haha: smart-vault-contracts (pilot) analysis report

- Repository: `Permutize/smart-vault-contracts`
- Vulnerabilities: 15
- Warnings: 28

Summary

This analysis reviewed the haha: smart-vault-contracts (pilot) smart contracts using Octane's automated analysis and included team feedback on findings.

The analysis identified a total of 43 issues (15 vulnerabilities, 28 warnings), including 2 high vulnerabilities.

After triage and review, 15 of these issues have been acknowledged.

Vulnerabilities

1. [High] Omitted cWMON in AusdStrategy NAV causes over-minting and LP dilution

Status

Review status: Wont Fix
Remediation status: Acknowledged
Remediation note: Created by pipeline analysis
Acknowledgement reason: Wont Fix
Acknowledgement note: This issue is valid only if standalone `claimRewards()` is considered an intended operational path for cWMON rewards.The contract provides `claimAndProcessRewards()`, which atomically claims rewards and processes cWMON into WMON/MON/vault asset, avoiding the unvalued intermediate cWMON state. If the protocol's operational invariant is that cWMON rewards must always be handled through `claimAndProcessRewards()`, then the issue should be treated as a low-severity operator-flow/documentation concern rather than a direct exploit.

Description

AusdStrategy's NAV excludes cWMON balances that can reside on the strategy after rewards are claimed but before processing, causing cachedTotalAssets to be understated at deposit time and resulting in over-minting of shares and dilution of existing LPs.

AusdStrategy._calculateTotalCollateralUsd() sums protocol collateral, idle shMON, WMON, MON, stablecoin, and pending-unstake MON, but omits any cWMON balance held by the strategy. RewardsAdapter.claimRewards() can deliver cWMON to the strategy, and Swapper.processRewards() only later redeems cWMON into WMON (and optionally swaps to the vault asset). While cWMON remains unprocessed, the strategy's totalAllocatedValue understates true NAV. GatedVaultImpl computes deposit share amounts using cachedTotalAssets, which integrates the strategy's reported value. Therefore, deposits during the window when cWMON is uncounted mint too many shares, diluting existing LPs. Performance-fee accrual timing is also deferred until rewards are processed, reshaping fee recognition.

Severity

Impact Explanation: [High] Incorrect share minting at an understated NAV directly dilutes existing LPs' principal by granting excess shares to new depositors; this is a direct, material loss of principal to victims.

Likelihood Explanation: [Medium] Exploitation requires deposits to be ungated and a timing window where rewards are claimed but not yet processed. This is a plausible operational state but not guaranteed or fully under attacker control.

Exploitation

Exploitation Scenarios:

Scenario 1.

Public depositor front-runs reward processing when deposits are ungated: After the operator claims rewards that include cWMON but before calling processRewards, a public depositor uses ERC-4626 deposit or depositWithPermit2. Because NAV omits the resident cWMON, cachedTotalAssets is too low, so the depositor mints more shares than fair. When the operator later processes rewards and realizes the cWMON value, existing LPs are diluted by the depositor's excess shares.

Preconditions / Assumptions

- (a). Governance has set deposits to be ungated (public deposits allowed).
- (b). Rewards have been claimed that include cWMON and are held by the strategy.
- (c). Operator has not yet called processRewards to redeem cWMON.
- (d). Attacker can detect the timing window and submit a deposit before processing occurs.

Scenario 2.

Operator collusion during gated deposits: While deposits are gated, the WITHDRAWAL_MANAGER claims cWMON to the strategy and then executes depositWithPermit2 for a favored address before processing rewards. Shares are minted against an understated cachedTotalAssets, granting the favored depositor excess shares. When rewards are processed later, the value surfaces and existing LPs are diluted.

Preconditions / Assumptions

- (a). Deposits are gated; depositWithPermit2 can only be executed by WITHDRAWAL_MANAGER.
- (b). WITHDRAWAL_MANAGER claims rewards that include cWMON to the strategy.
- (c). WITHDRAWAL_MANAGER executes a deposit for a favored address before calling processRewards.

Scenario 3.

External cWMON transfer followed by deposit: An attacker transfers cWMON to the strategy (if the token is transferable) to create hidden, uncounted value, then promptly deposits via the public deposit path to mint shares at a depressed NAV. Later, when the operator processes rewards and realizes the cWMON value, the attacker holds a larger share of the now-higher NAV, diluting existing LPs. This path is economically weaker for the attacker due to the donation-like step and dependence on operator processing.

Preconditions / Assumptions

- (a). Deposits are ungated (public).
- (b). cWMON is practically transferable to the strategy address.
- (c). Operator will eventually call processRewards to redeem cWMON.

Proposed fix

RewardsAdapter.sol

File: `src/adapters/rewards/RewardsAdapter.sol`

Source

```
 // SPDX-License-Identifier: AGPL-3.0
 pragma solidity 0.8.28;

 import { MerklAdapter, MerklClaimRequest } from "../merkl/MerklAdapter.sol";
 import { ShMonSwapAdapter } from "../../common/ShMonSwapAdapter.sol";
 import { IUniswapV4 } from "../interfaces/IUniswapV4.sol";
 import { Swapper } from "./Swapper.sol";
+import { ICurvanceWrappedMON } from "../interfaces/ICurvanceWrappedMON.sol";
 import { IWrappedToken } from "../../interface/IWrappedToken.sol";

 ... 193 unchanged lines ...
         MerklAdapter.claim(distributor, address(this), claimRequest);
         emit RewardsClaimed(address(this), distributor, claimRequest.tokens.length);
+        uint256 len = claimRequest.tokens.length;
+        for (uint256 i = 0; i < len; ++i) {
+            address t = claimRequest.tokens[i];
+            try ICurvanceWrappedMON(t).maxRedeem(address(this)) returns (uint256 shares) {
+                if (shares > 0) {
```

```
+                    try ICurvanceWrappedMON(t).redeem(shares, address(this), address(this)) returns (uint256) {
+                }
+            } catch { }
+        }
     }

 ... 50 unchanged lines ...
```

2. [High] Misinterpretation of Curvance cooldown timestamp in CurvanceCollateralAdapter causes extended cooldown and blocked deallocation/withdrawals

Status

Review status: Wrong Severity
Remediation status: Acknowledged
Remediation note: Created by pipeline analysis
Acknowledgement reason: Wrong Severity
Acknowledgement note: We will mark it as a a medium severity.

Description

CurvanceCollateralAdapter treats ICurvanceMarketManager.accountAssets as a last-action timestamp and adds MIN_HOLD_PERIOD again, while the interface documents it as the cooldown end. This doubles the enforced hold, blocking deallocation batches and collateral withdrawals for an extra full period, causing significant temporary availability loss.

ICurvanceMarketManager.accountAssets(account) is documented to return the cooldown end timestamp (time after which actions may be re-enabled). In CurvanceCollateralAdapter._verifyCooldownPassed, the adapter computes cooldownEnd = accountAssets + MIN_HOLD_PERIOD and reverts until block.timestamp >= cooldownEnd. This misinterpretation adds an extra full hold period. BaseStrategy.deallocateFunds calls _verifyCooldowns(), which invokes both collateral adapters' verifyCooldownPassed and therefore blocks any deallocation batch until the extended cooldown elapses. CurvanceCollateralAdapter.withdraw also calls _verifyCooldownPassed internally, directly preventing collateral withdrawals. Additionally, BaseStrategy.maxWithdraw uses canWithdrawNow(), which relies on the same verification and returns 0 longer than intended. The issue causes a deterministic, significant, temporary availability loss of core functionality (unwind/withdraw), with high likelihood whenever cooldowns apply.

Severity

Impact Explanation: [Medium] Significant but temporary availability loss of core strategy functionality (deallocation/withdraw) due to an extra full cooldown period being enforced.

Likelihood Explanation: [High] Deterministically occurs after any action that sets a cooldown, provided accountAssets returns cooldown end as documented; no special constraints required.

Exploitation

Exploitation Scenarios:

Scenario 1.

After a strategy action sets Curvance cooldown (accountAssets set to t_end), the operator executes BaseStrategy.deallocateFunds to withdraw collateral. BaseStrategy._verifyCooldowns() calls CurvanceCollateralAdapter.verifyCooldownPassed, which incorrectly adds MIN_HOLD_PERIOD again and reverts until t_end + MIN_HOLD_PERIOD. Deallocation (including withdrawCollateral) is blocked for an extra full hold period.

Preconditions / Assumptions

- (a). ICurvanceMarketManager.accountAssets returns cooldown end timestamp as documented
- (b). MIN_HOLD_PERIOD > 0
- (c). Strategy uses CurvanceCollateralAdapter for a collateral leg
- (d). Only the vault/operator triggers deallocation batches

Scenario 2.

The market turns volatile shortly after the true Curvance cooldown ends (t > t_end). The operator attempts to de-risk via BaseStrategy.deallocateFunds (withdraw/repay). Due to the double-enforced cooldown, deallocateFunds reverts until t_end + MIN_HOLD_PERIOD. While debt repayment can still be routed via allocateFunds, the inability to withdraw collateral during this window hampers timely risk management and can increase liquidation risk if alternative funding is insufficient.

Preconditions / Assumptions

- (a). ICurvanceMarketManager.accountAssets returns cooldown end timestamp as documented
- (b). MIN_HOLD_PERIOD > 0
- (c). Market volatility during the extra enforced hold window
- (d). Operator may have limited alternative funding; debt repayment via allocateFunds is possible but may be insufficient to fully mitigate risk

Scenario 3.

Frontends or the vault query BaseStrategy.maxWithdraw after the true cooldown has ended (t > t_end). canWithdrawNow() runs cooldown verification and reverts until t >= t_end + MIN_HOLD_PERIOD, causing maxWithdraw to return 0 for an extra full hold period. This misreports available liquidity and degrades UX/ops.

Preconditions / Assumptions

- (a). ICurvanceMarketManager.accountAssets returns cooldown end timestamp as documented
- (b). MIN_HOLD_PERIOD > 0
- (c). Strategy uses CurvanceCollateralAdapter
- (d). Observers or the vault query maxWithdraw during the extra enforced hold window

Proposed fix

CurvanceCollateralAdapter.sol

File: `src/adapters/curvance/CurvanceCollateralAdapter.sol`

Source

```
 ... 118 unchanged lines ...
         // the account has waited long enough since the last relevant position change.
         _verifyCooldownPassed(address(this)); // called via delegatecall, so account context is preserved
-        uint256 shares = COLLATERAL_TOKEN.withdraw(amount, address(this), address(this));
+        uint256 assets = COLLATERAL_TOKEN.withdraw(amount, address(this), address(this));

-        // Curvance returns the number of shares burned rather than the asset amount received.
-        // A zero-share burn indicates that the requested withdrawal could not be executed.
-        if (shares == 0) revert CollateralWithdrawalFailed(amount, 0);
-        emit CollateralWithdrawn(asset, amount, address(this));
+        if (assets < amount) revert CollateralWithdrawalFailed(amount, assets);
+        emit CollateralWithdrawn(asset, assets, address(this));
     }

 ... 98 unchanged lines ...
      */
     function _verifyCooldownPassed(address account) internal view {
-        // Curvance stores the last relevant account action timestamp in accountAssets. A zero
-        // value means the account has no active cooldown to enforce.
-        uint256 cooldownTimestamp = MARKET_MANAGER.accountAssets(account);
-
-        if (cooldownTimestamp > 0) {
-            uint256 minHoldPeriod = MARKET_MANAGER.MIN_HOLD_PERIOD();
-            uint256 cooldownEnd = cooldownTimestamp + minHoldPeriod;
-
-            if (block.timestamp < cooldownEnd) {
-                uint256 remaining = cooldownEnd - block.timestamp;
-                revert CooldownNotPassed(remaining);
-            }
+        uint256 cooldownEnd = MARKET_MANAGER.accountAssets(account);
```

```
+        if (cooldownEnd > 0 && block.timestamp < cooldownEnd) {
+            revert CooldownNotPassed(cooldownEnd - block.timestamp);
         }
     }
 }
```

3. [Medium] Unconditional cooldown checks in BaseStrategy deallocateFunds block idle returns during Curvance cooldown, causing withdrawal DoS

Status

Review status: Wont Fix
Remediation status: Acknowledged
Remediation note: Created by pipeline analysis
Acknowledgement reason: Wont Fix
Acknowledgement note: I already fix in this PR: https://github.com/Permutize/smart-vault-contracts/pull/181/commits

Description

BaseStrategy globally gates deallocation (including balance-only returnIdleToVault) and maxWithdraw on both collateral adapters' cooldown checks. When Curvance's cooldown is active, a pure idle-asset return is blocked and maxWithdraw returns 0, causing withdrawal liveness DoS despite available idle funds.

In BaseStrategy, deallocateFunds unconditionally calls _verifyCooldowns() before executing any commands. The default _verifyCooldowns() invokes verifyCooldownPassed on both collateral adapters. CurvanceCollateralAdapter.verifyCooldownPassed reverts during its hold period, while returnIdleToVault is a balance-only transfer callable only within deallocateFunds. As a result, even a deallocation batch containing only returnIdleToVault is blocked by a Curvance cooldown. Additionally, maxWithdraw calls canWithdrawNow(), which also invokes the same cooldown checks and returns 0 if they fail, masking available idle liquidity. This yields a significant, temporary liveness/DoS of withdrawals sourced from the strategy during Curvance cooldown windows.

Severity

Impact Explanation: [Medium] This causes a significant but temporary availability loss/DoS of a core function (servicing withdrawals from the strategy) during Curvance cooldown windows, but does not permanently freeze funds or break protocol-wide functionality.

Likelihood Explanation: [Medium] Requires Curvance integration and a cooldown window overlapping with the need to return idle funds—an uncommon but plausible operational state; no attacker needed.

Exploitation

Exploitation Scenarios:

Scenario 1.

The strategy integrates Curvance on a collateral leg and currently holds a nontrivial idle balance of the vault asset (e.g., stablecoin). Users have pending withdrawals. The vault submits a deallocation batch containing only returnIdleToVault to forward idle funds. deallocateFunds first calls _verifyCooldowns(), which queries CurvanceCollateralAdapter.verifyCooldownPassed and reverts due to the Curvance hold period, so returnIdleToVault never executes and users' withdrawals are delayed despite available idle funds.

Preconditions / Assumptions

- (a). The strategy uses Curvance on at least one collateral leg
- (b). Curvance hold period is active for the strategy account
- (c). The strategy holds a nontrivial idle balance of the vault asset
- (d). The vault submits a deallocation batch with only returnIdleToVault

Scenario 2.

The vault or coordinator queries maxWithdraw() to plan liquidity sourcing while Curvance cooldown is active and the strategy holds idle funds. maxWithdraw calls canWithdrawNow() → _verifyCooldowns(), which reverts due to the cooldown; maxWithdraw catches and returns 0. This misreports capacity, potentially causing misrouting or delays, even though idle liquidity exists. The underlying availability loss stems from the same overbroad gate that blocks deallocation.

Preconditions / Assumptions

- (a). The strategy uses Curvance on at least one collateral leg
- (b). Curvance hold period is active for the strategy account
- (c). The strategy holds idle vault-asset balance
- (d). The vault/coordinator relies on maxWithdraw() for planning

Proposed fix

BaseStrategy.sol

File: `src/strategies/BaseStrategy.sol`

Source

```
 ... 413 unchanged lines ...
     function maxWithdraw() external view returns (uint256) {
         try this.canWithdrawNow() returns (bool canWithdraw) {
-            if (!canWithdraw) return 0;
+            if (!canWithdraw) return IERC20(this.asset()).balanceOf(address(this));
         } catch {
-            return 0;
+            return IERC20(this.asset()).balanceOf(address(this));
         }

         try this.getCurrentValue() returns (uint256 currentValue) {
             return currentValue;
         } catch {
-            return 0;
+            return IERC20(this.asset()).balanceOf(address(this));
         }
     }
 ... 49 unchanged lines ...
         $.deallocationTracker = 0; // Reset tracker at start

-        _verifyCooldowns();
         _executeCommands(batch.commands, false);

 ... 913 unchanged lines ...
```

4. [Medium] Curvance-style full-repay sentinel in AusdStrategy._terminatePositions when PRIMARY_DEBT_ADAPTER is Euler causes termination revert

Status

Review status: True Positive
Remediation status: Acknowledged
Remediation note: Created by pipeline analysis
Acknowledgement reason: True Positive
Acknowledgement note: Fixed

Description

AusdStrategy._terminatePositions repays primary WMON debt by calling repay(WMON, 0), assuming Curvance's "repay all" sentinel. If the primary debt adapter is Euler, repay(0) reverts, causing terminatePositions to fail despite sufficient WMON to close the debt.

AusdStrategy._terminatePositions hardcodes a full-repay of the primary WMON debt by delegatecalling PRIMARY_DEBT_ADAPTER.repay(WMON, 0). CurvanceDebtAdapter interprets amount == 0 as a full-repay sentinel and

works as intended. EulerDebtAdapter, however, requires an explicit amount; repay(0) approves and repays zero and then reverts with DebtRepayFailed(). Because _terminatePositions bypasses the adapter-agnostic BaseStrategy._repay/resolveRepayInstruction flow, any configuration that sets Euler as the primary debt adapter will make terminatePositions revert whenever wmonDebt > 0, even if wmonBalance >= wmonDebt. There is no on-chain enforcement that the primary debt adapter must be Curvance, and the strategy header claims the primary leg can be Curvance or Euler. Operators can work around this by using the generic repay command (closeAllDebt=true) prior to calling terminatePositions or by crafting explicit unwind steps, but the one-shot termination helper fails under Euler-as-primary.

Severity

Impact Explanation: [Medium] The issue causes a significant but temporary availability loss of an important unwind function (terminatePositions) under a plausible configuration, without directly causing principal loss and with a viable operator workaround.

