is `feeAmount18 * (1 - 1/price)`, bounded by the 10% max fee and the $2.00 sanity ceiling at ~5% of the deposit value.

Impact. The per-block mint rate limit — a safety control — is not exact and can be exceeded by up to a few percent per transaction whenever a collateral trades above par. No direct loss; the excess [redacted] is fully [redacted]-backed. It does mean an emergency mint throttle is slightly leaky exactly during the abnormal price conditions it exists for.

Recommendation. Move the cap accounting to the actual minted amount: increment `mintedPerBlock` with the `[redacted]Amount` returned by `[redacted].mint` and re-check the ceiling after the mint, rather than pre-charging the estimate.

> PROOF OF CONCEPT — EXECUTED [redacted]/contracts/test/vari_redo/[redacted].t.sol::test_VR9_MintPerBlockCapUnderCounted
>
> ```
> [PASS] test_VR9_MintPerBlockCapUnderCounted() (gas: 302328)\nLogs:\n  actually minted: 904761910000000000\n  counted against the per-block cap: 900000000000000000\n  under-count (wei): 476191000000000000
> ```

PROTO-08 · **Low**

Silo wiring is never locked by the deploy scripts, leaving a one-call path to seize every escrowed balance

LOCATION	[redacted]/contracts/contracts/[redacted]-plus/[redacted]+Silo.sol:68 (setStakingVault) / :75 (lockWiring); [redacted]/contracts/contracts/[redacted]/[redacted]RedeemSilo.sol:68 (setVault) / :75 (lockWiring); deploy/01-FullSystem.ts

Both silos expose `setVault`/`setStakingVault`, gated only by Ownable2Step, and a one-way `lockWiring()` that freezes them. `grep -rl lockWiring` over deploy/, scripts/ and tasks/ returns no call site — the deploy pipeline sets the wiring and stops, so the setters stay live indefinitely. Re-pointing the vault to an arbitrary address makes `withdraw(to, amount)` callable by that address, draining every user's escrowed [redacted] (redeem queue) or [redacted]+ (cooldown) in a single transaction with no timelock, no pause interaction and no event that names it as a seizure.

Impact. Privileged, un-timelocked access to user escrow. In fairness this does not exceed what the same governance key already has through UUPS `_authorizeUpgrade` on every contract in the system, so the marginal risk is the absence of the mitigation the code was written to provide, not a new capability. It is still the shortest and least visible path to user funds.

Recommendation. Call `lockWiring()` on both silos as the last step of `04-Handover.ts` and assert `wiringLocked() == true` in `PostDeployValidation`. Longer term, put the UUPS admin behind a timelock so the upgrade path is not an equivalent bypass.

> PROOF OF CONCEPT — EXECUTED [redacted]/contracts/test/vari_redo/PlusVaultAndRoles.t.sol::test_VR18_AdminLeversHaveNoTimelock
>
> ```
> [PASS] test_VR18_AdminLeversHaveNoTimelock() (gas: 271474)\nassertions held: plusSilo.wiringLocked() == false and redeemSilo.wiringLocked() == false after the standard init sequence; after setStakingVault(attacker) a single silo.withdraw moved alice's full 1,000e18 cooldown escrow to carol
> ```

PROTO-14 · Low

A compliance seize outside a live reward window is a sandwichable share-price jump

LOCATION	Confirmed, but only on one of the two branches. [redacted]+.redistributeLockedAmount ([redacted]+.sol:291-395) burns the

`[redacted]+.redistributeLockedAmount` ([redacted]+.sol:291-395) burns a restricted holder's shares and, on the `to == address(0)` compliance branch, credits the seized assets with `vestingAmount += [redacted]ToVest` at :379 without touching `lastDistributionTimestamp`. `getUnvestedAmount()` returns 0 whenever `lastDistributionTimestamp == 0` or the previous window has already elapsed, so when no reward window is live the seized value is not amortised at all -- it lands in `totalAssets()` in the same block. The inline comment at :348-365 enumerates this as an accepted case and asserts there is no MEV or sandwich vector because the call is admin-only, which does not follow: an admin transaction is still a public mempool transaction, and the share-price jump it produces is observable before it lands. The other branch behaves correctly -- when a window is live the seized value joins that schedule and the price does not move -- so the defect is specific to the no-live-window path.

Impact. A seize executed outside a live reward window is an instantaneous, publicly predictable share-price jump: with two equal 1,000,000e18 stakers, seizing one moves the price from 1.000000000000000000 to 1.999999999999999999 in a single transaction while `getUnvestedAmount()` remains 0. With `cooldownDuration` at 0 that is an atomic sandwich -- an address depositing 1,000,000e18 immediately before the seize redeems 1,999,990e18 immediately after, taking 999,990e18 of risk-free profit from the honest stakers the seizure was meant to benefit. The default deployment sets `cooldownDuration = 7 days` ([redacted]+.sol:190), which forecloses the atomic version; the residual is a front-run followed by a seven-day locked exit, still profitable but carrying real price risk over the cooldown.

Recommendation. On the `to == address(0)` branch of `redistributeLockedAmount`, set `lastDistributionTimestamp = block.timestamp` alongside the `vestingAmount += [redacted]ToVest` at [redacted]+.sol:379, so a seize always opens a fresh vesting window rather than landing in `totalAssets()` in the same block when no window is live. If seized value is instead meant to accrue immediately to remaining holders, then the call must not be a public-mempool transaction -- route it through a private relay or a commit-reveal, and enforce a non-zero `cooldownDuration` as a deployment invariant. I would take the first: it makes the outcome independent of configuration. Either way, delete the comment at :348-365 asserting there is no MEV or sandwich vector because the call is admin-only.

PROOF OF CONCEPT — EXECUTED

test/vari_verify/PriorFindingsCore.t.sol::test_NSP02_SeizeOutsideVestingWindowIsAnInstantPriceJu
::test_NSP02_SandwichWithCooldownOff, ::test_NSP02_SeizeDuringActiveWindowDoesVest

```
[PASS] test_NSP02_SeizeOutsideVestingWindowIsAnInstantPriceJump() (gas: 295078)
  Logs:
    share price before seize: 1000000000000000000
    share price after  seize: 1999999999999999999
[PASS] test_NSP02_SandwichWithCooldownOff() (gas: 365390)
  Logs:
    carol in : 1000000000000000000000000
    carol out: 1999990000099999000009998
    carol profit: 999990000099999000009998
[PASS] test_NSP02_SeizeDuringActiveWindowDoesVest() (gas: 406069)
  Logs:
    unvested after seize (window live): 100100000000000000000000
    price before: 1000000000000000000
    price after : 1000000000000000000
```

PROTO-N06 · **Low**

[redacted] per-block mint cap under-counts whenever a collateral trades above $1: the fee is deducted in collateral at the real price while the credit uses the min(price,$1) clamp, so more [redacted] is minted than is charged against maxMintPerBlock

LOCATION [redacted]/contracts/contracts/[redacted]/[redacted].sol:1075-1081 and :1091-1107 (_mintWithCollateral)

_mintWithCollateral computes the value of the deposit through the mint clamp (`[redacted]Amount = collateralTo[redacted](asset, collateralAmount)` = normalized * min(price,$1)), derives the fee from it, and charges `expectedMintAmount = [redacted]Amount - feeAmount18` against the per-block cap with the comment 'Track minted amount using post-fee amount (actual amount that will be minted)' / 'actual circulating supply increase'. It then converts the fee back to collateral at the REAL, unclamped price (`_convertToCollateralAmount` -> [redacted]ToCollateral), subtracts it, and re-values the remainder through the clamp again inside [redacted].mint. When price > $1 the two directions do not cancel: the fee costs feeAmount18/price collateral, so the re-valued remainder is [redacted]Amount - feeAmount18/price, which is strictly greater than expectedMintAmount. The amount actually minted therefore exceeds the amount recorded in mintedPerBlock, and the cap check that ran before the mint used the smaller number. The gap is feeAmount18 * (price-1)/price — at the documented maximum mint fee of 10% and a price at the top of the sanity band ($2.00) it is 5% of the deposit; at $1.10 it is ~1%.

Impact. maxMintPerBlock — the protocol's only per-block rate limit on minting, and one of the two levers the audit rounds rely on to bound an incident — can be exceeded by up to ~5% per block per depositor whenever an oracle-priced collateral reads above $1 (a routine condition for USDT/USDC feeds, which regularly print 1.0002-1.001). The same accounting gap means mintedPerBlock, maxDeposit() and maxMint() all understate real issuance, so any monitoring built on them is systematically low during exactly the conditions that matter.

Recommendation. Charge the cap against the value actually minted rather than a predicted one: move `_checkBelowMaxMintPerBlock` and the `mintedPerBlock` accumulation to after `[redacted]Amount = [redacted].mint(...)` and use [redacted]Amount, or compute expectedMintAmount with the same clamp-then-real-price composition the real path uses (as previewMintWithCollateral already does).