Likelihood Explanation: [Medium] Configuring Euler as the primary adapter is plausible per the strategy's stated support; operators commonly use terminatePositions for unwinds, so the failure path is realistically reachable.

Exploitation

Exploitation Scenarios:

Scenario 1.

Adapter migration sets PRIMARY_DEBT_ADAPTER to EulerDebtAdapter while the strategy has nonzero primary WMON debt; deallocateFunds including terminatePositions reverts when repay(WMON, 0) is executed, blocking the one-shot unwind.

Preconditions / Assumptions

- (a). PRIMARY_DEBT_ADAPTER is EulerDebtAdapter
- (b). Primary WMON debt is nonzero (wmonDebt > 0)
- (c). Strategy holds enough WMON to fully repay (wmonBalance >= wmonDebt)
- (d). Adapter cooldowns (if any) permit deallocation
- (e). Operator calls deallocateFunds with terminatePositions

Scenario 2.

A new deployment configures Euler as primary; operational runbooks invoke terminatePositions expecting a full unwind; the batch reverts on repay(WMON, 0), preventing automated deallocation until the runbook is updated to use the generic repay helper.

Preconditions / Assumptions

- (a). New strategy instance configured with PRIMARY_DEBT_ADAPTER = EulerDebtAdapter
- (b). Primary WMON debt is nonzero
- (c). Automation/runbook invokes terminatePositions expecting a full unwind
- (d). Adapter cooldowns (if any) permit deallocation

Scenario 3.

During market stress, operators attempt a rapid unwind using terminatePositions with Euler as primary and sufficient WMON on hand; the repay(WMON, 0) call reverts, delaying the unwind until a corrected batch is sent, increasing exposure duration.

Preconditions / Assumptions

- (a). PRIMARY_DEBT_ADAPTER is EulerDebtAdapter
- (b). Primary WMON debt is nonzero and wmonBalance >= wmonDebt
- (c). Market stress motivates rapid unwind via terminatePositions
- (d). Adapter cooldowns (if any) permit deallocation

Proposed fix

AusdStrategy.sol

File: `src/strategies/AusdStrategy.sol`

Source

```
 ... 964 unchanged lines ...
                revert InsufficientPrimaryDebtCoverage(wmonBalance, wmonDebt, wmonDebt - wmonBalance);
            }
-           bytes memory repayData = abi.encodeWithSelector(IDebtAdapter.repay.selector, address(WMON), uint256
-           _delegateCallVoid(address(PRIMARY_DEBT_ADAPTER()), repayData);
+           _repay(ProtocolPosition.PRIMARY, address(WMON), 0, 0, true);
            primaryDebtClosedThisCall = true;
        }
 ... 218 unchanged lines ...
```

Related findings

[Medium] Persistent unlimited approval in CurvanceDebtAdapter repay() causes potential strategy asset drain

Description

CurvanceDebtAdapter sets an unlimited ERC20 allowance to the Curvance debt token when using the full-repay sentinel (amount == 0) and never clears it. Because adapters execute via delegatecall, the approval is granted by the strategy itself and persists across calls and adapter rotations. The sentinel is selected frequently when the visible balance covers the visible debt. If the external Curvance debt token is compromised, maliciously upgraded, or buggy, it can abuse this standing allowance to transferFrom future strategy balances of the debt asset (e.g., WMON), causing principal loss.

When BaseStrategy._repay runs, it queries the adapter's resolveRepayInstruction. In CurvanceDebtAdapter, if the available balance is at least the visible debt, it selects the protocol's full-repay sentinel (repay amount == 0) even when closeAllDebt is false. CurvanceDebtAdapter.repay then force-approves type(uint256).max to the Curvance debt token (DEBT_TOKEN) and calls DEBT_TOKEN.repay(0), but never clears the allowance. Because the adapter is invoked via delegatecall, the approval belongs to the strategy address itself. This approval persists and is not cleared on adapter rotation. Under normal, correct Curvance behavior, repay typically uses msg.sender as payer and repayFor is expected to be delegate-gated, limiting outsider exploitation. However, the persistent unlimited approval expands the trust boundary: if DEBT_TOKEN is ever compromised, maliciously upgraded, or buggy (e.g., owner-pays repayFor without proper delegate gating), it can transferFrom the strategy's future balances of the approved debt asset (such as WMON), causing direct principal loss to vault depositors.

Severity

Impact Explanation: [High] If exploited, the attacker can transfer principal tokens (the debt asset) directly from the strategy, reducing NAV for all vault depositors.

Likelihood Explanation: [Low] Exploitation generally requires the external Curvance debt token to be compromised, maliciously upgraded, or buggy, and also depends on the strategy holding the approved asset at the time of attack.

Exploitation

Exploitation Scenarios:

Scenario 1.

Compromised or maliciously upgraded Curvance debt token calls transferFrom using the strategy's persistent unlimited approval to drain future WMON (or other debt-asset) balances held by the strategy.

Preconditions / Assumptions

- (a). Strategy previously executed repay(0) through CurvanceDebtAdapter, setting an unlimited allowance from the strategy to DEBT_TOKEN for the debt asset.
- (b). Allowance is not cleared and persists on the strategy.
- (c). The strategy later holds a positive balance of the approved debt asset (e.g., WMON).
- (d). Curvance DEBT_TOKEN is compromised or maliciously upgraded to execute arbitrary transferFrom using its spender allowance.

Scenario 2.

After rotating away from Curvance, the old unlimited approval remains; a later compromise/malicious upgrade of the old Curvance debt token exploits the lingering approval to drain newly acquired balances of the debt asset from the strategy.

Preconditions / Assumptions

- (a). Strategy previously executed repay(0) via CurvanceDebtAdapter, creating an unlimited allowance to the old DEBT_TOKEN.
- (b). Governance rotates adapters away from Curvance without clearing prior approvals.
- (c). The strategy later holds a positive balance of the previously approved debt asset.
- (d). The old Curvance DEBT_TOKEN is compromised or maliciously upgraded and can call transferFrom using the lingering allowance.

Scenario 3.

A buggy Curvance repayFor implementation (owner-pays via transferFrom(owner, ...) without strict delegate gating) uses the strategy's persistent unlimited approval to pull the strategy's debt-asset funds.

Preconditions / Assumptions

- (a). Strategy previously executed repay(0) and holds a persistent unlimited allowance to DEBT_TOKEN for the debt asset.
- (b). The strategy currently holds a positive balance of the approved debt asset.
- (c). Curvance DEBT_TOKEN implements repayFor as owner-pays via transferFrom(owner, ...) and lacks robust delegate gating or has a logic bug allowing calls without proper gating.
- (d). An attacker can invoke repayFor targeting the strategy as owner.

Proposed fix

CurvanceDebtAdapter.sol

File: `src/adapters/curvance/CurvanceDebtAdapter.sol`

Source

```
 ... 112 unchanged lines ...
         IERC20(asset).forceApprove(address(DEBT_TOKEN), approvalAmount);
         DEBT_TOKEN.repay(amount);
+        // Clear unlimited approval after full-repay sentinel to avoid leaving a persistent spend-right
+        // on the strategy to the external Curvance debt token.
+        if (amount == 0) {
+            IERC20(asset).forceApprove(address(DEBT_TOKEN), 0);
+        }

         // When amount == 0 the protocol repays the full accrued balance even though the emitted
 ... 193 unchanged lines ...
```

[Medium] Full-repay sentinel misuse in CurvanceDebtAdapter during non-close-all repay causes reserve consumption and deallocation DoS

Description

CurvanceDebtAdapter converts a reserve-clamped repay amount into a repay-all sentinel when amountAvailable >= currentDebt. Because repay(0) sets unlimited approval and Curvance can pull post-accrual debt, this can spend beyond the runtime cap, consuming reserved idle stablecoin or reverting deallocation batches.

In BaseStrategy._repay, when token == asset() and reserveAmount > 0, the strategy clamps the repay budget to preserve a reserve of idle stablecoin. It then queries the adapter for an instruction. CurvanceDebtAdapter.resolveRepayInstruction returns a full-repay sentinel (repay(0)) whenever amountAvailable >= currentDebt, even when closeAllDebt == false. In CurvanceDebtAdapter.repay, amount==0 triggers type(uint256).max approval and lets Curvance pull the full post-accrual debt. If the adapter's currentDebt view is pre-accrual, the real debt at execution (after accrue) can exceed amountAvailable, causing Curvance to pull from the preserved reserve or, if larger than the wallet balance, revert the batch. This violates the base strategy's intent that non-close-all flows honor the runtime cap. The issue is most impactful

for stablecoin debt (the vault asset) during deallocation flows that rely on reserveAmount. EulerDebtAdapter does not use a sentinel and instead caps to explicit amounts, avoiding this risk.

Severity

Impact Explanation: [Medium] Reduces immediate liquidity returned to the vault and can temporarily revert deallocation batches; no direct principal loss.

Likelihood Explanation: [Medium] Plausible during normal operation if Curvance's debt view is pre-accrual and accrual delta exists; does not require attacker or admin actions.

Exploitation

Exploitation Scenarios:

Scenario 1.

Reserve consumption during deallocation: The strategy deallocates Curvance stablecoin debt with closeAllDebt=false and reserveAmount=R. Base clamps repay to excess E to preserve R. The adapter returns sentinel because E >= D_pre (pre-accrual). repay(0) sets unlimited approval and Curvance pulls post-accrual D_post > E, spending Δ = D_post−E from R, reducing liquidity returned to the vault.

Preconditions / Assumptions

- (a). CurvanceDebtAdapter is used for the stablecoin (vault asset) debt leg
- (b). Outstanding stablecoin debt exists on Curvance
- (c). deallocateFunds includes a repay with closeAllDebt=false and reserveAmount=R
- (d). Strategy wallet idle stablecoin = E + R
- (e). Adapter's currentDebt view is pre-accrual (D_pre), while actual post-accrual debt at repay is D_post > D_pre
- (f). E >= D_pre and D_post <= E + R

Scenario 2.

Batch revert due to overshoot: In the same flow, if D_post > E+R, repay(0) attempts to pull more than the wallet's stablecoin balance and the transfer fails, reverting the deallocation batch and temporarily blocking the unwind.

Preconditions / Assumptions

- (a). CurvanceDebtAdapter is used for the stablecoin (vault asset) debt leg
- (b). Outstanding stablecoin debt exists on Curvance
- (c). deallocateFunds includes a repay with closeAllDebt=false and reserveAmount=R
- (d). Strategy wallet idle stablecoin = E + R
- (e). Adapter's currentDebt view is pre-accrual (D_pre), actual post-accrual D_post > D_pre
- (f). D_post > E + R

Scenario 3.

Under-delivery to vault: A partial unwind expects to return exactly R to the vault after a reserved repay. Because repay(0) consumed Δ from R, the subsequent returnIdleToVault can only return R−Δ, under-delivering expected liquidity for that epoch.

Preconditions / Assumptions

- (a). CurvanceDebtAdapter is used for the stablecoin (vault asset) debt leg
- (b). Outstanding stablecoin debt exists on Curvance
- (c). A partial unwind aims to return R to the vault using reserveAmount=R
- (d). Adapter's currentDebt view is pre-accrual (D_pre), actual post-accrual D_post > D_pre
- (e). E >= D_pre and D_post = D_pre + Δ with 0 < Δ <= R

Proposed fix

BaseStrategy.sol

File: `src/strategies/BaseStrategy.sol`

Source

```
 ... 806 unchanged lines ...
             bytes memory instructionResult = address(adapter).functionStaticCall(instructionData);
             (uint256 toRepay, bool useFullRepaySentinel) = abi.decode(instructionResult, (uint256, bool));
-            if (toRepay > 0 || useFullRepaySentinel) {
+            // Prevent reserve bleed: do not use full-repay sentinel when a reserve is requested on the vault a
+            bool skipSentinelRepay = useFullRepaySentinel && reserveAmount > 0 && token == this.asset();
+            if (!skipSentinelRepay && (toRepay > 0 || useFullRepaySentinel)) {
                 bytes memory repayData =
                     abi.encodeWithSelector(IDebtAdapter.repay.selector, token, useFullRepaySentinel ? 0 : toRep
 ... 582 unchanged lines ...
```

5. [Medium] Direct secondary-debt repay bypasses adapter logic in AusdStrategy._terminatePositions causes termination revert/DoS with Curvance

Status

Review status: True Positive
Remediation status: Acknowledged
Remediation note: Created by pipeline analysis
Acknowledgement reason: True Positive
Acknowledgement note: Fixed

Description

AusdStrategy._terminatePositions repays the secondary stablecoin debt by directly calling repay(amount) and bypasses BaseStrategy._repay and CurvanceDebtAdapter.resolveRepayInstruction. With Curvance as the secondary debt adapter, this can revert due to pending-interest dust and MIN_LOAN_SIZE rules, blocking termination/unwind.

In AusdStrategy._terminatePositions, the strategy computes toRepay = min(stablecoinBalance, visibleDebt) for the secondary leg and directly delegatecalls SECONDARY_DEBT_ADAPTER.repay(asset, toRepay). This bypasses BaseStrategy._repay, which would have invoked resolveRepayInstruction on the adapter. CurvanceDebtAdapter.getDebtBalance returns a pre-interest debt value, while the Curvance repay path applies/accrues interest before validation. When attempting a "full" repay with an explicit amount matching pre-interest debt, a small residual (the accrued interest) remains. Curvance enforces a minimum USD loan size for residual debt during repay; a nonzero residual below MIN_LOAN_SIZE causes revert. For partial repays, if the residual would fall below MIN_LOAN_SIZE, Curvance also reverts. BaseStrategy._repay avoids these failures by using resolveRepayInstruction to select the full-repay sentinel (amount == 0) when balance covers debt, or to skip/size partial repayments safely. Because _terminatePositions bypasses this only for the secondary leg, termination can revert and block deallocation when Curvance is configured as secondary.

Severity

Impact Explanation: [Medium] Reverts during terminatePositions create a significant but temporary availability loss/DoS of the unwind/deallocation path. No principal loss occurs and operators can work around it by using the repay command that leverages resolveRepayInstruction.

Likelihood Explanation: [Medium] The issue manifests when the strategy is configured with Curvance as the secondary debt adapter and terminatePositions is used. Within that configuration, pending interest and MIN_LOAN_SIZE checks make the revert likely, but the configuration precondition reduces overall likelihood to medium.

Exploitation

Exploitation Scenarios:

Scenario 1.

Termination attempts a full repay of Curvance secondary debt: after withdrawing primary collateral, the strategy holds enough stablecoin to cover the visible debt and calls repay with that exact amount; Curvance applies interest first, leaving a small residual below MIN_LOAN_SIZE and the repay reverts, causing terminatePositions to fail and blocking unwind.

Preconditions / Assumptions

- (a). Secondary debt adapter is CurvanceDebtAdapter (stablecoin debt).
- (b). Secondary debt > 0.
- (c). Pending interest on Curvance debt is nonzero (normal behavior).
- (d). terminatePositions is invoked after primary WMON debt is closed and primary stablecoin collateral is withdrawn, leaving sufficient stablecoin to cover the visible secondary debt.
- (e). terminatePositions uses direct repay(amount) for the secondary leg (no resolveRepayInstruction).

Scenario 2.

Termination attempts a partial repay of Curvance secondary debt: the strategy balance is insufficient to fully close the debt; toRepay = min(balance, debt) leaves a residual whose USD value is below MIN_LOAN_SIZE; Curvance rejects the partial repay and terminatePositions reverts.

Preconditions / Assumptions

- (a). Secondary debt adapter is CurvanceDebtAdapter.
- (b). Secondary debt > 0.
- (c). Strategy stablecoin balance is positive but less than the visible debt.
- (d). The residual (debt - toRepay) has USD value below Curvance MIN_LOAN_SIZE.
- (e). terminatePositions uses direct repay(amount) for the secondary leg (no resolveRepayInstruction).

Scenario 3.

Two-step termination flow aggravates the issue: the first call closes primary WMON debt and returns early; on the subsequent call, the strategy reads a pre-interest secondary debt value and repays it exactly without the sentinel, increasing the chance of a residual dust-induced revert that blocks termination.

Preconditions / Assumptions

- (a). Secondary debt adapter is CurvanceDebtAdapter.
- (b). Operators follow the intended two-step termination sequence (first call closes primary debt; second call withdraws primary collateral and repays secondary debt).
- (c). Pending interest exists at the time of the second call.
- (d). terminatePositions uses direct repay(amount) for the secondary leg (no resolveRepayInstruction).

Proposed fix

AusdStrategy.sol

File: `src/strategies/AusdStrategy.sol`

Source

```
 ... 21 unchanged lines ...
 import { ICollateralAdapter } from "../adapters/interfaces/ICollateralAdapter.sol";
 import { IDebtAdapter } from "../adapters/interfaces/IDebtAdapter.sol";
+import { IBaseStrategy } from "../interface/IBaseStrategy.sol";
 import { AusdStrategyStorageLib } from "../libraries/StrategyStorageLib.sol";
 import { StrategyRolesLib } from "../libraries/StrategyRolesLib.sol";
 ... 962 unchanged lines ...
         uint256 stablecoinBalance = STABLECOIN.balanceOf(address(this));
         if (stablecoinDebt > 0 && stablecoinBalance > 0) {
-            uint256 toRepay = stablecoinBalance < stablecoinDebt ? stablecoinBalance : stablecoinDebt;
-            bytes memory repayData = abi.encodeWithSelector(IDebtAdapter.repay.selector, address(STABLECOIN), t
-            _delegateCallVoid(address(SECONDARY_DEBT_ADAPTER()), repayData);
+            _repay(IBaseStrategy.ProtocolPosition.SECONDARY, address(STABLECOIN), 0, 0, false);
         }
     }
 ... 194 unchanged lines ...
```

6. [Medium] Direct WMON.withdraw to proxy in Swapper.processRewards (MON output) causes DoS of MON reward processing

Status

Review status: True Positive
Remediation status: Acknowledged
Remediation note: Created by pipeline analysis
Acknowledgement reason: True Positive
Acknowledgement note: Fixed

Description

The MON-output branch in Swapper.processRewards unwrapped WMON by calling withdraw directly to the strategy proxy. If WMON uses WETH9-style transfer(2300) semantics, the proxy cannot receive the native transfer, causing consistent reverts whenever toUnwrap > 0 and making the MON-output reward processing path unusable.