PROOF OF CONCEPT — EXECUTED

[redacted]/contracts/test/vari_novel/NovelCore.t.sol::test_N04_MintPerBlockCap_UnderCountsAbovePeg

```
[PASS] test_N04_MintPerBlockCap_UnderCountsAbovePeg() (gas: 312150)
  maxMintPerBlock: 900000000000000000000
  mintedPerBlock (tracked): 900000000000000000000
  actually minted: 909090910000000000000
(USDC feed at $1.10, mintFeeBps=1000, maxMintPerBlock=900e18, deposit 1000e6 -> 909.09091e18 [redacted] minted
in a single call against a 900e18 cap)
```

PROTO-N07 · **Low**

[redacted]+ OFT._update omits the blacklisted-operator guard its sibling [redacted] OFT and both hub tokens enforce, letting a restricted address keep spending pre-existing allowances

LOCATION [redacted]/contracts/contracts/[redacted]-plus/[redacted]+ OFT.sol:109-119 (_update) vs [redacted]/contracts/contracts/[redacted]/[redacted] OFT.sol:_update

The two spoke share-OFTs are otherwise line-for-line siblings (same quarantine mechanism, same roles, same pause). [redacted] OFT._update contains a third check absent from [redacted]+ OFT: `if (hasRole(FULL_RESTRICTED_ROLE, msg.sender)) revert OperationNotAllowed();`, commented 'Prevent blacklisted operators from moving tokens via transferFrom'. The same guard exists on [redacted]._update (':1624') and [redacted]+._update (':758'), so [redacted]+ OFT is the single place in the system where it is missing. A restricted address that already holds an allowance — typically a router, vault or aggregator contract the user approved before the restriction was applied — can still pull the victim's [redacted]+ on the spoke and route it to any unrestricted address. The same pair also diverges on input validation: [redacted] OFT's addToBlacklist/removeFromBlacklist reject address(0) and no-op changes, [redacted]+ OFT's do neither, so FULL_RESTRICTED_STAKER_ROLE can be granted to address(0) and no-op revocations pass silently.

Impact. A sanctioned or frozen operator retains the ability to move third-party [redacted]+ on spoke chains, defeating the operator half of the restriction model on one of the five deployed tokens. Low direct loss (it needs a pre-existing allowance and the destination must be unrestricted) but it is a compliance-control gap on a non-upgradeable contract, so closing it requires an OFT redeploy and holder migration.

Recommendation. Add the missing `if (hasRole(FULL_RESTRICTED_STAKER_ROLE, msg.sender)) revert OperationNotAllowed();` to [redacted]+ OFT._update and port the zero-address / ValueUnchanged guards from [redacted] OFT's blacklist setters, so the pair is byte-for-byte symmetric. Batch with the already-deferred Phase-4 OFT redeploy.

PROOF OF CONCEPT — EXECUTED

[redacted]/contracts/test/vari_novel/NovelOftDiff.t.sol::test_N08_PlusOFT_MissingBlacklistedOperatorG
and ::test_N08b_PlusOFT_BlacklistSettersUnvalidated

```
[PASS] test_N08_PlusOFT_MissingBlacklistedOperatorGuard() (gas: 262210)
[PASS] test_N08b_PlusOFT_BlacklistSettersUnvalidated() (gas: 53273)
Same victim/allowance/blacklist setup on both OFTs: usdOft.transferFrom(victim, sink, 100e18) from the
sanctioned operator REVERTS OperationNotAllowed, while plusOft.transferFrom(victim, sink, 100e18)
SUCCEEDS and sink ends with 100e18 [redacted]+. N08b: usdOft.addToBlacklist(address(0)) reverts
ZeroAddressException, plusOft.addToBlacklist(address(0)) succeeds.
```

PROTO-N08 · Low

[redacted].burn() documents a deflationary benefit that cannot exist: [redacted] redeems at oracle price, not pro-rata, so a burn is a pure transfer of the burner's collateral into the COLLATERAL_MANAGER's discretionary withdrawal lever

LOCATION [redacted]/contracts/contracts/[redacted]/[redacted].sol:987-997 (burn), and the same claim repeated at [redacted]/contracts/contracts/[redacted]-plus/[redacted]+.sol:219-223 (burnAssets NatSpec)