In Swapper.processRewards, when params.outputAsset == MON, the code computes toUnwrap = max(currentWmon − targetRemainingWmon, 0) and, if positive, calls IWrappedToken(wmon).withdraw(toUnwrap) directly to the strategy proxy. For proxy-based strategies (UUPS/1967), a WETH9-like WMON typically uses transfer with a 2300-gas stipend, which is insufficient for the proxy's fallback/receive path, causing the withdraw to revert. The codebase otherwise uses a dedicated WMonUnwrapper (in VAULT_ASSET and other flows) specifically to avoid this 2300-gas hazard, but the MON-output path bypasses it. As a result, processing rewards to MON is consistently DoS'd under common deployment conditions whenever toUnwrap > 0. Reverts are atomic, so no funds are locked or lost; however, the advertised MON-output mode cannot be used until operators switch modes or the code is fixed to route the MON path via the unwrapper.

Severity

Impact Explanation: [Medium] Breaks important non-core functionality (the MON-output reward processing mode) and can cause operational DoS of that path, though no principal loss or long-term fund freeze occurs and workarounds exist.

Likelihood Explanation: [Medium] Depends on deployment conditions (proxy-based strategy) and WMON withdraw semantics being WETH9-like, both of which are plausible and anticipated by the codebase; no attacker is required for failure.

Exploitation

Exploitation Scenarios:

Scenario 1.

Griefing DoS: An attacker sends dust WMON to the strategy so currentWmon > targetRemainingWmon. The rewards manager calls processRewards with outputAsset = MON, making toUnwrap > 0. WMON.withdraw attempts a native transfer to the proxy and reverts due to 2300-gas stipend, DoS'ing the MON-output reward processing.

Preconditions / Assumptions

- (a). Strategy deployed behind a UUPS/1967 proxy
- (b). WMON withdraw uses transfer with 2300-gas stipend (WETH9-like behavior)
- (c). Attacker can send dust WMON to the strategy address
- (d). Rewards manager calls processRewards with outputAsset = MON and targetRemainingWmon < currentWmon

Scenario 2.

Operator-only failure: Without any attacker, the strategy accumulates WMON (e.g., from rewards or cWMON redemption). The operator sets targetRemainingWmon below currentWmon and calls processRewards with outputAsset = MON. The direct withdraw to the proxy reverts for the same 2300-gas reason, failing the MON-output processing (and any bundled claim+process).

Preconditions / Assumptions

- (a). Strategy deployed behind a UUPS/1967 proxy
- (b). WMON withdraw uses transfer with 2300-gas stipend (WETH9-like behavior)
- (c). Strategy holds WMON (e.g., from rewards or cWMON redemption)

- (d). Rewards manager calls processRewards with outputAsset = MON and targetRemainingWmon < currentWmon

Scenario 3.

Automation failure: A scheduled job periodically calls processRewards with outputAsset = MON and a fixed targetRemainingWmon. With a proxy strategy and WETH9-like WMON, toUnwrap > 0 frequently, causing withdraw to revert each run and blocking the MON-output task until reconfigured.

Preconditions / Assumptions

- (a). Strategy deployed behind a UUPS/1967 proxy
- (b). WMON withdraw uses transfer with 2300-gas stipend (WETH9-like behavior)
- (c). Scheduled automation calls processRewards with outputAsset = MON
- (d). Strategy often has currentWmon > targetRemainingWmon at run time

Proposed fix

Swapper.sol

File: `src/adapters/rewards/Swapper.sol`

Source

```
 ... 188 unchanged lines ...
         // Convert WMON into native MON while preserving the configured reserve.
         if (params.outputAsset == RewardAsset.MON) {
-            uint256 monBefore = address(this).balance;
             uint256 currentWmon = IWrappedToken(wmon).balanceOf(address(this));
             uint256 toUnwrap = currentWmon > params.targetRemainingWmon ? currentWmon - params.targetRemainingW
-            if (toUnwrap > 0) {
-                IWrappedToken(wmon).withdraw(toUnwrap);
+            if (toUnwrap == 0) {
+                return 0;
             }
-            return address(this).balance - monBefore;
+            // Use the dedicated unwrapper to avoid 2300-gas transfer issues to proxies.
+            uint256 monReceived = _unwrapWMon(wmon, wmonUnwrapper, toUnwrap);
+            return monReceived;
         }

 ... 121 unchanged lines ...
```

7. [Medium] Pre-unwrapping full WMON input with partial settlement in ShMonSwapAdapter._swapV4 (WMON->shMON) causes idle MON and reduced yield

Status

Review status: True Positive
Remediation status: Acknowledged
Remediation note: Created by pipeline analysis
Acknowledgement reason: True Positive
Acknowledgement note: Fixed

Description

During WMON->shMON Uniswap v4 swaps, the adapter unwraps the entire input to MON before swapping but settles only the amount actually consumed. If the swap partially fills, the unused MON remains idle and is not rewrapped, leaving unproductive MON exposure and reducing yield until explicitly swept.

In ShMonSwapAdapter._swapV4, when tokenIn == WMON, the function unwraps the full requested amountIn to native MON before calling PoolManager.unlock. Inside unlockCallback, the Uniswap v4 swap returns a BalanceDelta that may reflect a partial fill; the adapter then settles only the consumed MON (settleAmount = -rawInput) and pulls the output. If the swap does not consume the entire amountIn (e.g., due to a price limit or insufficient liquidity), the unconsumed MON

remains on the strategy as native balance. The adapter's post-swap wrap-back runs only when tokenOut == WMON, so no automatic sweep occurs in the WMON->shMON direction. While NAV accounting includes idle MON, batches that expected full conversion can end up with idle MON and less shMON collateral than intended, reducing productive exposure until a subsequent sweep occurs.

Severity

Impact Explanation: [Medium] Idle MON left after partial fills results in a direct, material loss of yield because a portion of volatile exposure remains unproductive until it is explicitly swept.

Likelihood Explanation: [Medium] Partial fills are uncommon but plausible under realistic operator parameterization (permissive minAmountOut or set price limits) and market liquidity conditions; no attacker action or operator mistake is required.

Exploitation

Exploitation Scenarios:

Scenario 1.

Partial fill of a WMON->shMON v4 swap leaves unconsumed MON idle; the batch does not include a cleanup step, resulting in reduced shMON collateral and lower yield until a later sweep.

Preconditions / Assumptions

- (a). Strategy holds WMON intended to be converted to shMON via swapViaUniswapV4
- (b). Operator sets a permissive minAmountOut or a non-zero sqrtPriceLimitX96
- (c). Uniswap v4 pool liquidity or the price limit results in a partial fill (settleAmount < amountIn)
- (d). The batch does not include a follow-up sweep (e.g., stakeMonToShMon(0, minOut) or wrapMon(0))

Scenario 2.

Partial fill leaves idle MON; off-chain monitoring that relies on getPositionState (which excludes idle balances) under-reports secondary collateral exposure while NAV remains correct.

Preconditions / Assumptions

- (a). Strategy performs a WMON->shMON v4 swap that partially fills, leaving idle MON
- (b). Off-chain monitoring primarily uses getPositionState (protocol-centric snapshot) rather than full NAV, leading to a temporary exposure discrepancy

Proposed fix

ShMonSwapAdapter.sol

File: `src/common/ShMonSwapAdapter.sol`

Source

```
 ... 217 unchanged lines ...
         uint256 balanceBefore = IERC20(params.tokenOut).balanceOf(address(this));
         uint256 nativeBalanceBefore;
+        uint256 nativeBalanceBeforeInput;

         if (inputIsWMon) {
+            nativeBalanceBeforeInput = address(this).balance;
             _unwrapWMon(wmon, wmonUnwrapper, amountIn);
         }
 ... 26 unchanged lines ...
             _wrapMon(wmon, nativeReceived);
         }
+        if (inputIsWMon) {
+            // Wrap any unconsumed MON (from partial fills) back into WMON to avoid leaving idle native balance
+            uint256 leftover = address(this).balance - nativeBalanceBeforeInput;
```

```
+            _wrapMon(wmon, leftover);
+        }

         uint256 amountOut = IERC20(params.tokenOut).balanceOf(address(this)) - balanceBefore;
 ... 66 unchanged lines ...
```

Related findings

[Low] Missing full-fill enforcement in Uniswap v4 reward swap (Swapper.processRewards/unlockCallback) causes leftover MON exposure after partial fills

Description

When processing rewards to the vault asset, the swapper unwraps all excess WMON to MON and attempts an exact-input Uniswap v4 swap with a price limit. If the price limit is hit, Uniswap can partially fill the swap. The callback only checks output >= minAmountOut and settles the consumed input, never enforcing full input consumption. Any unconsumed MON remains on the strategy, creating unintended MON exposure.

In the VAULT_ASSET branch of Swapper.processRewards, all excess WMON above a configured reserve is unwrapped to native MON and then swapped via a Uniswap v4 PoolManager using an exact-input swap with a stored sqrtPriceLimitX96. In unlockCallback, the code reads the actual consumed input from the BalanceDelta (settleAmount = -amount0), settles only that amount, and validates the output side (takeAmount >= minAmountOut). It never verifies that settleAmount equals the intended monAmount. Under canonical Uniswap v4 semantics, if the price limit is reached mid-swap, only part of the input is consumed, and the function still succeeds if minAmountOut is met. The remainder of the pre-unwrapped MON stays on the strategy, leaving unintended native MON exposure rather than fully converting the rewards into the vault asset. This does not lose principal (NAV accounts for MON), but it deviates from intended policy, can accumulate across harvests, and can be made more likely via MEV that pushes price to the configured bound if operator minAmountOut is lenient.

Severity

Impact Explanation: [Low] No principal loss or freeze; NAV remains correct as MON is accounted. The impact is operational drift and unintended MON exposure (timing risk) if partial fills are accepted, not a direct, material loss.

Likelihood Explanation: [Low] Requires trusted-role configurations (tight admin price limit and lenient operator minAmountOut) and, for attacker-driven cases, MEV timing and pool conditions. These dependencies on operator/admin choices reduce overall likelihood.

Exploitation

Exploitation Scenarios:

Scenario 1.

An MEV actor front-runs a rewards swap to push the MON/stable price to the admin-configured sqrtPriceLimitX96, causing the swap to partially fill. Because the operator set a lenient minAmountOut, the swap succeeds, settling only the consumed portion and leaving residual MON on the strategy instead of fully converting to the vault asset.

Preconditions / Assumptions

- (a). Strategy holds excess WMON (above targetRemainingWmon)
- (b). Admin-set sqrtPriceLimitX96 can bind near prevailing price
- (c). Operator-supplied minAmountOut is lenient enough for partial-fill output to pass
- (d). Uniswap v4 MON/stable pool liquidity allows price movement to the bound
- (e). Attacker can front-run/back-run (MEV in scope)

Scenario 2.

Market price naturally drifts to the configured sqrtPriceLimitX96 during a harvest. The swap partially fills and still meets a lenient minAmountOut, leaving a leftover MON balance on the strategy contrary to the intent to fully convert excess WMON.

Preconditions / Assumptions

- (a). Strategy holds excess WMON (above targetRemainingWmon)
- (b). Admin-set sqrtPriceLimitX96 is relatively tight
- (c). Operator-supplied minAmountOut is lenient enough for partial-fill output to pass
- (d). Market price naturally touches the configured bound during the swap

Scenario 3.

Across multiple predictable harvests, an MEV actor repeatedly nudges price to the configured bound right before each swap, inducing small partial fills each time. Residual MON accumulates over time, creating exposure timing drift until operators explicitly normalize the balance.

Preconditions / Assumptions

- (a). Predictable harvest cadence
- (b). Admin-set sqrtPriceLimitX96 remains tight across harvests
- (c). Operator minAmountOut remains lenient across harvests
- (d). Attacker can repeatedly induce price near the bound with reasonable cost (MEV in scope)

Proposed fix

Swapper.sol

File: `src/adapters/rewards/Swapper.sol`

Source

```
 ... 127 unchanged lines ...
     error InvalidRewardSqrtPriceLimit(uint160 sqrtPriceLimitX96);
     /// @notice Revert when unwrap is requested before the helper is configured.
+    error IncompleteRewardSwap(uint256 settledAmount, uint256 intendedAmount);
     error WMonUnwrapperNotConfigured();

 ... 135 unchanged lines ...
         // forge-lint: disable-next-line(unsafe-typecast)
         uint256 settleAmount = uint256(uint128(-raw0));
+        if (settleAmount != callbackData.monAmount) revert IncompleteRewardSwap(settleAmount, callbackData.monA
         // Settle the MON that the pool manager is owed for the swap.
         IPoolManager(TRUSTED_POOL_MANAGER).settle{ value: settleAmount }();
 ... 50 unchanged lines ...
```

8. [Low] Lack of native-balance verification and proxy-safe delivery in immediate shMON→MON redemption (SwapAdapter.unstakeShMonToMon) causes DoS of convert/termination when shMON uses 2300-gas sends

Status

Review status: False Positive
Remediation status: Acknowledged
Remediation note: Created by pipeline analysis
Acknowledgement reason: False Positive
Acknowledgement note: After re-checking this finding against the current shMON implementation, we believe the reported 2300-gas liveness failure is not applicable.The report assumes shMON redemption transfers native MON using transfer/send-style stipend behavior, which could fail when sending to a proxy. However, shMON currently uses:call(gas(), to, amount, ...)inside safeTransferETH, i.e. a full-gas native transfer call, not a 2300-gas stipend transfer.Because of that, the specific failure mode “redeem to strategy proxy reverts due to stipend-limited native send” is stale/invalid for this deployed code path.ref: 1. https://github.com/FastLane-Labs/fastlane-contracts/blob/2cf37b43d855ca2ff8d561f7db97603e5db28399/src/shmonad/FLERC4626.sol#L2152. https://github.com/Vectorized/solady/blob/5dd6f93498b8ebdc9d72194cfa680a90b738e1ad/src/utils/SafeTransferLib.sol#L89C1-L98C6

Description

Immediate shMON→MON redemption calls ERC-4626 redeem to the strategy proxy and trusts the returned value without verifying native balance delta, unlike the WMON unwrap path that uses a helper. If shMON delivers MON using 2300-gas transfer/send, native delivery to the UUPS proxy can revert, blocking immediate redemption and termination flows. Workarounds (DEX swaps, async unstake) exist.

SwapAdapter.unstakeShMonToMon redeems shMON shares directly to address(this) via IERC4626.redeem and treats the return value as the received MON, without measuring the native balance change. By contrast, unwrapWMon routes through WMonUnwrapper to avoid 2300-gas send issues to a proxy and to measure the actual native delta. Because the strategy is a UUPS (ERC1967) proxy, if shMON's redeem implementation uses 2300-gas-limited transfer/send to deliver native MON to the receiver, the send to the proxy can revert. This blocks immediate redemption used by AusdStrategy._unstakeShMonToMon and _swapShMonToWMon (including inside _terminatePositions). Although alternative paths exist (swapViaUniswapV3/v4 and asynchronous unstake/complete flows), the provided immediate-redeem and termination helpers can be unavailable under these conditions.

Severity

Impact Explanation: [Medium] Significant but temporary availability loss/DoS of important strategy operations (immediate conversion and built-in termination flow). No direct principal loss and workarounds exist (DEX swaps, async unstake).

Likelihood Explanation: [Low] Depends on shMON using 2300-gas transfer/send for native delivery; modern implementations often use call to avoid this. The condition is plausible but uncertain, so overall likelihood is low.

Exploitation

Exploitation Scenarios:

Scenario 1.

Immediate redemption DoS: Operator runs a batch that performs immediate shMON→MON via SwapAdapter.unstakeShMonToMon. shMON's redeem pays native MON using 2300-gas transfer/send to the receiver (the UUPS proxy). The native send fails, redeem reverts, and the batch aborts, blocking the immediate conversion path.

Preconditions / Assumptions

- (a). Strategy is deployed behind an OZ UUPS (ERC1967) proxy
- (b). Immediate shMON→MON redemption path is invoked in a batch
- (c). shMON's redeem implementation delivers native MON using 2300-gas transfer/send to the receiver

Scenario 2.

Termination blocked: Governance calls terminatePositions, which uses _swapShMonToWMon to convert shMON→MON→WMON via immediate redemption. If redeem reverts due to 2300-gas send to the proxy, termination halts at the conversion step. Operators must re-route using Uniswap swaps or async flows to proceed.

Preconditions / Assumptions

- (a). Strategy is deployed behind an OZ UUPS (ERC1967) proxy
- (b). Governance calls terminatePositions which relies on immediate redemption via _swapShMonToWMon
- (c). shMON's redeem implementation delivers native MON using 2300-gas transfer/send to the receiver

Proposed fix

SwapAdapter.sol

File: `src/common/SwapAdapter.sol`

Source

```
 ... 90 unchanged lines ...

         // Redeem directly into native MON and validate the received amount against the caller's
-        // minimum-out guard.
-        uint256 monReceived = shMon.redeem(sharesToRedeem, address(this), address(this));
```

```
+        // minimum-out guard. Measure actual native delta instead of trusting ERC-4626 return value.
+        // NOTE: To also avoid 2300-gas send issues to proxies, redeem to a native-forwarder helper
+        // (like WMonUnwrapper) and forward with full gas in a follow-up change.
+        uint256 monBefore = address(this).balance;
+        shMon.redeem(sharesToRedeem, address(this), address(this));
+        uint256 monReceived = address(this).balance - monBefore;
         if (monReceived == 0) revert ZeroMonOutput();
         if (monReceived < minMonOut) revert InsufficientMonOutput(monReceived, minMonOut);
 ... 34 unchanged lines ...
```

9. [Low] Non-idempotent role grant helper in BaseStrategy.setStrategyAdmin causes admin rotation revert when newAdmin already holds a role

Status

Review status: True Positive
Remediation status: Acknowledged
Remediation note: Created by pipeline analysis
Acknowledgement reason: True Positive
Acknowledgement note: Fixed

Description

BaseStrategy.setStrategyAdmin uses a helper that reverts if OpenZeppelin v5 _grantRole returns false. In OZ v5, _grantRole returns false when the role is already assigned, so rotating to a newAdmin who already has STRATEGY_ADMIN or OPERATOR_ROLE reverts and blocks the one-call handoff.

OpenZeppelin v5's internal _grantRole returns false when no state change occurs (i.e., the account already has the role). BaseStrategy._grantRequiredRole treats a false return as failure and reverts. In setStrategyAdmin(newAdmin), the contract sets $.strategyAdmin, then calls _grantRequiredRole for STRATEGY_ADMIN and OPERATOR_ROLE. If newAdmin already holds either role, the call reverts, undoing the storage write and blocking rotation. This is an operational/idempotency issue that requires governance to first revoke roles from newAdmin and then call setStrategyAdmin, or to manage role grants/revokes directly outside the helper. Additionally, setStrategyAdmin attempts revokeRole(OPERATOR_ROLE, previousAdmin), which requires the caller to also hold OPERATOR_ADMIN; otherwise the function reverts even in a clean case. No external attacker exploitation is enabled; impact is limited to operational friction and temporarily delayed rotation.

Severity

Impact Explanation: [Low] The issue causes an unexpected revert and delays admin rotation but does not directly impact user funds or core protocol functionality. It is a correctness/idempotency issue with operational friction.

Likelihood Explanation: [Medium] Pre-granting roles to future admins or operators is a plausible and not-rare operational practice, making the revert state reasonably likely during real operations.

Exploitation

Exploitation Scenarios:

Scenario 1.

Rotation to an address that already has OPERATOR_ROLE: setStrategyAdmin writes the new admin, grants STRATEGY_ADMIN successfully, then attempts to grant OPERATOR_ROLE; OpenZeppen _grantRole returns false (role already present), causing a revert and blocking rotation.

Preconditions / Assumptions

- (a). Caller holds STRATEGY_ADMIN_ADMIN and OPERATOR_ADMIN
- (b). newAdmin already has OPERATOR_ROLE
- (c). previousAdmin != newAdmin
- (d). OpenZeppelin v5 semantics for _grantRole (returns false if role already assigned)

Scenario 2.

Rotation to an address that already has STRATEGY_ADMIN: setStrategyAdmin writes the new admin, then attempts to grant STRATEGY_ADMIN; OpenZeppelin _grantRole returns false (role already present), causing a revert and blocking rotation.

Preconditions / Assumptions

- (a). Caller holds STRATEGY_ADMIN_ADMIN
- (b). newAdmin already has STRATEGY_ADMIN
- (c). previousAdmin != newAdmin
- (d). OpenZeppelin v5 semantics for _grantRole (returns false if role already assigned)

Scenario 3.

Caller lacks OPERATOR_ADMIN: setStrategyAdmin grants both roles to a clean newAdmin, then attempts revokeRole(OPERATOR_ROLE, previousAdmin) but reverts because the caller does not have OPERATOR_ADMIN, blocking rotation.

Preconditions / Assumptions

- (a). Caller holds STRATEGY_ADMIN_ADMIN but not OPERATOR_ADMIN
- (b). newAdmin initially has neither STRATEGY_ADMIN nor OPERATOR_ROLE
- (c). previousAdmin != newAdmin
- (d). admin(OPERATOR_ROLE) is OPERATOR_ADMIN per StrategyRolesLib

Proposed fix

BaseStrategy.sol

File: `src/strategies/BaseStrategy.sol`

Source

```
 ... 1012 unchanged lines ...
     /// @dev Role grants should never silently fail because missing permissions strand a deployed strategy.
     function _grantRequiredRole(bytes32 role, address account) internal {
-        if (!_grantRole(role, account)) revert RoleGrantFailed(role, account);
+        // Make idempotent: only grant when the account does not already hold the role.
+        // In OZ v5, _grantRole returns false when the role is already present; that is not an error condition.
+        if (!hasRole(role, account)) {
+            _grantRole(role, account);
+        }
     }

 ... 376 unchanged lines ...
```

10. [Low] Live PPS-based basis guard in AusdStrategy causes temporary DoS of allocation and deallocation

Status

Review status: True Positive
Remediation status: Acknowledged
Remediation note: Created by pipeline analysis
Acknowledgement reason: True Positive
Acknowledgement note: Fixed

Description

AusdStrategy gates allocateFunds and deallocateFunds on a basis computed from shMON convertToAssets versus the shMON oracle. Rapid PPS changes that outpace the oracle can push the basis above the guard and revert both paths, temporarily blocking unwinds needed for withdrawals until the oracle updates or the guard is relaxed.

AusdStrategy.getBasisBps() derives a basis from SHMON.convertToAssets(1 share) (valued via WMON oracle) against the shMON oracle price. allocateFunds always enforces _checkBasisGuard(); deallocateFunds enforces the same guard unless getBasisBps() reverts (e.g., stale oracle), in which case it bypasses the guard. If getBasisBps() returns a high value (not a revert), both allocateFunds and deallocateFunds revert with BasisOutOfRange. When shMON's PPS (convertToAssets) can move quickly (e.g., unsolicited MON counted in SHMON's assets or internal accounting events) and the oracle remains fresh but lags this change, getBasisBps() can exceed basisGuardBps (default 200 bps). This creates a bounded, heartbeat-limited liveness/DoS window where both allocation and deallocation are blocked, delaying unwinds for withdrawals. Operators can mitigate by adjusting basisGuardBps via setBasisGuardBps, but that weakens the protection. Deallocation only auto-bypasses the guard when the oracle reverts (stale), not when it returns a high value.

Severity

Impact Explanation: [Low] Temporary, heartbeat-bounded unavailability of allocation/deallocation in an asynchronous system, with an explicit admin override (setBasisGuardBps) and no principal loss.

Likelihood Explanation: [Low] Donation-driven scenarios are griefing with timing and capital requirements and rely on SHMON behavior; non-adversarial PPS jumps large enough to exceed the guard while the oracle lags are uncommon. Windows are short and bounded by oracle heartbeats.

Exploitation

Exploitation Scenarios:

Scenario 1.

Donation-driven DoS during withdrawals: An attacker donates enough MON to the shMON contract to raise convertToAssets above the basis guard threshold while the shMON oracle is still fresh but lagging. The vault then calls deallocateFunds; getBasisBps() resolves to a high value and deallocateFunds reverts, temporarily blocking the unwind until the oracle updates or governance relaxes the guard.

Preconditions / Assumptions

- (a). SHMON accepts unsolicited MON or internal accounting promptly increases assets used by convertToAssets
- (b). SHMON_ORACLE and WMON_ORACLE are within heartbeat; SHMON_ORACLE lags the PPS jump
- (c). basisGuardBps configured to a relatively tight threshold (e.g., 200 bps)
- (d). Vault attempts deallocation during the lag window
- (e). Attacker can donate and time the action near the vault operation

Scenario 2.

Early low-TVL DoS affecting both allocation and deallocation: With small shMON equity, a modest donation raises PPS by more than the default 2% guard, causing both allocateFunds (via _checkBasisGuard) and deallocateFunds to revert during the oracle-lag window, preventing growth or unwinds until the oracle updates or the guard is adjusted.

Preconditions / Assumptions

- (a). Low shMON TVL/equity so a small donation can exceed the guard threshold
- (b). SHMON_ORACLE remains fresh but lags PPS during the window
- (c). basisGuardBps set (e.g., 200 bps)
- (d). Vault attempts allocation or deallocation during the lag window

Scenario 3.

Non-adversarial PPS jump: Normal shMON operations (e.g., reward/epoch transitions) cause convertToAssets to increase abruptly. The shMON oracle remains fresh but lags, so getBasisBps exceeds the guard and deallocateFunds (and allocateFunds) revert with BasisOutOfRange, temporarily blocking unwinds until the oracle catches up or the guard is relaxed.

Preconditions / Assumptions

- (a). Internal shMON events can cause abrupt PPS increases > basisGuardBps
- (b). SHMON_ORACLE remains within heartbeat but lags PPS
- (c). Vault attempts allocation or deallocation during the lag window

Proposed fix

AusdStrategy.sol

File: `src/strategies/AusdStrategy.sol`

Source

```
 ... 407 unchanged lines ...
         if (basisResolved) {
             uint256 guardBps = AusdStrategyStorageLib.fetch().basisGuardBps;
-            if (basisBps > guardBps) revert BasisOutOfRange(basisBps, guardBps);
+            uint8 shDec = SHMON.decimals();
+            uint256 oneShare = 10 ** shDec;
+            uint256 pps = SHMON.convertToAssets(oneShare);
+            uint256 impliedUsd = OracleHelpers.convertToUsd(pps, WMON_ORACLE, 18, WMON_ORACLE_HEARTBEAT);
+            uint256 oracleUsd = OracleHelpers.convertToUsd(oneShare, SHMON_ORACLE, shDec, SHMON_ORACLE_HEARTBEA
+            if (oracleUsd != 0 && impliedUsd < oracleUsd && basisBps > guardBps) {
+                revert BasisOutOfRange(basisBps, guardBps);
+            }
         }

 ... 776 unchanged lines ...
```

11. [Low] Strict fail-closed oracle valuation in AusdStrategy during withYieldAccrual causes vault-wide DoS

Status

Review status: True Positive
Remediation status: Acknowledged
Remediation note: Created by pipeline analysis
Acknowledgement reason: True Positive
Acknowledgement note: Fixed

Description

AusdStrategy's NAV uses strict OracleHelpers that revert on stale/failed oracles. Concrete's withYieldAccrual calls totalAllocatedValue() without try/catch, so any stale/failing feed on non-zero positions makes accrual revert and blocks deposits, withdrawals, epoch processing, and allocation until oracles recover or the strategy is disabled.

AusdStrategy._calculateTotalValue() normalizes balances and debts via OracleHelpers.convertToUsd/convertFromUsd, which deliberately revert when oracle data is stale, invalid, or the oracle call fails. BaseStrategy.totalAllocatedValue() returns this value directly without try/catch. Concrete vault operations are widely wrapped with withYieldAccrual, which executes _accrueYield() and calls IStrategyTemplate(strategy).totalAllocatedValue() for Active strategies without try/catch. Therefore, whenever AusdStrategy holds non-zero exposure and any relevant oracle (stablecoin, WMON/MON, or shMON) is stale/failing, accrual reverts and so do user/manager flows (deposits, mints, withdraws, redeems, allocate, processEpoch, and Permit2 deposits). Governance can restore liveness by toggling the strategy Inactive or removing it (these admin calls are not gated by withYieldAccrual), but until then the vault experiences a temporary DoS.

Severity

Impact Explanation: [Medium] The result is a significant but temporary DoS of core vault functionality (deposits, withdrawals, allocation, epoch processing). A clear operational workaround exists (disable the strategy), so it is not a prolonged freeze without workaround.

Likelihood Explanation: [Low] It requires an external oracle integration to be stale/failing or attacker-induced staleness (griefing). Such reliance on an integration behaving incorrectly classifies as low likelihood.

Exploitation

Exploitation Scenarios:

Scenario 1.

WMON/MON oracle becomes stale/failing while AusdStrategy has non-zero WMON or MON exposure (or positive net USD). A user attempts withdraw/redeem or the manager calls processEpoch; withYieldAccrual triggers valuation, which calls _wmonToUsd and reverts due to stale oracle. The operation reverts, blocking withdrawals and epoch processing until recovery or strategy deactivation.

Preconditions / Assumptions

- (a). AusdStrategy is Active in the vault
- (b). Strategy holds non-zero WMON or MON exposure, or net USD > 0
- (c). WMON oracle answer is stale/invalid or oracle call fails
- (d). User/manager calls a withYieldAccrual-wrapped function (e.g., withdraw/redeem/processEpoch)

Scenario 2.

Stablecoin oracle becomes stale/failing while AusdStrategy has positive net USD (requiring convertFromUsd) or any non-zero stablecoin balance/debt. A user attempts deposit/mint (including Permit2 deposit) or the allocator invokes allocate; withYieldAccrual triggers valuation, which calls _stablecoinToUsd/_stablecoinFromUsd and reverts. Deposits and allocation revert until recovery or strategy deactivation.

Preconditions / Assumptions

- (a). AusdStrategy is Active in the vault
- (b). Strategy has positive net USD requiring convertFromUsd or non-zero stablecoin balance/debt
- (c). Stablecoin oracle answer is stale/invalid or oracle call fails
- (d). User/manager calls a withYieldAccrual-wrapped function (e.g., deposit/mint/allocate/depositWithPermit2)

Scenario 3.

shMON oracle becomes stale/failing while AusdStrategy has non-zero shMON exposure (secondary collateral or idle). Any withYieldAccrual-wrapped action (withdraw/redeem/allocate/processEpoch) triggers valuation, which calls _shMonToUsd and reverts. User and manager flows revert until recovery or strategy deactivation.

Preconditions / Assumptions

- (a). AusdStrategy is Active in the vault
- (b). Strategy has non-zero shMON exposure (secondary collateral or idle)
- (c). shMON oracle answer is stale/invalid or oracle call fails
- (d). User/manager calls a withYieldAccrual-wrapped function (e.g., withdraw/redeem/allocate/processEpoch)

Proposed fix

AusdStrategy.sol

File: `src/strategies/AusdStrategy.sol`

Source

```
 // SPDX-License-Identifier: AGPL-3.0
 pragma solidity 0.8.28;

 import { IERC20 } from "@openzeppelin/contracts/token/ERC20/IERC20.sol";
 import { IERC20Metadata } from "@openzeppelin/contracts/token/ERC20/extensions/IERC20Metadata.sol";
 import { IERC4626 } from "@openzeppelin/contracts/interfaces/IERC4626.sol";
 import { SafeERC20 } from "@openzeppelin/contracts/token/ERC20/utils/SafeERC20.sol";
 import { Address } from "@openzeppelin/contracts/utils/Address.sol";

 import { Strings } from "@openzeppelin/contracts/utils/Strings.sol";

+import { IConcreteStandardVaultImpl } from "@concrete/interface/IConcreteStandardVaultImpl.sol";
 import { IWrappedToken } from "../interface/IWrappedToken.sol";
 import { IShMon } from "../interface/IShMon.sol";
 ... 713 unchanged lines ...
     /**
```

```
      * @notice Calculate total net strategy value in vault-asset units.
-     * @dev Value is computed as total collateral-like exposure minus total debt-like exposure, all normalized
+     * @dev Attempts strict valuation; on failure, falls back to vault-recorded allocated amount to preserve li
      */
     function _calculateTotalValue() internal view override returns (uint256) {
-        uint256 totalCollateralUsd = _calculateTotalCollateralUsd();
-        uint256 totalDebtUsd = _calculateTotalDebtUsd();
-        if (totalCollateralUsd <= totalDebtUsd) return 0;
-        return _stablecoinFromUsd(totalCollateralUsd - totalDebtUsd);
+        try this.strictTotalValue() returns (uint256 v) { return v; } catch {
+            return uint256(IConcreteStandardVaultImpl(VAULT()).getStrategyData(address(this)).allocated);
+        }
     }

+    /// @notice Strict valuation path used for external try/catch; reverts on oracle failures.
+    function strictTotalValue() external view returns (uint256 v) {
+        uint256 c = _calculateTotalCollateralUsd();
+        uint256 d = _calculateTotalDebtUsd();
+        v = c <= d ? 0 : _stablecoinFromUsd(c - d);
+    }
+
     /**
      * @notice Sum every asset-like position the strategy controls in USD terms.
 ... 449 unchanged lines ...
```

12. [Low] Donation-driven PPS skew in AusdStrategy.getBasisBps causes basis-guard DoS on allocate/deallocate

Status

Review status: True Positive
Remediation status: Acknowledged
Remediation note: Created by pipeline analysis
Acknowledgement reason: True Positive
Acknowledgement note: Fixed

Description

AusdStrategy’s basis guard compares shMON’s convertToAssets-implied price to a shMON oracle. If shMON accepts native MON donations that raise totalAssets without minting shares, the price-per-share (PPS) rises, widening the measured basis beyond the threshold and causing allocateFunds/deallocateFunds to revert until governance intervenes.

AusdStrategy.getBasisBps derives an implied shMON USD price from SHMON.convertToAssets(1 share) valued via the WMON oracle and compares it to a shMON USD oracle. If shMON accepts native MON transfers that count toward totalAssets without minting shares, external actors can inflate PPS by donating MON. This raises the implied price while the oracle price may lag, pushing the basis above the configured guard (default 200 bps). allocateFunds always enforces the guard and deallocateFunds enforces it unless the oracle call reverts (which donations do not cause). This can block both allocation and unwind operations, creating a liveness DoS until governance adjusts the guard or the oracle converges. The attack is most feasible at small TVL or when the system operates near the guard; at large TVL, the required donation becomes prohibitively costly. This is a griefing-style attack with no direct profit to the attacker.