[redacted].burn is permissionless and its NatSpec says: 'This keeps the collateral in [redacted] but reduces [redacted] supply / Making remaining [redacted] more valuable (since same collateral backs fewer tokens).' That is true of a pro-rata vault and false of this one. [redacted] does not redeem pro-rata: [redacted].redeem pays `_convertToCollateralAmount(asset, [redacted]Amount)`, an absolute oracle-priced quantity that does not reference collateralBalance except as a sufficiency check, and [redacted] is pegged 1:1 to [redacted]. Burning therefore leaves every other holder's claim numerically identical and simply creates unclaimed collateral inside [redacted]. That surplus is not distributed, not tracked, and reachable only through withdrawCollateral (COLLATERAL_MANAGER_ROLE) — which OPEN_ITEMS.md itself records as 'the uncapped, no-timelock, by-design collateral lever'. So a user who calls burn() in reliance on the documented benefit hands 100% of the burned value to that lever. (The same sentence is copy-pasted into [redacted]+.burnAssets, where the real effect is the opposite of what it says: it lowers every staker's claim, as the adjacent line correctly states.) Secondary: burn() does not decrement [redacted]'s unbackedMinted counter, so retiring incentive supply through it silently consumes the unbacked-mint budget the C-1 cap is meant to meter — the runbook warns operators to use burnUnbacked instead, but nothing enforces it and [redacted]'s own path uses plain burn.

Impact. Any holder — or an integrator, or a UI that surfaces 'burn to make [redacted] more valuable' — who acts on the documentation destroys their own principal for zero protocol-wide benefit. No accounting anywhere records the resulting surplus, so it is invisible to totalCollateralValue()-based solvency monitoring as anything other than over-collateralisation, and it is extractable without timelock by the collateral manager.

Recommendation. Correct the NatSpec on both functions to state plainly that a burn transfers the burned value to protocol surplus with no pro-rata benefit to other holders, or remove the permissionless [redacted].burn entirely (nothing in the protocol calls it except [redacted]+.burnAssets, which could use a role-gated internal path). Additionally route [redacted]'s [redacted] burn through burnUnbacked when the supply being retired was unbacked, so the C-1 counter stays exact.

PROOF OF CONCEPT — EXECUTED

[redacted]/contracts/test/vari_novel/NovelCore.t.sol::test_N05_Burn_DonatesCollateralToManager_NotToH

```
[PASS] test_N05_Burn_DonatesCollateralToManager_NotToHolders() (gas: 480500)
  collateralBalance USDC (6d): 2000000000
  collateral-backed [redacted] outstanding (18d): 1500000000000000000000
Asserted: after alice burns 500e18, [redacted].collateralBalance(USDC) is UNCHANGED, [redacted].totalSupply() drops
by exactly 500e18, and bob's previewRedeem(USDC, 1000e18) is byte-identical to before the burn;
[redacted].withdrawCollateral(USDC, 500e6, X) then succeeds and bob still redeems 1000e18 -> 1000e6, i.e. the
burned value went entirely to the collateral manager.
```

PROTO-P0019 · Low

Collateral transferred directly to [redacted] is permanently immobilised -- [redacted] has no rescue path

LOCATION

[redacted]/contracts/contracts/[redacted]/[redacted].sol:406-443 (withdrawCollateral, bounded by collateralBalance) with no sweep/rescue function anywhere in the contract

[redacted] tracks collateralBalance[asset] and credits it only inside mint() (measured balance delta) and depositCollateral(). Every egress -- withdrawCollateral and redeem -- is hard-bounded by that counter. Tokens that reach [redacted] any other way (a user paying the vault address directly, a mis-targeted bridge delivery, an airdrop, an ERC20 with a rebase or a reflection distribution) increase the raw balanceOf without increasing collateralBalance, and there is no rescueTokens/sweep function on [redacted] at all. Every sibling contract in the system has one ([redacted]+.rescueTokens, [redacted] Composer.rescueTokens, [redacted]+ Composer.rescueTokens, StakingRewardsDistributor.rescueTokens); [redacted] -- the contract that actually custodies all protocol collateral -- does not.

Impact. Any such balance is lost forever, including to the protocol itself: it silently over-collateralises [redacted] without ever being creditable to a holder or reclaimable by governance. Executed PoC: 50,000 USDC transferred directly to [redacted] stays at [redacted] after governance evacuates 100% of the tracked balance, with no call that can move it.

Recommendation. Add an admin-gated sweep that can move only the excess (IERC20(asset).balanceOf(address(this)) - collateralBalance[asset]) for supported assets, and the full balance for non-supported tokens. Bounding it to the excess keeps the backing invariant intact while making stray value recoverable.