Severity

Impact Explanation: [Medium] Blocking allocate/deallocate is a significant but temporary availability loss of core strategy functionality; a governance workaround (adjusting the guard) exists, preventing prolonged freezes.

Likelihood Explanation: [Low] Exploitation requires unrecoverable attacker donations (griefing) plus multiple aligned conditions (donation semantics and oracle lag). At large TVL, cost becomes prohibitive; overall, this yields low likelihood.

Exploitation

Exploitation Scenarios:

Scenario 1.

Early-TVL donation DoS: An attacker donates enough MON directly to shMON to increase PPS by >200 bps. The vault calls deallocateFunds to serve withdrawals; getBasisBps returns a basis above the guard, causing BasisOutOfRange and blocking unwinds and withdrawals until governance changes the guard or prices converge.

Preconditions / Assumptions

- (a). shMON accepts native MON transfers not via deposit() and includes them in totalAssets used by convertToAssets
- (b). The shMON USD oracle lags PPS changes but remains within heartbeat so oracle reads do not revert
- (c). basisGuardBps is configured (e.g., 200 bps by default)
- (d). Vault operations rely on allocateFunds/deallocateFunds to allocate/unwind

Scenario 2.

Top-up near threshold: With normal basis drift already close to the guard, the attacker donates a small amount to nudge PPS just over the threshold. Subsequent deallocateFunds or allocateFunds calls revert due to BasisOutOfRange, freezing operations until admin action or convergence.

Preconditions / Assumptions

- (a). All preconditions of Scenario 1
- (b). Normal operation exhibits small basis drift close to the configured guard, so a small donation suffices to cross the threshold

Scenario 3.

DoS during risk reduction: The strategy is near max LTVs; operator attempts to unwind. The attacker donates MON to push PPS above the guard, causing deallocateFunds to revert. If market moves continue adversely while unwinds are blocked, liquidation risk and losses can increase.

Preconditions / Assumptions

- (a). All preconditions of Scenario 1
- (b). Strategy is operating near max LTVs and a timely unwind is needed (e.g., due to WMON price movement)

Proposed fix

AusdStrategy.sol

File: `src/strategies/AusdStrategy.sol`

Source

```
 ... 840 unchanged lines ...
         uint256 oneShare = 10 ** shDec;
         uint256 pps = SHMON.convertToAssets(oneShare);
-        uint256 impliedShMonUsd = OracleHelpers.convertToUsd(pps, WMON_ORACLE, 18, WMON_ORACLE_HEARTBEAT);
+        uint256 impliedShMonUsd = OracleHelpers.convertToUsd(pps, WMON_ORACLE, WMON.decimals(), WMON_ORACLE_HEA
         uint256 oracleShMonUsd = OracleHelpers.convertToUsd(oneShare, SHMON_ORACLE, shDec, SHMON_ORACLE_HEARTBE

-        if (oracleShMonUsd == 0) return 0;
+        if (oracleShMonUsd == 0 || impliedShMonUsd >= oracleShMonUsd) return 0;

-        uint256 diff =
-            impliedShMonUsd > oracleShMonUsd ? impliedShMonUsd - oracleShMonUsd : oracleShMonUsd - impliedShMon
-        basisBps = (diff * LTVCalculations.BPS) / oracleShMonUsd;
+        basisBps = ((oracleShMonUsd - impliedShMonUsd) * LTVCalculations.BPS) / oracleShMonUsd;
     }

 ... 336 unchanged lines ...
```

Related findings

[Low] NAV counts arbitrary strategy idle balances and accrues before deposit pricing in AusdStrategy/GatedVaultImpl causes PPS manipulation and reduced shares/DoS

Description

Strategy NAV includes any idle balances at the strategy address; vault accrues this NAV before pricing deposits, so out-of-band transfers to the strategy can raise PPS just before deposits, reducing minted shares or triggering Permit2 minShares/limit failures.

AusdStrategy’s NAV calculation explicitly includes idle STABLECOIN, WMON, native MON, and SHMON balances held by the strategy, as well as pending unstake amounts. BaseStrategy has a payable receive(), allowing native MON donations; ERC20s can also be transferred directly to the strategy. GatedVaultImpl.depositWithPermit2 and the standard ERC-4626 deposit path both use withYieldAccrual, which calls the vault’s _accrueYield() to pull each strategy’s totalAllocatedValue into cachedTotalAssets before computing deposit shares. An attacker can donate counted assets to the strategy immediately before a deposit is executed, causing accrual to crystallize this donation as positive yield, increasing cachedTotalAssets and PPS. The subsequent deposit is priced against this higher PPS, minting fewer shares than expected or failing the Permit2 minShares/limit checks. While this can be used to grief or cause minor slippage extraction, it is typically uneconomic for the attacker (fees and pro-rata dilution), and profitable extraction requires rare conditions (very large attacker share and large in-flight deposit).

Severity

Impact Explanation: [Low] No principal theft or systemic unavailability; the impact is adverse pricing (fewer shares) and minor, temporary deposit reverts (Permit2 floors/limits), consistent with live-NAV semantics.

Likelihood Explanation: [Low] Attacks require pre-donating value with limited recoup, precise timing, and, for extraction, rare conditions (large attacker share and large in-flight deposit). Fees further erode incentives; most cases are uneconomic grief.

Exploitation

Exploitation Scenarios:

Scenario 1.

ERC-4626 deposit slippage extraction (ungated): Attacker front-runs a large visible ERC-4626 deposit by transferring a donation D of a counted asset to AusdStrategy. The victim deposit triggers withYieldAccrual, booking D as positive yield and raising cachedTotalAssets. The deposit then mints fewer shares at the inflated PPS, transferring economic value to existing holders (and fee recipients).

Preconditions / Assumptions

- (a). Deposits are reopened (depositsGated == false); ERC-4626 deposit path is available.
- (b). Attacker can observe public mempool to front-run.
- (c). AusdStrategy is active and included in vault accrual; its NAV includes idle balances and pending amounts.
- (d). Attacker can transfer ERC20s or MON to the strategy address.
- (e). A large user deposit relative to AUM is pending; attacker may also hold a non-trivial share of vault supply.

Scenario 2.

Permit2 fewer-shares or DoS (ungated): Attacker donates D to AusdStrategy and immediately executes the victim’s still-valid Permit2 signature via depositWithPermit2. Accrual books D as yield, PPS rises, and the deposit mints fewer shares (if minShares is lax) or reverts with Permit2SharesBelowMin (if minShares is tight), creating economic slippage or a one-off DoS.

Preconditions / Assumptions

- (a). Deposits are reopened (depositsGated == false); depositWithPermit2 is permissionless.
- (b). Victim’s Permit2 signature is valid (nonce unused, deadline not expired) and known to the attacker.
- (c). Attacker can donate counted assets to AusdStrategy and then call depositWithPermit2 using the victim’s signature.
- (d). AusdStrategy NAV includes idle balances; accrual runs before share pricing.

Scenario 3.

Gated batch Permit2 DoS: During gated operation, the manager's public transaction batches multiple depositWithPermit2 calls. An attacker front-runs with a small donation D to AusdStrategy; accrual raises PPS and/or total assets bounds, causing some deposits to fail minShares or deposit-limit checks, increasing operational friction and requiring retries.

Preconditions / Assumptions

- (a). Deposits remain gated (depositsGated == true); only WITHDRAWAL_MANAGER can execute depositWithPermit2.
- (b). Manager submits a public transaction batching multiple depositWithPermit2 calls.
- (c). Attacker can front-run with a small donation D to AusdStrategy.
- (d). Some batched deposits have tight minShares or are near deposit limits.

Proposed fix

AusdStrategy.sol

File: `src/strategies/AusdStrategy.sol`

Source

```
 ... 754 unchanged lines ...
             abi.decode(address(SECONDARY_COLLATERAL_ADAPTER()).functionStaticCall(secondaryColData), (uint256))
         total += _shMonToUsd(secondaryCol);
+        // SECURITY: The following raw on-contract idle balances include unsolicited transfers.
+        // Migrate to accounted-idle tracking: replace direct balance reads with storage-backed counters
+        // updated only inside strategy-controlled flows (pulls, swaps, wraps/unwraps, rewards, etc).

         total += _shMonToUsd(SHMON.balanceOf(address(this)));
 ... 430 unchanged lines ...
```

BaseStrategy.sol

File: `src/strategies/BaseStrategy.sol`

Source

```
 ... 1324 unchanged lines ...

     /// @notice Allow the strategy to receive native MON during unwrap or unstake flows.
+    // SECURITY: Native MON donations currently count toward NAV via idle-balance reads.
+    // Do not remove this (unwrap/unstake need it); adopt accounted-idle tracking in strategy NAV instead.
     receive() external payable { }

 ... 65 unchanged lines ...
```

RewardsAdapter.sol

File: `src/adapters/rewards/RewardsAdapter.sol`

Source

```
 ... 208 unchanged lines ...
     ///      never leave the local strategy and do not enter the trusted pool-manager callback path.
     /// @param params Reward swap parameters consumed by `Swapper`.
+    // SECURITY: Reward claims/swaps impact strategy idle balances. Add before/after hooks or
+    // bracket these flows in the strategy to update accounted-idle NAV instead of raw balances.
     /// @return outputAmount Amount of the requested output asset produced.
     function _processRewards(Swapper.RewardSwapParams calldata params) internal returns (uint256 outputAmount)
 ... 44 unchanged lines ...
```

13. [Low] Close-all repay instruction in EulerDebtAdapter that discards cushion in close-all flows causes collateral-withdraw revert and unwind DoS

Status

Review status: True Positive
Remediation status: Acknowledged
Remediation note: Created by pipeline analysis
Acknowledgement reason: True Positive
Acknowledgement note: Fixed

Description

When closeAllDebt is requested on the Euler leg, the adapter returns an exact repay amount equal to the visible debt and discards the available balance cushion. If the protocol's view debt is slightly stale vs. the state-changing repay calculation, a small residual debt can remain. A subsequent zero-LTV withdrawal then requests full collateral (per LTV fallback) and reverts due to non-zero debt, blocking deallocation/termination despite sufficient funds.

BaseStrategy._repay sets amount = availableBalance when closeAllDebt is true, then asks the adapter to resolve the instruction. EulerDebtAdapter.resolveRepayInstruction(amountAvailable, currentDebt, true) requires amountAvailable >= currentDebt but returns (currentDebt, false), discarding the cushion and instructing a fixed-amount repay. EulerDebtAdapter.repay approves exactly currentDebt and calls IEVault.repay(currentDebt). If the protocol accrues debt to 'now' during state change while debtOf(account) is a slightly stale view, repaying exactly currentDebt can leave tiny residual dust. Typical unwind batches then call withdrawCollateral with targetLtv=0 and amount=0. LTVCalculations.calculateSafeWithdrawal returns full collateral when targetLtv=0 and debt>0, so the withdraw reverts under protocol health checks. This creates a liveness failure for deallocation/exit even though the strategy held enough tokens to fully close. Additionally, AusdStrategy._terminatePositions uses repay(amount=0) as a Curvance sentinel; if EulerDebtAdapter is configured as the primary debt adapter, repay(0) reverts (assetsRepaid==0), breaking termination under that configuration.

Severity

Impact Explanation: [Medium] The issue causes significant but temporary availability loss of core unwind/deallocation functionality (transaction reverts) without direct loss of principal; operators can resolve by altering batch composition.

Likelihood Explanation: [Low] It depends on a combination of protocol accrual/view semantics and specific batch composition (targetLtv=0/amount=0, single-shot close-all). Operators can readily mitigate by reordering steps or adding a follow-up repay.

Exploitation

Exploitation Scenarios:

Scenario 1.

Close-all repay uses exactly currentDebt causing residual dust; the subsequent zero-LTV withdrawal requests full collateral and reverts, blocking the unwind batch.

Preconditions / Assumptions

- (a). Debt leg handled by EulerDebtAdapter
- (b). Strategy holds debt-token balance >= debtOf(view) + small cushion
- (c). IEVault.debtOf(account) can be slightly stale vs. accrual applied during repay
- (d). Unwind batch calls withdrawCollateral with targetLtv=0 and amount=0 immediately after repay

Scenario 2.

Termination path uses repay(0) sentinel assuming Curvance semantics; if EulerDebtAdapter is configured as primary, repay(0) reverts immediately, preventing termination.

Preconditions / Assumptions

- (a). Primary debt adapter configured as EulerDebtAdapter
- (b). Non-zero primary debt exists
- (c). Termination flow calls repay(token, 0) assuming Curvance full-repay sentinel

Scenario 3.

With closeAllDebt=true, if reserve clamping reduces amountAvailable slightly below currentDebt(view), resolveRepayInstruction reverts with InsufficientDebtCoverageForCloseAll and the batch fails early.

Preconditions / Assumptions

- (a). closeAllDebt=true on EulerDebtAdapter
- (b). BaseStrategy reserveAmount clamp reduces available balance below currentDebt(view)
- (c). No intermediate conversion step to increase available balance before repay

Proposed fix

EulerDebtAdapter.sol

File: `src/adapters/euler/EulerDebtAdapter.sol`

Source

```
 ... 101 unchanged lines ...
         IERC20(asset).forceApprove(address(DEBT_VAULT), amount);
         uint256 assetsRepaid = DEBT_VAULT.repay(amount, address(this));
+        // Reset approval to zero after repay to avoid leaving residual allowance.
+        IERC20(asset).forceApprove(address(DEBT_VAULT), 0);

         // Euler returns the amount of assets actually consumed for repayment. A zero result means
 ... 82 unchanged lines ...
                 revert InsufficientDebtCoverageForCloseAll(amountAvailable, currentDebt);
             }
-            return (currentDebt, false);
+            return (amountAvailable, false);
         }

         // For partial repayment, simply clamp by the smaller of available balance and current debt
         // because Euler does not impose a minimum residual-loan-size rule in this adapter.
         return (amountAvailable < currentDebt ? amountAvailable : currentDebt, false);
     }

     /// @inheritdoc IDebtAdapter
     function getProtocolName() external pure returns (string memory) {
         return "Euler";
     }
 }
```

AusdStrategy.sol

File: `src/strategies/AusdStrategy.sol`

Source

```
 ... 964 unchanged lines ...
                 revert InsufficientPrimaryDebtCoverage(wmonBalance, wmonDebt, wmonDebt - wmonBalance);
             }
-            bytes memory repayData = abi.encodeWithSelector(IDebtAdapter.repay.selector, address(WMON), uint256
-            _delegateCallVoid(address(PRIMARY_DEBT_ADAPTER()), repayData);
+            _repay(ProtocolPosition.PRIMARY, address(WMON), 0, 0, true);
             primaryDebtClosedThisCall = true;
         }
 ... 218 unchanged lines ...
```

Related findings

[Low] Unit mismatch for EVault.borrow return in EulerDebtAdapter.borrow causes borrow-step DoS

Description

EulerDebtAdapter.borrow treats IEVault.borrow's return (documented as debtIssued) as if it were assets received and compares it to the requested asset amount, creating a unit mismatch that can spuriously revert borrows or fail to detect shortfalls.

The in-repo IEVault interface documents borrow(amount, receiver) as returning debtIssued (a debt-accounting value), but EulerDebtAdapter.borrow assigns this return to a variable named assets and compares it directly to the requested underlying amount. If debtIssued is not guaranteed to equal the underlying assets transferred, the comparison is unit-inconsistent. This can cause otherwise-successful borrows to revert (if debtIssued < amount while full assets are transferred) or allow shortfalls to go undetected (if debtIssued >= amount but fewer assets are actually received). No balance-delta measurement or alternative verification exists in the adapter or BaseStrategy, and the Borrowed event emits the requested amount rather than the actual received amount.

Severity

Impact Explanation: [Low] The issue causes borrow-step failures or misleading monitoring but does not directly result in principal loss or prolonged fund freezing; it primarily affects availability of a borrow leg and operational reliability.

Likelihood Explanation: [Low] Manifestation depends on deployment using the Euler adapter and specific dependency semantics (non-1:1 debt units or transfer-level shortfalls), which are not guaranteed and are outside direct attacker control; typical lending behavior often reverts on shortfalls.

Exploitation

Exploitation Scenarios:

Scenario 1.

If EVault.borrow returns non-1:1 accounting units (e.g., debt shares) smaller than the requested underlying amount, EulerDebtAdapter.borrow compares these against the requested amount and reverts, preventing the borrow step from executing and causing a DoS on Euler-based borrow flows.

Preconditions / Assumptions

- (a). The strategy is configured to use EulerDebtAdapter for a borrow leg.
- (b). IEVault.borrow returns debtIssued that is not 1:1 with underlying asset units (e.g., debt shares exist or an index is applied).
- (c). A borrow step is executed via the strategy's BaseStrategy._borrow path.

Scenario 2.

If EVault.borrow returns debtIssued in underlying units while transferring fewer assets than requested (e.g., fee withheld on transfer), the adapter's check passes and later steps that assume full receipt may revert due to insufficient balance, causing operational failures and misleading Borrowed events.

Preconditions / Assumptions

- (a). The strategy is configured to use EulerDebtAdapter for a borrow leg.
- (b). IEVault.borrow returns debtIssued in underlying units equal to or greater than the requested amount, but the vault transfers fewer assets than requested to the receiver (e.g., transfer-level fee withheld).
- (c). No balance-delta measurement is performed after borrow.

Scenario 3.

Borrowed events always emit the requested amount instead of the actual assets received, degrading monitoring accuracy and complicating incident diagnosis during partial or anomalous transfers.

Preconditions / Assumptions

- (a). Any mismatch between requested amount and assets actually received occurs during EulerDebtAdapter.borrow.
- (b). Off-chain monitoring relies on the Borrowed event for amount reporting.