> PROOF OF CONCEPT — EXECUTED
>
> [redacted]/contracts/test/vari_sweep/SweepTriage.t.sol::SweepTriage::test_PAT0019_directTransferToMct
>
> ```
> [PASS] test_PAT0019_directTransferToMct_isPermanentlyStuck() (gas: 98761)
> ```

PROTO-P0084 · Low

No cross-field assertion between minRedeemAmount and maxRedeemPerBlock -- one admin write bricks every redemption path

LOCATION

[redacted]/contracts/contracts/[redacted]/[redacted].sol:823 (setMinRedeemAmount, only checks itself against minRedeemFeeAmount) and :700 -> :1167 (setMaxRedeemPerBlock / _setMaxRedeemPerBlock, no bounds at all); consumed at :399 (min check), :1484 (queue cap) and the belowMaxRedeemPerBlock modifier at :196

redeem() requires [redacted]Amount >= minRedeemAmount. Both settlement legs then impose the opposite bound: the instant leg runs belowMaxRedeemPerBlock([redacted]Amount) and the queue leg rejects [redacted]Amount > maxRedeemPerBlock outright (the '#4' guard added so an uncompletable request cannot be created). If minRedeemAmount is ever set above maxRedeemPerBlock the two windows do not intersect and no amount is redeemable at all. Neither setter asserts the relationship, and the pattern is inconsistent with the rest of the file, which DOES assert its cross-field invariants (setMinMintAmount/setMinMintFeeAmount and setMinRedeemAmount/setMinRedeemFeeAmount all revert InvalidFee on a bad pair).

Impact. A single mis-ordered governance write -- e.g. tightening maxRedeemPerBlock during an incident while a raised minRedeemAmount is already live -- silently disables 100% of redemptions, both instant and queued, with no revert at configuration time to warn the operator. Existing queued requests above the new cap also become uncompletable, leaving cancelRedeem as the only exit (and that exit is itself pause-gated, see finding S1). The state is admin-recoverable, which caps this at Low.

Recommendation. Assert the invariant in both setters: reject minRedeemAmount > maxRedeemPerBlock in setMinRedeemAmount, and reject maxRedeemPerBlock < minRedeemAmount in setMaxRedeemPerBlock (treating 0 as 'disabled' consistently with maxDeposit/maxMint). Mirror the same check for the mint side if minMintAmount is ever raised above maxMintPerBlock.

PROOF OF CONCEPT — EXECUTED

[redacted]/contracts/test/vari_sweep/SweepTriage.t.sol::SweepTriage::test_PAT0084_minRedeemAboveBlock
;
[redacted]/contracts/test/vari_sweep/SweepTriage.t.sol::SweepTriage::test_PAT0084_setCollateralFeedRe

```
[PASS] test_PAT0084_minRedeemAboveBlockCap_bricksAllRedemption() (gas: 200559)
[PASS] test_PAT0084_setCollateralFeedRejectsOutOfBandFeed() (gas: 279593)
```

PROTO-P0103 · **Low**

[redacted].totalCollateralValue() reverts whole-aggregate when any ONE supported asset's feed is stale or out of band

LOCATION [redacted]/contracts/contracts/[redacted]/[redacted].sol:571-578 (totalCollateralValue) -> :615 (_convertToMctAmount) -> :670 (_getValidatedPrice)

totalCollateralValue() loops every entry of _supportedAssets and prices each through _convertToMctAmount, which for an oracle-priced asset calls the fail-closed _getValidatedPrice. A single asset whose feed has stopped updating (deprecated aggregator), whose answer is non-positive, or whose price falls outside the [MIN_SANE_PRICE, MAX_SANE_PRICE] band reverts the entire aggregate -- there is no per-asset skip and no partial/best-effort variant. The function's own NatSpec names its purpose as 'For monitoring / solvency invariants (backing value >= [redacted] supply)'.

Impact. The protocol's single on-chain solvency read is unavailable precisely in the scenario it exists for: one collateral crashing through the sanity floor, or one feed going dark. Any on-chain consumer, dashboard, keeper or circuit-breaker built on totalCollateralValue() goes blind at the moment of the incident, and the blindness is caused by the smallest, least material asset just as easily as by the largest. Value is not at direct risk (the function is view-only and nothing in the mint/redeem path calls it), which is why this is Low, but it degrades incident response.

Recommendation. Add a non-reverting sibling that prices each asset inside a try/catch and returns (total, unpricedAssets[]) -- or, at minimum, skip assets whose price read reverts and return the priced subtotal alongside a count of unpriced assets, so monitors can distinguish 'insolvent' from 'unmeasurable'.

PROOF OF CONCEPT — EXECUTED

[redacted]/contracts/test/vari_sweep/SweepTriage.t.sol::SweepTriage::test_PAT0103_totalCollateralValu

```
[PASS] test_PAT0103_totalCollateralValue_brickedByOneBadFeed() (gas: 295537)
```

PROTO-09 · **Info**

The synchronous cross-chain composer's redeem leg is permanently unavailable while [redacted]+ has a cooldown (the initializer default)

LOCATION

`[redacted]/contracts/contracts/[redacted]-plus/[redacted]+ Composer.sol (inherits VaultComposerSync._redeem -> VAULT.redeem) against [redacted]/contracts/contracts/[redacted]-plus/[redacted]+.sol:403,415 (withdraw/redeem are ensureCooldownOff)`

`VaultComposerSync._redeem` calls the plain ERC4626 `VAULT.redeem(shares, this, this)`. [redacted]+ gates both `withdraw` and `redeem` on `ensureCooldownOff`, and `initialize` sets `cooldownDuration = 7 days`. With the shipped default every cross-chain [redacted]+ redemption reverts inside `handleCompose` and is refunded back to the source chain, and the local `redeemAndSend` entry point reverts outright. The deposit leg is unaffected. This is inherent to pairing a sync composer with a two-step (cooldown) vault and is not documented in the composer.

Impact. No loss — refunds work — but the advertised omnichain unstake flow is silently non-functional under the default configuration, and each failed attempt costs the user the LayerZero fee for the outbound hop plus the refund hop. Users are pushed onto the hub-side `cooldownShares`/`unstake` path with no in-contract signal.

Recommendation. Either document that cross-chain redemption requires `cooldownDuration == 0`, or override `_redeem` in [redacted]+ Composer to fail fast with a named error (and make `quoteSend` return zero) whenever `VAULT.cooldownDuration() != 0`, so integrators get a clean signal instead of a refunded message.

PROOF OF CONCEPT — EXECUTED

[redacted]/contracts/test/vari_redo/CrossChainAttacks.t.sol::test_CC7_PlusComposerRedeemDeadUnderCool

```
[PASS] test_CC7_PlusComposerRedeemDeadUnderCooldown() (gas: 239085)\nLogs:\n  cross-chain [redacted]+ redemption is unavailable whenever a cooldown is configured\nassertion held: [redacted]PlusComposer.redeemAndSend reverts (OperationNotAllowed from ensureCooldownOff) with cooldownDuration == 7 days
```

PROTO-10 · **Info**

withdrawCollateral rate limit uses a fixed window and can be re-indexed by the role that owns it

LOCATION

`[redacted]/contracts/contracts/[redacted]/[redacted].sol:428-437 (withdrawCollateral epoch cap), :592 (setWithdrawalRateLimit)`

The epoch bucket is `block.timestamp / withdrawalEpochLength`, so (a) two full caps can be consumed seconds apart across a window boundary, and (b) changing `withdrawalEpochLength` re-indexes the bucket and resets the in-window consumption. Both are acknowledged in the code comments. The limiter is set by DEFAULT_ADMIN_ROLE and constrains COLLATERAL_MANAGER_ROLE, so it only bites a compromised manager key, not a compromised admin (who can also simply grant themselves the manager role). Confirmed working as designed for the case it targets.

Impact. The rate limit is a soft control against a compromised operational key only. It does not bound governance. No unprivileged bypass exists.

Recommendation. Consider a sliding window (or a decaying bucket) instead of a fixed epoch, and preserve the consumed amount across an `epochLength` change. Place the limiter's own setters behind the timelock that guards the unbacked-mint cap.