Proposed fix

EulerDebtAdapter.sol

File: `src/adapters/euler/EulerDebtAdapter.sol`

Source

```
 ... 88 unchanged lines ...
     /// @inheritdoc IDebtAdapter
     function borrow(address asset, uint256 amount) external {
-        // Euler returns the amount of assets actually borrowed from the debt vault, so compare
-        // that value directly against the requested amount.
-        uint256 assets = DEBT_VAULT.borrow(amount, address(this));
-        if (assets < amount) revert DebtBorrowFailed(amount, assets);
-        emit Borrowed(asset, amount, address(this));
+        // Measure balance delta to validate actual assets received regardless of debtIssued semantics.
+        uint256 beforeBal = IERC20(asset).balanceOf(address(this));
+        DEBT_VAULT.borrow(amount, address(this));
+        uint256 afterBal = IERC20(asset).balanceOf(address(this));
+        uint256 received = afterBal > beforeBal ? afterBal - beforeBal : 0;
+        if (received < amount) revert DebtBorrowFailed(amount, received);
+        emit Borrowed(asset, received, address(this));
     }

 ... 105 unchanged lines ...
```

[Informational] Missing liquidity clamp in EulerDebtAdapter.getMaxBorrowAmount causes borrow-batch reverts during auto-quote

Description

EulerDebtAdapter.getMaxBorrowAmount returns an LTV-based max without clamping to live EVault liquidity. BaseStrategy._borrow uses this value when amount == 0 and EulerDebtAdapter.borrow enforces exact fill, so if EVault.cash() is lower than the quoted amount the borrow reverts and the entire batch fails atomically. This is a reliability/UX issue, not a funds-loss bug.

The EulerDebtAdapter helper getMaxBorrowAmount computes a maximum borrow based on collateral/debt and target LTV plus maxDebt, but it does not clamp the result to the Euler EVault's current liquidity (IEVault.cash()). BaseStrategy._borrow calls this helper when amount == 0 and then immediately delegates to EulerDebtAdapter.borrow with the returned amount. EulerDebtAdapter.borrow requires exact fulfillment and reverts if the EVault supplies fewer assets than requested. Consequently, when live liquidity is below the adapter's quoted amount, the borrow step reverts and the entire allocation/rebalance batch fails. All state changes are reverted atomically, so there is no funds loss or stranded state. The issue can also be induced by adversarial front-running that drains liquidity just before execution. Operators can mitigate by passing explicit amounts or setting a conservative maxDebt reflecting live liquidity; alternatively, clamping to EVault.cash() in getMaxBorrowAmount would reduce avoidable reverts.

Severity

Impact Explanation: [Low] The effect is a transient transaction revert (allocation/rebalance batch fails) with no funds loss or persistent state changes; retrying with corrected parameters or after liquidity improves resolves the issue.

Likelihood Explanation: [Low] Multiple multiplicative preconditions are required (use of amount == 0, insufficient liquidity, non-conservative maxDebt). The adversarial case is griefing at attacker cost without clear profit; operator misconfiguration is also low likelihood.

Exploitation

Exploitation Scenarios:

Scenario 1.

Natural liquidity shortfall: The strategy uses borrow with amount == 0; getMaxBorrowAmount returns an LTV-based amount larger than EVault.cash(); EulerDebtAdapter.borrow attempts to borrow the full amount, the EVault cannot fulfill it, and the adapter reverts, causing the entire batch to fail.

Preconditions / Assumptions

- (a). Strategy integrates EulerDebtAdapter for the debt leg
- (b). Borrow command uses amount == 0 (auto-quote path)
- (c). IEVault.cash() is lower than the LTV-based headroom
- (d). maxDebt is not set conservatively to bound the auto-quote below current liquidity

Scenario 2.

Adversarial front-run: An attacker observes the pending transaction and reduces EVault liquidity beforehand; the strategy's getMaxBorrowAmount (amount == 0) returns an LTV-based amount; borrow then fails due to newly lowered cash(), reverting the batch.

Preconditions / Assumptions

- (a). Public mempool; attacker can front-run with a higher-priority transaction
- (b). Borrow command uses amount == 0 (auto-quote path)
- (c). Attacker reduces EVault.cash() between the quote (staticcall) and borrow execution
- (d). Euler EVault behaves correctly per its interface

Scenario 3.

Misconfigured maxDebt: The off-chain orchestrator sets a very high maxDebt and uses amount == 0; getMaxBorrowAmount returns a value not bounded by live liquidity; borrow attempts that amount and reverts, failing the batch.

Preconditions / Assumptions

- (a). Operator/off-chain orchestrator sets maxDebt very high
- (b). Borrow command uses amount == 0 (auto-quote path)
- (c). EVault liquidity is lower than the returned LTV-based amount
- (d). No on-chain clamp to EVault.cash() in adapter getMaxBorrowAmount

Proposed fix

EulerDebtAdapter.sol

File: `src/adapters/euler/EulerDebtAdapter.sol`

Source

```
 ... 152 unchanged lines ...
         // Euler does not impose the Curvance-style minimum loan-size rule here, so the generic
         // oracle-aware LTV calculation is sufficient for maximum borrow sizing.
-        return LTVCalculations.calculateMaxBorrow(
+        uint256 finalBorrow = LTVCalculations.calculateMaxBorrow(
             collateral,
             debt,
             COLLATERAL_ORACLE,
             DEBT_ORACLE,
             COLLATERAL_DECIMALS,
             DEBT_DECIMALS,
             targetLtv,
             maxDebt,
             COLLATERAL_ORACLE_HEARTBEAT,
             DEBT_ORACLE_HEARTBEAT
         );
+        // Clamp to live EVault liquidity so auto-quote borrows remain realistically fillable.
+        uint256 availableLiquidity = DEBT_VAULT.cash();
+        if (finalBorrow > availableLiquidity) {
+            finalBorrow = availableLiquidity;
+        }
+
+        return finalBorrow;
     }
```

```
... 34 unchanged lines ...
```

14. [Low] Permissionless full-balance distribution with owner-favoring rounding in VaultFeeAllocator causes partner underpayment due to timing

Status

Review status: True Positive
Remediation status: Acknowledged
Remediation note: Created by pipeline analysis
Acknowledgement reason: True Positive
Acknowledgement note: Fixed

Description

VaultFeeAllocator distributes the full unreserved balance permissionlessly and uses per-call floor division for partner shares with the remainder paid to the owner. Without carry-forward or a minimum threshold, repeated small-balance distributions can underpay partners relative to configured BPS, and the owner can micro-stream amounts to bias payouts.

VaultFeeAllocator.distributeFees is permissionless and distributes the entire unreserved token balance. For each active partner, the payout is computed as floor(distributableBalance * bps / 10,000), and any remainder is sent to the owner in the same call. There is no carry-forward of fractional partner entitlements and no minimum distribution threshold. This creates call-cadence dependence: if distributions are triggered when balances are small, per-partner amounts round down to zero and the owner receives the entire small amount as dust. While third parties cannot split a large balance (the function always distributes the full available balance), they can trigger distributions during naturally occurring small-balance windows to bias dust to the owner. Separately, the owner-only distributeFeesAmount allows streaming a large balance in micro-chunks that round partner payouts to zero, directing most streamed value to the owner.

Severity

Impact Explanation: [Medium] Direct, potentially material loss of fees to partners can occur if the owner streams a large balance in micro-amounts that always round partner shares to zero. Third-party-triggered dust draining typically remains dust-level per call but can cumulatively reduce partner payouts if small-balance windows are frequent.

Likelihood Explanation: [Low] Highest-impact case requires trusted role (owner) misuse. Third-party scenarios are griefing with no direct profit and rely on uncommon system states (frequent small-balance windows). Attackers cannot split a large balance via the permissionless path, which always distributes the full unreserved balance.

Exploitation

Exploitation Scenarios:

Scenario 1.

Third-party dust draining on frequent micro-balance windows: An external caller monitors the allocator and calls distributeFees(token) whenever a small positive balance appears. Because partner shares round down to zero for tiny amounts, the entire small balance is paid to the owner each time. Repeating this prevents accumulation to fairer amounts and underpays partners relative to BPS.

Preconditions / Assumptions

- (a). VaultFeeAllocator is the performance-fee recipient and holds token T
- (b). Active partners with nonzero BPS are configured
- (c). distributeFees is permissionless and distributes the full unreserved balance
- (d). System behavior produces frequent small-balance windows for token T

Scenario 2.

Owner micro-streaming via distributeFeesAmount: The owner repeatedly calls distributeFeesAmount(token, amount) with amounts deliberately below per-partner rounding thresholds. Each call rounds partner shares to zero and pays the streamed amount to the owner. Streaming a large balance in many such calls materially underpays partners.

Preconditions / Assumptions

- (a). Allocator holds a large balance of a token
- (b). Owner controls distributeFeesAmount (owner-only)
- (c). Owner chooses micro-amounts below per-partner rounding thresholds

Scenario 3.

Multi-token dust draining: A third party monitors multiple tokens held by the allocator and immediately calls distributeFees for any token when small balances appear. This repeatedly routes dust across tokens to the owner, reducing partner payouts over time.

Preconditions / Assumptions

- (a). Allocator receives performance fees in multiple tokens
- (b). Active partners with nonzero BPS are configured
- (c). distributeFees is permissionless and distributes the full unreserved balance
- (d). Small-balance windows occur intermittently across these tokens

Proposed fix

VaultFeeAllocator.sol

File: `src/fee/VaultFeeAllocator.sol`

Source

```
 ... 249 unchanged lines ...
     ////////////////////////////////////////////////////////////*/

+    /*
+     * FIXME(security): Per-call floor division and permissionless timing can underpay partners on small-balanc
+     * distributions. Complete remediation should:
+     * 1) Accrue per-recipient entitlements in fixed-point on new inflows only (delta = available - accounted).
+     * 2) Settle matured whole-token dues; carry forward fractional remainders across calls.
+     * 3) Track accountedFreeBalance per token and exclude it from _availableBalance and emergencyWithdraw.
+     * 4) Apply the same settlement to distributeFeesAmount; keep pending-fee reservations unchanged.
+     */
     /**
      * @notice Distribute the full unreserved token balance to active partners.
 ... 175 unchanged lines ...
     * @param amount Amount to withdraw.
     */
+    // NOTE: After accrual redesign, enforce amount <= (balance - totalPendingPartnerFees[token] - accountedFre
     function emergencyWithdraw(address token, address recipient, uint256 amount) external onlyOwner {
         if (token == address(0) || recipient == address(0)) revert ZeroAddress();
 ... 157 unchanged lines ...
     /// @param token Token whose free balance is requested.
     /// @return availableBalance Token balance minus all reserved pending-fee obligations.
+    // NOTE: After implementing fixed-point accrual, also subtract accountedFreeBalance[token] from available b
     function _availableBalance(address token) private view returns (uint256) {
         uint256 totalBalance = IERC20(token).balanceOf(address(this));
 ... 19 unchanged lines ...
```

Related findings

[Informational] Missing balance-delta checks in VaultFeeAllocator payout and retry logic causes silent underpayment and unreserved shortfalls

Description

VaultFeeAllocator treats any successful ERC-20 trySafeTransfer as full settlement without verifying how much was actually debited from the allocator. With nonstandard ERC-20s that short-deliver on successful transfers, partners are underpaid

while the shortfall remains unreserved and can later be emergency-withdrawn by the owner or redistributed. Pending-fee retries can also clear reservations by the nominal amount while sending less.

VaultFeeAllocator uses _tryTransfer (SafeERC20.trySafeTransfer) and treats a true return as full payment in both _payOrReserve and _retryPendingFees. It does not measure balance deltas, so when a token returns success while actually transferring less than the requested amount, the allocator emits PartnerPaid for the nominal amount and does not reserve the shortfall. In the retry path, it additionally reduces pendingPartnerFees and totalPendingPartnerFees by the nominal amount on success, even if fewer tokens were sent, immediately freeing a shortfall as unreserved. emergencyWithdraw(token) allows the owner to withdraw any unreserved balance. With exact-transfer tokens this is fine, but for nonstandard ERC-20s that short-deliver on success (e.g., partial-send, phantom-success), partners are silently underpaid and the shortfall can be extracted or reallocated.

Severity

Impact Explanation: [Medium] Partners can suffer direct, material loss of fee payouts (underpayment or cleared reservations without full payment). This impacts fees/yield rather than principal.

Likelihood Explanation: [Low] All scenarios rely on nonstandard ERC-20 behaviors (partial-send or phantom success). Monetization typically also requires owner emergencyWithdraw; the permissionless retry path is largely griefing without direct attacker profit.

Exploitation

Exploitation Scenarios:

Scenario 1.

Distribution with a partial-send-on-success token: distributeFees computes partner amounts from the initial balance and calls transfer(amount) for each partner. The token returns success but debits less than amount; the allocator logs PartnerPaid for the full amount and does not reserve the shortfall. The leftover remains in the contract as unreserved balance and the owner can emergencyWithdraw it.

Preconditions / Assumptions

- (a). At least one active partner with nonzero allocation
- (b). Allocator holds a balance of token T
- (c). Token T short-delivers on successful transfer (debited by less than requested amount without revert)
- (d). A caller invokes distributeFees(T)
- (e). Owner later calls emergencyWithdraw to extract the unreserved shortfall

Scenario 2.

Permissionless retry clears a partner's pending claim nominally: retryPendingFeesAmount(token, partner, pending) is called; the token returns success while sending less. The allocator reduces pendingPartnerFees and totalPendingPartnerFees by the full nominal amount, freeing the shortfall as unreserved immediately. Later, the owner can emergencyWithdraw or future distributions can consume this shortfall, underpaying the partner.

Preconditions / Assumptions

- (a). pendingPartnerFees[T][P] > 0 for some partner P
- (b). Allocator holds token T sufficient to attempt transfer
- (c). Token T short-delivers on successful transfer (without revert)
- (d). Any address calls retryPendingFeesAmount(T, P, pending amount)
- (e). Owner later may call emergencyWithdraw or future distributions run to consume the freed shortfall

Scenario 3.

Phantom-success token: distributeFees is called on a token that returns success but does not debit the allocator at all. The allocator logs PartnerPaid events and considers all partners paid, yet the full balance remains unreserved and the owner can emergencyWithdraw it.

Preconditions / Assumptions

- (a). Allocator holds a balance of token U

- (b). Token U returns success but does not debit the sender (phantom success)
- (c). A caller invokes distributeFees(U)
- (d). Owner calls emergencyWithdraw to extract the full unreserved balance

Proposed fix

VaultFeeAllocator.sol

File: `src/fee/VaultFeeAllocator.sol`

Source

```
 ... 535 unchanged lines ...
     /// @param amount Transfer amount.
     /// @return success Whether the transfer call reported success.
+    // SECURITY: Do not treat a 'true' return as proof that `amount` was fully debited from this contract.
+    // Safe mitigation requires balance-delta checks in callers:
+    // - Measure pre/post balances around this call.
+    // - If debited == 0, treat as failure and reserve full `amount`.
+    // - If 0 < debited < amount, emit PartnerPaid(debited) and reserve the shortfall (amount - debited).
+    // - Only when debited == amount should this be considered fully settled.
     function _tryTransfer(address token, address recipient, uint256 amount) private returns (bool) {
         return IERC20(token).trySafeTransfer(recipient, amount);
     }

     /// @notice Try to pay a recipient immediately or reserve the amount for later retry.
     /// @param token Token being paid.
     /// @param recipient Intended payout recipient.
     /// @param amount Amount to pay or reserve.
+    // SECURITY FIX NEEDED: Measure ERC20 balance deltas here instead of trusting boolean success:
+    // uint256 pre = IERC20(token).balanceOf(address(this));
+    // bool ok = _tryTransfer(token, recipient, amount);
+    // uint256 debited = pre > IERC20(token).balanceOf(address(this)) ? pre - IERC20(token).balanceOf(address(t
+    // - If !ok or debited == 0: reserve full `amount`.
+    // - If 0 < debited < amount: emit PartnerPaid(debited) and reserve (amount - debited). If debited == amoun
     function _payOrReserve(address token, address recipient, uint256 amount) private {
         if (_tryTransfer(token, recipient, amount)) {
 ... 21 unchanged lines ...
     /// @param partner Partner whose pending fees should be retried.
     /// @return success Whether the retry transfer succeeded.
+    // SECURITY FIX NEEDED: Reduce pending amounts by actual debited value, not by nominal `amount`.
+    // Suggested flow:
+    // uint256 pre = IERC20(token).balanceOf(address(this));
+    // bool ok = _tryTransfer(token, partner, amount);
+    // uint256 debited = pre > IERC20(token).balanceOf(address(this)) ? pre - IERC20(token).balanceOf(address(t
+    // If debited == 0: treat as failure. Else: pendingPartnerFees[token][partner] -= debited; totalPendingPart
     function _retryPendingFees(address token, address partner, uint256 amount) private returns (bool success) {
         uint256 pendingAmount = pendingPartnerFees[token][partner];
 ... 40 unchanged lines ...
```

15. [Low] Missing disable path for Euler collaterals in BaseStrategy causes migration blockage and potential account lockout under large collateral lists

Status

Review status: True Positive
Remediation status: Acknowledged
Remediation note: Created by pipeline analysis
Acknowledgement reason: True Positive
Acknowledgement note: Fixed

Description

The strategy automatically enables Euler collaterals/controllers on deposit but exposes no governance-accessible path to disable them later. Over successive migrations, enabled collaterals can accumulate. If the external EVC cap is not enforced or is raised, the large collaterals array may cause IEVault.checkAccountStatus to run out of gas and revert, potentially bricking the account. Even with a strict cap (e.g., 10), future migrations can be blocked once the cap is reached.