PROOF OF CONCEPT — EXECUTED [redacted]/contracts/test/vari_redo/[redacted].t.sol::test_VR12_Refuted_WithdrawalHardeningBinds

```
[PASS] test_VR12_Refuted_WithdrawalHardeningBinds() (gas: 475664)\nassertions held: an un-allowlisted
destination reverts; a 900e6 withdrawal fills the bucket and a further 200e6 reverts
WithdrawalEpochCapExceeded; after setWithdrawalRateLimit(2 days, ...) the re-indexed bucket reads 0 and
1,800e6 total leaves custody against a 1,000e18 cap
```

PROTO-N09 · **Info**

The uint152 narrowing in RedemptionRequest buys no packing — the struct still occupies two storage slots — so the L-1 guard and the hard 2^152 redemption ceiling it forces are cost without benefit

LOCATION [redacted]/contracts/contracts/interfaces/[redacted]/I[redacted].sol:61-64 (struct RedemptionRequest), consumed at [redacted]/contracts/contracts/[redacted]/[redacted].sol:1480 and :622

RedemptionRequest is declared as `{ uint152 [redacted]Amount; address collateralAsset; }`. uint152 is 19 bytes and address is 20 bytes, totalling 39 — the struct cannot fit in one 32-byte slot, so Solidity lays it out across two slots with 13 bytes of padding wasted in the first and 12 in the second. The narrowing therefore saves nothing. It does, however, cost: it forced the L-1 remediation (`if ([redacted]Amount > type(uint152).max) revert InvalidAmount()` in both _queueRedeem and updateRedemptionRequest) to guard a downcast that only exists to enable a packing that never happened, and it imposes a permanent protocol ceiling on any single queued redemption. The sibling struct UserCooldown `{ uint104 cooldownEnd; uint152 sharesAmount; }` is 13+19 = 32 bytes and does pack correctly, which is presumably where the uint152 was copied from — but the second member there is a uint104, not a 20-byte address.

Impact. No security impact. Slightly higher gas on every queue/update/complete/cancel than the design intended, an unnecessary revert branch on two user-facing paths, and a misleading in-code rationale ('the full uint256 is escrowed to the silo, so silently truncating the stored request amount would strand the difference') that suggests the layout was reasoned about and validated when it was not.

Recommendation. Change [redacted]Amount to uint96 (12 bytes + 20-byte address = exactly one slot) if the packing is wanted — note this is a storage-layout change to a mapping value type and must go through OZ validateUpgrade — or widen it to uint256 across two slots and drop the L-1 guard and the associated ceiling. Either way, verify with `forge inspect` rather than by inspection.

PROOF OF CONCEPT — EXECUTED [redacted]/contracts/test/vari_novel/NovelCore.t.sol::test_N07_RedemptionRequestDoesNotActuallyPack

```
[PASS] test_N07_RedemptionRequestDoesNotActuallyPack() (gas: 166808)
  slot0 ([redacted]Amount): [redacted]0000003635c9adc5dea00000
  slot1 (collateralAsset): [redacted]fa0139dfa1b3d433cc23b72f
(vm.load of keccak256(abi.encode(alice, 7)) and +1 after a queued redemption: the amount alone occupies
slot 0 and the asset needs a second slot)
```

05 Action Points

Findings at Medium severity and above, with the change we recommend. The status column is for your team to track remediation; we re-check these on a follow-up review.

ID	SEVERITY	POTENTIAL SOLUTION	STATUS
PROTO-11	Critical	Override `_revokeRole` in each of the four token contracts to revert when `role` is `FULL_RESTRICTED_ROLE` / `FULL_RESTRICTED_STAKER_ROLE` and the caller is not the role admin -- `renounceRole` routes through `_revokeRole`, so one override covers both entry points instead of patching the five `renounceRole` overrides separately. If you keep the per-function fix, widen the condition at [redacted].sol:1634 (and [redacted]+.sol:765, [redacted].sol:733, [redacted] OFT.sol:198, [redacted]+ OFT.sol:173) from `role == DEFAULT_ADMIN_ROLE` to cover the restriction roles as well. The structural alternative is to stop representing the freeze as an AccessControl role at all and hold it in a dedicated `mapping(address => bool)` written only by `addToBlacklist`/`removeFromBlacklist`; that removes the whole class rather than one instance of it, and is what I would take if you are willing to migrate the state.	
PROTO-01	High	Make the no-feed branch fail closed: revert in `_convertToMctAmount`/`_convertToCollateralAmount` when no feed is set, or require a feed at `addSupportedAsset` time (take feed+maxAge as parameters and validate atomically). If the 1:1 fallback must be kept for a migration window, gate it behind an explicit per-asset `allowUnpricedCollateral` flag that governance must set, emit an event for it, and correct the three NatSpec blocks plus `add-supported-asset.js:41` so operators are not told the system is fail-closed when it is not.	
PROTO-N01	High	Make the invariant hold at the state that defines it, not at one call site. In setVestingPeriod, compute the prospective unvested amount under the NEW period (not the current one) and reject any change that would make it exceed IERC20(asset()).balanceOf(address(this)); equivalently, zero out vestingAmount/lastDistributionTimestamp whenever vestingPeriod is set to 0, so a period of 0 cannot leave a latent schedule behind. Independently, make totalAssets() saturating (`bal > unvested ? bal - unvested : 0`) so no configuration can brick the vault, and add an explicit admin `resetVestingSchedule()` escape hatch.	
PROTO-N02	High	(1) Bound minMintFeeAmount/minRedeemFeeAmount unconditionally, not only when the corresponding minimum amount is non-zero — e.g. require minRedeemAmount > 0 whenever minRedeemFeeAmount > 0, and require minRedeemFeeAmount < minRedeemAmount always. (2) Restore fail-closed behaviour on the redeem path: instead of clamping to 100%, revert BelowMinimumAmount when feeAmountCollateral >= receivedCollateral, mirroring _mintWithCollateral; the stranded-queue concern the clamp addressed is already covered by cancelRedeem().	

		(3) Give redeem() a minCollateralOut parameter so integrators and the composer can set slippage on the fee as well as on the price.
PROTO-02	Medium	Seed the vault atomically at deployment by minting MIN_SHARES of dead shares to address(0)/the protocol (this is the standard fix and removes case (a) and (b) entirely), and either raise `_decimalsOffset()` so donations cannot move the rate materially, or replace the hard MIN_SHARES revert with a floor that is only enforced when totalSupply was already non-zero. Also make `unstake` respect the floor or, better, allow deposits whenever totalSupply()==0 regardless of the asset balance.
PROTO-12	Medium	Port the `[redacted]+ Composer.lzCompose` guard verbatim into `[redacted] Composer.lzCompose`: decode `sendParam.to` before dispatch and revert `BeneficiaryRestricted(beneficiary)` when the beneficiary is blacklisted or lacks valid credentials, on both the deposit and the redeem leg. On the redeem leg place it ahead of the collateral transfer at [redacted] Composer.sol:449-455, since that is where raw USDC leaves the system carrying no protocol-level restriction of its own. Fail into the existing de-whitelist refund path rather than reverting the compose outright, so a bad beneficiary returns the funds to the originator instead of stranding the message.
PROTO-13	Medium	Settle the finished schedule instead of leaving it resident: once `block.timestamp >= lastDistributionTimestamp + vestingPeriod`, zero `vestingAmount` (and stamp `lastDistributionTimestamp`) so `getUnvestedAmount()` and `totalAssets()` cannot re-amortise a distribution that has already been paid out. The minimal patch is to clear both fields inside `setVestingPeriod` ([redacted]+.sol:551-561) after the `StillVesting` check passes and before the new period is written; the structural fix is to do the clearing in `_updateVestingAmount` / a shared `_settleVesting()` called by every entry point, so no caller can ever observe the stale pair. Take the second -- the first leaves the same residue readable by `totalAssets` between a window ending and the next admin call.
PROTO-N03	Medium	Override depositAndSend and redeemAndSend on [redacted]+ Composer to run the same originator (msg.sender) and beneficiary (sendParam.to) restriction checks lzCompose performs — or, better, hoist the checks into overrides of the internal _depositAndSend/_redeemAndSend so every present and future entrypoint inherits them. Enumerate the composer's full inherited ABI and explicitly account for every entry before redeploy.
PROTO-N04	Medium	Remove whenNotPaused from cancelRedeem (keeping nonReentrant and the blacklist check, which is what actually blocks a restricted user via the silo transfer) so it matches cancelCooldown and matches its own NatSpec; at minimum, gate it on the inverse of completionsAllowedWhilePaused so users can always exit whenever the protocol can still settle. Consider the same treatment for the decrease branch of updateRedemptionRequest.

PROTO-N05	Medium	Add the same `renounceOwnership() public view override onlyOwner { revert ... }` to [redacted]RedeemSilo and [redacted]+Silo — they are UUPS-upgradeable, so this ships in the already-planned silo upgrade with no redeploy. Separately, either make lockWiring also freeze upgrades (or move the silos behind a timelocked proxy admin) or soften the 'permanently / one-way' wording so operators do not treat wiringLocked as a hard guarantee.
PROTO-P0095	Medium	Drop whenNotPaused from cancelRedeem (keep nonReentrant and the blacklist check), matching [redacted]+.cancelCooldown and its written rationale. If a pause must be able to freeze the queue, add a separate, narrower flag rather than reusing the global pause, and document the divergence from the sibling contract either way.