EulerCollateralAdapter.deposit() calls EVC.enableCollateral and EVC.enableController for the strategy account, enabling collateral/controller on first use. EulerCollateralAdapter.deposit() calls EVC.enableCollateral and EVC.enableController. EulerCollateralAdapter also implements disableCollateral(), but it is onlyStrategy-gated and there is no command wrapper in BaseStrategy to reach it, so governance cannot disable previously enabled Euler collaterals through the normal command surface. implements disableCollateral(), onlyStrategy-gated, command wrapper in BaseStrategy. Over multiple adapter migrations, this can accumulate enabled collaterals in EVC state. The EVC passes the array of enabled collaterals to the controller vault's IEVault.checkAccountStatus; if the array becomes large (e.g., due to absent or increased external caps), the status check can consume excessive gas and revert, which the interface warns may render the account unusable. IEVault.checkAccountStatus, the interface warns may render the account unusable. Even if the EVC enforces a small cap (e.g., 10), once that many collaterals are enabled, enabling a new collateral reverts and migration is blocked because there is no pathway in the strategy to disable older collaterals. While UUPS upgrades can add a disable wrapper to remediate, the current design introduces an operational DoS risk and a conditional lockout risk dependent on external EVC behavior.

Severity

Impact Explanation: [Medium] The issue can break important operational functionality (migration/upgrade) and, under certain external conditions, significantly degrade availability; direct principal loss is not demonstrated, and complete lockout depends on external integration behavior.

Likelihood Explanation: [Low] Exploitation relies on uncommon states (many migrations without cleanup) and/or external integration behavior (EVC cap not enforced or increased); governance can also mitigate via upgrades adding a disable-collateral pathway.

Exploitation

Exploitation Scenarios:

Scenario 1.

Large enabled-collateral list accumulates across many migrations; a subsequent EVC-triggered account status check passes a very large collaterals array to IEVault.checkAccountStatus, which reverts due to excessive gas use, causing operations that require EVC checks to fail and potentially bricking the strategy account.

Preconditions / Assumptions

- (a). External EVC does not strictly enforce a low cap on enabled collaterals or increases it
- (b). IEVault.checkAccountStatus complexity scales with the number of collaterals
- (c). Many migrations to different Euler collaterals occurred, each enabling a new collateral
- (d). No governance-accessible wrapper exists to disable previously enabled collaterals
- (e). An operation triggers EVC to perform an account status check

Scenario 2.

With a strict external cap (e.g., 10), repeated migrations enable 10 collaterals; attempting to enable an 11th collateral reverts due to the cap; governance cannot disable older collaterals via the strategy (no wrapper), blocking migrations and emergency responses.

Preconditions / Assumptions

- (a). External EVC enforces a low cap on enabled collaterals (e.g., 10)
- (b). At least that many migrations have enabled distinct Euler collaterals without cleanup
- (c). No governance-accessible wrapper exists to disable previously enabled collaterals
- (d). Attempt to enable a new collateral requires EVC.enableCollateral, which reverts at the cap

Scenario 3.

Within a small cap (e.g., 10), if per-collateral logic in checkAccountStatus is heavy and network gas is spiking, EVC-triggered status checks intermittently fail for operations that require checks, degrading liveness at critical moments.

Preconditions / Assumptions

- (a). External EVC enforces a small cap (e.g., up to 10)
- (b). Enabled collaterals count is near the cap (e.g., 8–10)
- (c). IEVault.checkAccountStatus performs non-trivial per-collateral work
- (d). Network congestion/gas spikes occur during operations that trigger EVC checks

Proposed fix

EulerCollateralAdapter.sol

File: `src/adapters/euler/EulerCollateralAdapter.sol`

Source

```
 ... 112 unchanged lines ...
     ////////////////////////////////////////////////////////////////*/

+    // TODO(security): Add a small strategy-level cap on enabled Euler collaterals.
+    // Before enabling a new collateral (both here and in enableCollateral()), read
+    // EVC.getCollaterals(address(this)).length and revert if it exceeds a low bound (e.g., 2–3).
+    // This prevents unbounded accumulation across migrations and large checkAccountStatus arrays.
     /// @inheritdoc ICollateralAdapter
     function deposit(address asset, uint256 amount) external {
         // Euler requires the account to opt into the debt vault as a controller before that vault
         // can manage borrowing power against this collateral position.
         if (!EVC.isControllerEnabled(address(this), address(DEBT_VAULT))) {
             EVC.enableController(address(this), address(DEBT_VAULT));
         }

         // Euler also tracks whether a given vault balance is enabled as collateral. Enabling it
         // here keeps first deposit flows self-contained and avoids a separate setup transaction.
         if (!EVC.isCollateralEnabled(address(this), address(COLLATERAL_VAULT))) {
             EVC.enableCollateral(address(this), address(COLLATERAL_VAULT));
         }

         // Approve only the amount being supplied so the Euler vault can pull the collateral asset
+        // NOTE: Apply the same small-cap check in enableCollateral() before calling EVC.enableCollateral
+        // to keep both paths consistent and bounded.
         // without leaving a broader-than-needed token allowance behind.
         IERC20(asset).forceApprove(address(COLLATERAL_VAULT), amount);
         uint256 shares = COLLATERAL_VAULT.deposit(amount, address(this));

         // Euler returns the vault shares minted for the deposit. A zero-share result means the
         // deposit did not create usable collateral and should be treated as a failure.
         if (shares == 0) revert CollateralDepositFailed();
         emit CollateralDeposited(asset, amount, address(this));
     }

     /// @inheritdoc ICollateralAdapter
     function withdraw(address asset, uint256 amount) external {
         // Euler withdraw returns the amount of underlying assets released from the vault rather
         // than the number of shares burned, so compare the returned assets directly.
         uint256 assets = COLLATERAL_VAULT.withdraw(amount, address(this), address(this));
         if (assets < amount) revert CollateralWithdrawalFailed(amount, assets);
         emit CollateralWithdrawn(asset, assets, address(this));
     }

     /// @inheritdoc ICollateralAdapter
     function enableCollateral() external onlyStrategy {
         // Keep this call idempotent so repeated operator batches do not fail when collateral is
```

```
+        // TODO(security): Enforce the same small-cap bound as in deposit() by checking
+        // EVC.getCollaterals(address(this)).length before enabling a new collateral.
         // already enabled for the strategy account.
         if (!EVC.isCollateralEnabled(address(this), address(COLLATERAL_VAULT))) {
 ... 93 unchanged lines ...
```

BaseStrategy.sol

File: `src/strategies/BaseStrategy.sol`

Source

```
 ... 1282 unchanged lines ...
         ////////////////////////////////////////////////////////////*/

+    // TODO(security): Add admin-only EVC/EVault maintenance utilities to prune legacy Euler state:
+    // - evcDisableControllers(address[] controllerVaults): for each vault, IEVault(v).disableController().
+    // - evcDisableCollaterals(address evc, address[] collateralVaults):
+    //     IEVC(evc).disableCollateral(address(this), vault).
+    // - evcEnableController(address evc, address controllerVault):
+    //     IEVC(evc).enableController(address(this), controllerVault).
+    // Run in sequence: disableControllers -> disableCollaterals -> enableController to avoid controller-select
+    // Gate these functions with onlyStrategyAdmin.
     /**
      * @dev The admin section collects the operational controls that auditors usually classify as privileged ac
 ... 107 unchanged lines ...
```

Warnings

1. [Medium] Unsupported migration selector handling in BaseStrategy/AusdStrategy upgrade causes strategy DoS

Status

Review status: Unresolved
Remediation status: Unremediated
Remediation note: Created by pipeline analysis

Description

BaseStrategy's reinitializer only accepts a specific migrateStorage(...) selector, while AusdStrategy does not override migration. The repository's legacy storage-layout migration script encodes an unsupported child-specific selector, so reinitializeUpgrade consumes a version but performs no state migration. This can leave basisGuardBps = 0 and/or the WMON unwrapper unset, causing allocate/deallocate and unwrap-dependent conversions to revert until an admin repair. In realistic vault setups, this can block withdrawals if deallocation is required.

During an upgrade/migration, BaseStrategy._migrateUpgrade only executes when the payload selector matches migrateStorage(address,address,address,address,address). AusdStrategy does not implement its own migration hook and initializes its child-specific storage (basisGuardBps) only in initialize(). The repository's legacy storage-layout migration script encodes a child-specific migrateAusdStrategy(...) selector and updateAdapters(...), which the on-chain BaseStrategy ignores. As a result, reinitializeUpgrade consumes the reinitializer version but leaves Ausd-specific storage unchanged: migrateStorage(address,address,address,address,address) may remain 0 and the WMON unwrapper may remain unset. With basisGuardBps = 0, AusdStrategy.allocateFunds reverts under normal oracle conditions, and deallocateFunds also reverts unless oracles fail (best effort bypass). If the WMON unwrapper is unset, unwrap-dependent steps revert. In vaults where this strategy is the sole or dominant allocation, failed deallocations can block withdrawals until a privileged fix (e.g., setBasisGuardBps, set/deploy unwrapper, or re-run reinitializer with the correct selector) is executed.

Severity

Impact Explanation: [High] In realistic vault configurations (single or dominant strategy), failed deallocations caused by basisGuardBps = 0 can block withdrawals entirely, matching core protocol functionality unusable.

Likelihood Explanation: [Low] The issue requires a trusted operator/admin to run the legacy storage-layout migration script that encodes an unsupported selector and fail to apply an immediate repair; this is a privileged operational mistake.

Exploitation

Exploitation Scenarios:

Scenario 1.

Deallocation blocked under healthy oracles: An operator runs the legacy storage-layout migration with an unsupported selector; the reinitializer no-ops, leaving basisGuardBps = 0. A subsequent deallocateFunds() call enforces the guard against a non-zero basis and reverts, preventing unwinds and potentially blocking withdrawals when the vault depends on this strategy for liquidity.

Preconditions / Assumptions

- (a). Legacy AusdStrategy proxy without AusdStrategyStorage previously initialized (basisGuardBps defaults to 0)
- (b). Operator/admin uses the legacy storage-layout migration script that encodes an unsupported child-specific selector
- (c). BaseStrategy ignores the payload and consumes the reinitializer version without applying state changes
- (d). Oracles are healthy and report a non-zero shMON/W-MON basis
- (e). The vault relies on this strategy's deallocation for withdrawals (single or dominant strategy configuration)

Scenario 2.

Allocation bricked after migration: After the same mis-migration, allocateFunds() triggers the basis guard with basisGuardBps = 0 under normal oracles, reverting and preventing new or expanded positions, halting strategy performance.

Preconditions / Assumptions

- (a). Legacy AusdStrategy proxy without AusdStrategyStorage previously initialized (basisGuardBps defaults to 0)
- (b). Operator/admin uses the legacy storage-layout migration script that encodes an unsupported child-specific selector
- (c). BaseStrategy ignores the payload and consumes the reinitializer version without applying state changes
- (d). Oracles are healthy and report a non-zero shMON/W-MON basis

Scenario 3.

Conversions fail due to missing WMON unwrapper: The same mis-migration leaves the WMON unwrapper unset; unwrap-dependent conversions (e.g., WMON → MON → shMON) revert, breaking loop/unwind steps and further degrading availability.

Preconditions / Assumptions

- (a). Legacy AusdStrategy proxy where the new WMON unwrapper field was never initialized
- (b). Operator/admin uses the legacy storage-layout migration script that encodes an unsupported child-specific selector
- (c). BaseStrategy ignores the payload and consumes the reinitializer version without applying state changes
- (d). Subsequent batches require unwrap-dependent conversions (e.g., WMON → MON → shMON)

Proposed fix

upgrade-strategy-storage-layout.cli.ts

File: `script/ts/upgrade-strategy-storage-layout.cli.ts`

Source

```
... 103 unchanged lines ...
};
```

```
+// IMPORTANT: BaseStrategy.reinitializeUpgrade only recognizes the top-level payload
+// selector migrateStorage(address,address,address,address,address). Do not rely on
+// child-specific wrappers; encode migrateStorage(...) directly for the migration call.
 const BASE_STRATEGY_STORAGE_LOCATION =
     "0xb06bc46f4ae71579c5dc76296e5c48e10478929da6d808cf30c72da6b4118c00" as const;
 ... 158 unchanged lines ...
 ] as const;

+// NOTE: For the reinitializer migration, build and pass migrateStorage(...) directly.
+// After the reinitializer, if repairBasisGuard is true (basisGuardBps was zero),
+// call setBasisGuardBps(migrationState.basisGuardBps) as a separate admin write
+// to seed the AUSD-specific guard.
 const migrationAbi = [
     {
 ... 852 unchanged lines ...
         }

+        // FIXME: Build migrationData using migrateStorage(address,address,address,address,address)
+        // and pass it directly to reinitializeUpgrade. Then, if migrationState.repairBasisGuard
+        // is true, call setBasisGuardBps(migrationState.basisGuardBps) as a separate write.
         const nextVersion = initializedVersion + 1n;
         const baseMigrationData = encodeFunctionData({
 ... 138 unchanged lines ...
```

Related findings

[Low] Changing ERC-7201 storage slot constant in AusdStrategyStorageLib causes stranded async unstake workers and incorrect termination gating

Description

AusdStrategyStorageLib anchors AUSD-specific state to a hardcoded ERC-7201 storage slot derived from a placeholder namespace. If a future implementation changes this STORAGE_LOCATION constant, deployed proxies will read a fresh empty namespace, losing access to basisGuardBps and async-unstake bookkeeping. This can strand pending unstake workers and allow termination to proceed without reclaiming funds; operations may also be temporarily blocked by a zeroed basis guard.

AusdStrategyStorageLib uses a fixed bytes32 STORAGE_LOCATION to bind AUSD-specific state (basisGuardBps, totalPendingUnstakeMon, activeUnstakeRequestIds, unstakeRequests). The current code and tooling consistently use a placeholder-derived slot, which is safe as long as the constant never changes. However, if a future upgrade alters this constant (e.g., by re-deriving from a 'fixed' namespace string), the proxy will read and write a different, empty namespace for AUSD state. Consequences include: (1) matured async-unstake workers are no longer discoverable, so _finalizeMaturedUnstakes() never completes them; (2) _assertNoPendingShMonadUnstakes() reads zeros and does not block terminatePositions(), so termination can proceed without reclaiming funds; (3) basisGuardBps appears as zero and allocate/deallocate revert until governance resets it; and (4) NAV under-reporting by excluding pending MON. This is not attacker-exploitable; it requires a privileged upgrade mistake. Funds are recoverable with a follow-up upgrade (e.g., adding a recovery function to complete workers by address or migrating from the old slot), but remain effectively frozen until then.

Severity

Impact Explanation: [Medium] Funds from pending unstake workers can be stranded until a corrective upgrade, and strategy operations (allocate/deallocate) can be significantly but temporarily unavailable. A viable workaround (follow-up upgrade or admin action) exists, so the impact is not high.

Likelihood Explanation: [Low] Exploitation requires a trusted admin/governance mistake to deploy an implementation that changes the AUSD STORAGE_LOCATION constant; no attacker-driven path.

Exploitation

Exploitation Scenarios:

Scenario 1.

A governance upgrade changes AusdStrategyStorageLib.STORAGE_LOCATION. After upgrade, existing async-unstake workers become invisible (empty activeUnstakeRequestIds and mapping). _finalizeMaturedUnstakes() finds none, _assertNoPendingShMonadUnstakes() passes, and terminatePositions proceeds without reclaiming pending MON, stranding funds until a corrective upgrade restores bookkeeping or allows completing workers by address.

Preconditions / Assumptions

- (a). Strategy has active async-unstake workers recorded in AUSD storage; some requests have matured per IShMon.getInternalEpoch().
- (b). Governance upgrades the strategy to an implementation that changes AusdStrategyStorageLib.STORAGE_LOCATION.

Scenario 2.

A governance upgrade changes the AUSD storage slot. basisGuardBps reads as zero in the new namespace; allocateFunds and deallocateFunds enforce a zero-threshold basis guard and revert on BasisOutOfRange until governance calls setBasisGuardBps to restore a nonzero threshold, temporarily DoS-ing strategy operations.

Preconditions / Assumptions

- (a). Governance upgrades the strategy to an implementation that changes AusdStrategyStorageLib.STORAGE_LOCATION.
- (b). Vault/operator invokes allocateFunds or deallocateFunds.
- (c). getBasisBps() > 0 at the time of the call.

Scenario 3.

A governance upgrade changes the AUSD storage slot. totalPendingUnstakeMon reads as zero and is excluded from _calculateTotalCollateralUsd(), causing getCurrentValue() to under-report NAV and potentially misinform operational decisions until repaired.

Preconditions / Assumptions

- (a). Governance upgrades the strategy to an implementation that changes AusdStrategyStorageLib.STORAGE_LOCATION.
- (b). totalPendingUnstakeMon was nonzero before the upgrade and is relied upon by reporting (getCurrentValue()).

Proposed fix

StrategyStorageLib.sol

File: `src/libraries/StrategyStorageLib.sol`

Source

```
 ... 69 unchanged lines ...
     // // keccak256(abi.encode(uint256(keccak256("haha.storage.AusdStrategy")) - 1)) &
     // // ~bytes32(uint256(0xff))
+    /// @notice Canonical ERC-7201 storage slot for AUSD strategy state.
+    /// @dev DO NOT CHANGE for existing deployments. Changing this constant will cause deployed
+    ///      proxies to read a fresh empty namespace and lose access to live async-unstake
+    ///      bookkeeping (basisGuardBps, activeUnstakeRequestIds, unstakeRequests). If a future
+    ///      migration is required, implement an explicit one-time migration path instead of
+    ///      altering this slot.
     bytes32 internal constant STORAGE_LOCATION = 0x7020ec6974f11c5329c9d1381dc3321d631ccdd5d8b99fd012cb7ea7afa1

     /// @notice Tracks an asynchronous shMON unstake bucket owned by the strategy.
     /// @dev Each bucket represents capital that has left the active shMON position but has
     ///      not yet returned as withdrawable MON.
     struct UnstakeRequest {
         /// @notice Dedicated worker that isolates the unstake bucket.
         address worker;
         /// @notice Expected MON amount the strategy uses for pending exposure accounting.
```

```
        uint256 expectedAmountMon;
        /// @notice Epoch after which the request can be finalized.
        uint64 completionEpoch;
        /// @notice Whether the request is still active and awaiting completion or forced removal.
        bool active;
    }

    /// @notice AUSD-specific state appended after the shared base strategy storage.
    /// @dev The active request id array is intentionally dense so the strategy can iterate
    ///      only over currently pending requests when reconciling exposure.
+   /// @custom:storage-location erc7201:haha.storage.AusdStrategy
    struct AusdStrategyStorage {
        /// @notice Maximum acceptable shMON basis deviation in basis points for guarded actions.
 ... 22 unchanged lines ...
```

[Medium] Unrepaired storage layout migration in BaseStrategy upgrades causes misdirected admin payouts

Description

Upgrading legacy-layout strategy proxies directly to the current BaseStrategy/AusdStrategy without the one-time storage-layout repair makes the implementation read strategyAdmin and borrow-cap thresholds from wrong ERC-7201 offsets. Privileged payout functions can then send funds to an unintended address or revert until repaired. Threshold fields are unused in current borrowing logic, so risk controls are unaffected.

The BaseStrategy ERC-7201 storage layout moved primaryBorrowCapThresholdBps, secondaryBorrowCapThresholdBps, and strategyAdmin to earlier offsets (10, 11, 12). Legacy proxies stored these at later offsets (15, 16, 17). The on-chain migration hook (reinitializeUpgrade/_migrateUpgrade) only rewires adapters and the WMON unwrapper; it does not repair these legacy offsets. After a direct UUPS upgrade of a legacy proxy without running the documented storage-layout repair, the current code reads the strategyAdmin from the wrong slot. As a result, BaseStrategy.withdrawToOwner (onlySelf) and BaseStrategy.rescueToken (onlyStrategyAdmin) transfer to an unintended address or revert (if the slot decodes to zero) until strategyAdmin is repaired via setStrategyAdmin by STRATEGY_ADMIN_ADMIN. Borrow-cap threshold fields, though mismatched, are not used by in-scope runtime borrowing logic, so they do not degrade risk controls. AccessControl roles persist across the upgrade, allowing governance to fix the storage after the implementation switch.

Severity

Impact Explanation: [High] WithdrawToOwner can move core principal to an unintended address if used post mis-upgrade, constituting a direct material loss of principal funds. RescueToken can misdirect non-core balances, causing loss of yield/fees.

Likelihood Explanation: [Low] Exploitation requires multiple trusted-operator missteps: a legacy proxy must be upgraded without the documented storage-layout repair, and privileged payout functions must be used before repairing strategyAdmin. There is no external attacker path due to onlySelf/role gating.

Exploitation

Exploitation Scenarios:

Scenario 1.

Operator mis-upgrades a legacy-layout strategy proxy without the storage-layout repair and then executes a batch containing withdrawToOwner for the core asset. The strategy reads a non-zero unintended strategyAdmin from the wrong slot and transfers the requested amount to that address, causing direct loss of principal.

Preconditions / Assumptions

- (a). A legacy-layout strategy proxy exists on mainnet
- (b). The proxy is upgraded directly to the current implementation without running the one-time storage-layout repair
- (c). The value at the new strategyAdmin slot (offset 12) decodes to a non-zero unintended address
- (d). Operator includes withdrawToOwner for the core asset in a self-called batch before repairing strategyAdmin

Scenario 2.

Operator mis-upgrades as above and, before repair, calls rescueToken to sweep non-core reward/dust tokens. The function transfers the tokens to the unintended strategyAdmin address read from the wrong slot, losing those balances.

Preconditions / Assumptions

- (a). A legacy-layout strategy proxy exists on mainnet
- (b). The proxy is upgraded directly to the current implementation without running the one-time storage-layout repair
- (c). The value at the new strategyAdmin slot (offset 12) decodes to a non-zero unintended address
- (d). The strategy holds non-core ERC20 balances (rewards/dust)
- (e). Caller holds STRATEGY_ADMIN role and invokes rescueToken before repairing strategyAdmin

Scenario 3.

Operator mis-upgrades as above and the wrong slot decodes to address(0). Attempts to use withdrawToOwner or rescueToken revert (InvalidParams), causing temporary unavailability of these admin tools until strategyAdmin is repaired.

Preconditions / Assumptions

- (a). A legacy-layout strategy proxy exists on mainnet
- (b). The proxy is upgraded directly to the current implementation without running the one-time storage-layout repair
- (c). The value at the new strategyAdmin slot (offset 12) decodes to address(0)
- (d). Operator attempts withdrawToOwner or rescueToken before repairing strategyAdmin

2. [Medium] withdrawToOwner batch exposure without core-asset restriction in BaseStrategy causes admin-directed exfiltration of vault/borrowed assets

Status

Review status: Unresolved
Remediation status: Unremediated
Remediation note: Created by pipeline analysis

Description

BaseStrategy exposes withdrawToOwner as a normal batch command callable via the strategy's self-call batch engine, with no restriction against the core vault asset. A vault-side allocator can compose batches (e.g., pullFromVault + withdrawToOwner or borrow + withdrawToOwner) to route funds held by the strategy directly to strategyAdmin, enabling privileged exfiltration of principal or borrowed assets.

The strategy command table maps the hashed command id for withdrawToOwner to an externally callable wrapper that enforces onlySelf; this is satisfied by the internal batch executor that self-calls wrappers for each command in a vault-submitted batch. The wrapper transfers any specified ERC20 to the stored strategyAdmin and does not restrict the token to non-core assets. The batch executor does not forbid withdrawToOwner in either allocate or deallocate flows and only blocks mixing pullFromVault with returnIdleToVault. Paused state can also be bypassed for these commands by setting the batch's allowWhenPaused flag. Unlike rescueToken, which blocks the core asset, withdrawToOwner imposes no such check. This is a privileged-path risk, not externally exploitable; severity hinges on whether the vault-side allocator and strategyAdmin are the same trusted entity or intentionally separated roles.

Severity

Impact Explanation: [High] The flows enable direct, material loss of principal funds from the vault/depositors and can create insolvency by draining borrowed assets while leaving debt behind.

Likelihood Explanation: [Low] Exploitation requires misuse or compromise of a trusted, privileged vault-side role to construct and submit malicious batches; there is no external attacker path.

Exploitation

Exploitation Scenarios:

Scenario 1.

Direct pull-and-drain: An allocateFunds batch first pulls core assets from the vault into the strategy (pullFromVault), then immediately calls withdrawToOwner to transfer those assets from the strategy to strategyAdmin, resulting in depositor

principal loss.

Preconditions / Assumptions

- (a). Strategy has a nonzero strategyAdmin address configured
- (b). Vault holds the core asset and has granted allowance to the strategy for pullFromVault
- (c). A privileged vault-side allocator/manager/governance constructs and submits the batch

Scenario 2.

Borrow-and-drain leaving debt: An allocateFunds batch deposits pulled core assets as collateral, borrows a volatile asset against that collateral, and immediately transfers the borrowed tokens to strategyAdmin via withdrawToOwner, leaving debt on the strategy collateralized by depositor funds.

Preconditions / Assumptions

- (a). Strategy has a nonzero strategyAdmin address configured
- (b). Adapters are wired and functional to deposit collateral and borrow
- (c). Vault holds the core asset and has granted allowance to the strategy
- (d). A privileged vault-side allocator/manager/governance constructs and submits the batch

Scenario 3.

Drain while paused via allowWhenPaused: Even if the strategy is paused, an allocateFunds batch with allowWhenPaused=true includes withdrawToOwner (optionally after pullFromVault) to move strategy-held assets to strategyAdmin during an emergency stop.

Preconditions / Assumptions

- (a). Strategy is paused
- (b). Strategy has a nonzero strategyAdmin address configured
- (c). A privileged vault-side allocator/manager/governance constructs and submits an allocateFunds batch with allowWhenPaused=true

Proposed fix

BaseStrategy.sol

File: `src/strategies/BaseStrategy.sol`

Source

```
 ... 620 unchanged lines ...
     /// @param data ABI-encoded `(address asset_, uint256 amount)`.
     /// @dev Batch payloads generally use this for dust recovery after explicit governance approval.
-    function withdrawToOwner(bytes memory data) external onlySelf {
+    function withdrawToOwner(bytes memory data) external onlyStrategyAdmin {
         (address asset_, uint256 amount) = abi.decode(data, (address, uint256));
+        if (asset_ == this.asset()) {
+            revert CannotRescueCoreAsset();
+        }
         address admin = _getBaseStrategyStorage().strategyAdmin;
         if (admin == address(0)) revert InvalidParams();
 ... 767 unchanged lines ...
```

3. [Medium] Immutable token/oracle wiring in UUPS-upgradeable AusdStrategy without compatibility checks causes DoS and enables admin exfiltration of idle core assets

Status

Review status: Unresolved
Remediation status: Unremediated
Remediation note: Created by pipeline analysis

Description

AusdStrategy stores core token/oracle wiring as implementation immutables while running behind a UUPS proxy. Upgrades replace these immutables without on-chain compatibility checks, so a misconfigured or malicious upgrade can silently repoint the strategy to different assets/oracles. This can brick allocation/unwind flows, degrade safety guards, misprice accounting, and allow a privileged admin to rescue previously core assets.

AusdStrategy is a UUPS-upgradeable strategy whose constructor sets critical references (STABLECOIN, SHMON, WMON, three oracles, their heartbeats, and STABLECOIN_DECIMALS) as implementation immutables. Because the proxy reads immutables from the current implementation's bytecode, upgrading to an implementation compiled with different constructor arguments silently changes these values for the live proxy. There is no upgrade-time invariant check in BaseStrategy._authorizeUpgrade or reinitializeUpgrade to assert immutables remain consistent. The strategy's runtime relies on these immutables for vault denomination (asset()), pulling assets from the vault (_pullFromVault), conversions and unwrap/stake flows (WMON/SHMON), valuation and LTV/basis-guard checks (OracleHelpers with the configured oracles/heartbeats), and helper deployment (WMonUnwrapper bound to WMON). A misconfigured upgrade can therefore (1) brick allocate/deallocate by pointing STABLECOIN to a different token than the vault's asset, (2) bypass or degrade the basis-guard on deallocation or return incorrect NAV/LTV due to wrong oracles/heartbeats/decimals, and (3) enable a privileged admin to exfiltrate idle balances of the original core asset by first changing asset() (via STABLECOIN drift) and then calling rescueToken on the old token. While many miswirings fail safely (revert/no-op) and repair hooks exist, the lack of on-chain compatibility assertions creates a real operational safety gap with potential for material user harm under a malicious or compromised admin.

Severity

Impact Explanation: [High] In the worst case (Scenario 3), a malicious or compromised admin can directly exfiltrate principal funds (idle core asset), which qualifies as a high impact.

Likelihood Explanation: [Low] All impactful scenarios require a trusted admin to misconfigure or maliciously perform a UUPS upgrade; such trusted role misuse/compromise is low likelihood per rules.

Exploitation

Exploitation Scenarios:

Scenario 1.

DoS from STABLECOIN mismatch: Strategy admin upgrades to a new implementation with STABLECOIN set to a different token than the vault's asset. asset() now returns the wrong token and _pullFromVault attempts to transfer that token from the vault, reverting due to missing balances/approvals. Allocation/deallocation flows brick until a corrective upgrade.

Preconditions / Assumptions

- (a). Vault's asset is Stablecoin X and the strategy previously used STABLECOIN = X
- (b). Trusted strategy admin performs an upgrade to an implementation with STABLECOIN = Y != X
- (c). Vault attempts allocateFunds or deallocateFunds using the strategy

Scenario 2.

Degraded risk guard and valuation drift: Strategy admin upgrades with incorrect oracle addresses/heartbeats/decimals. OracleHelpers calls in getBasisBps/_calculateTotalValue revert or produce mismatched outputs. allocateFunds reverts; deallocateFunds bypasses the basis guard (emitting BasisGuardBypassedOnDeallocate) and proceeds without that protection, while NAV/LTV reporting becomes unavailable or inaccurate.

Preconditions / Assumptions

- (a). Trusted strategy admin performs an upgrade to an implementation with incorrect/mismatched oracle addresses, heartbeats, or STABLECOIN_DECIMALS
- (b). Vault triggers allocateFunds or deallocateFunds, or systems read NAV/LTV via getCurrentValue/getPositionState

Scenario 3.

Admin exfiltration of idle core asset via asset() drift and rescueToken: A malicious or compromised strategy admin upgrades to an implementation with STABLECOIN changed to another token. asset() now points to the new token, so

rescueToken(oldCoreAsset, amount) is permitted. Any idle balance of the original stablecoin held by the strategy can be transferred to the admin.

Preconditions / Assumptions

- (a). Strategy holds a material idle balance of the original stablecoin X
- (b). Trusted strategy admin is malicious or compromised and upgrades to an implementation with STABLECOIN changed to Y != X
- (c). Admin calls rescueToken(X, amount) to transfer idle core-asset balance to themselves

4. [Medium] Unvalidated decimals/heartbeat in Euler/Curvance adapters causes incorrect LTV sizing and potential liquidation

Status

Review status: Unresolved
Remediation status: Unremediated
Remediation note: Created by pipeline analysis

Description

Adapters store constructor-supplied token decimals and oracle heartbeat values without validation and pass them into LTV/Oracle math for max-borrow and safe-withdraw sizing. Misconfiguration can inflate or deflate headroom or accept stale prices, causing policy-violating leverage or repeated execution failures. Underlying protocols still enforce health checks, but elevated liquidation risk or DoS-like friction can result.

Euler and Curvance adapters persist DEBT_DECIMALS/COLLATERAL_DECIMALS and oracle heartbeats directly from constructor inputs, with no on-chain validation or derivation from underlying tokens/feeds. These values are used by LTVCalculations and OracleHelpers to compute USD-normalized collateral/debt and enforce oracle freshness for getMaxBorrowAmount and getMaxSafeWithdrawal. The BaseStrategy uses these adapter "max" helpers when amount == 0, so misconfiguration can lead to: (1) inflated borrow amounts that push realized LTV above intended policy targets (though still within protocol limits), increasing liquidation sensitivity; or (2) repeated reverts on "max" paths (operational friction). CompositeOracle bounds heartbeats, and AusdStrategy derives token decimals from contracts, but adapters do not. The overall risk is a configuration-hardening gap that can manifest as elevated liquidation risk or execution failures.

Severity

Impact Explanation: [High] Mis-sized leverage can result in liquidation and penalties (principal loss) when market conditions move adversely; other scenarios cause operational friction but no asset loss.

Likelihood Explanation: [Low] All scenarios require trusted-role misconfiguration (constructor-supplied decimals/heartbeat) and, for the leverage case, specific operator choices (amount==0, non-binding maxDebt) and subsequent market movement.

Exploitation

Exploitation Scenarios:

Scenario 1.

Misconfigured adapter decimals inflate max-borrow: operators call borrow with amount==0 at a target LTV; the adapter, using wrong token decimals, undercounts debt USD or overstates collateral USD, returning an inflated borrowable amount. The borrow succeeds within protocol limits but leaves the position nearer liquidation than intended; a normal adverse market move later triggers liquidation and penalties.

Preconditions / Assumptions

- (a). Adapter deployed with incorrect token decimals (e.g., debt decimals larger than actual or collateral decimals smaller than actual)
- (b). Operators use amount==0 to request adapter-computed max borrow and set maxDebt high enough not to bind
- (c). Underlying protocol health/liquidity still allows the computed borrow (below hard limits)
- (d). Oracle data is fresh enough to pass heartbeat checks
- (e). Subsequent normal adverse market move occurs

Scenario 2.

Heartbeat too small causes StaleOraclePrice reverts: an adapter is deployed with an unrealistically small (or zero) heartbeat; any amount==0 max-borrow or max-withdraw computation reverts in OracleHelpers.getPrice as stale, causing batch failures until operators use explicit amounts or fix configuration.

Preconditions / Assumptions

- (a). Adapter deployed with very small or zero oracle heartbeat
- (b). Oracle feed updates less frequently than configured heartbeat
- (c). Operators rely on amount==0 max sizing for borrow/withdraw

Scenario 3.

Excessively large decimals trigger overflow reverts: an adapter is deployed with extremely large tokenDecimals (e.g., ≥78); OracleHelpers uses 10**tokenDecimals which overflows and reverts on any LTV-based max computation, causing all amount==0 sizing paths to fail until configuration is corrected or explicit amounts are provided.

Preconditions / Assumptions

- (a). Adapter deployed with extremely large tokenDecimals (e.g., ≥78)
- (b). Operators rely on amount==0 max sizing which triggers LTV/Oracle conversions

Proposed fix

EulerDebtAdapter.sol

File: `src/adapters/euler/EulerDebtAdapter.sol`

Source

```
 ... 77 unchanged lines ...
         COLLATERAL_ORACLE = AggregatorV3Interface(collateralOracle_);
         DEBT_DECIMALS = debtDecimals_;
+        // SECURITY: Derive decimals from IEVault(asset()) via IERC20Metadata.decimals() and require match; bou
+        // and ensure decimals <= 77 to avoid 10**n overflow in OracleHelpers.
         COLLATERAL_DECIMALS = collateralDecimals_;
         DEBT_ORACLE_HEARTBEAT = debtOracleHeartbeat_;
 ... 121 unchanged lines ...
```

CurvanceDebtAdapter.sol

File: `src/adapters/curvance/CurvanceDebtAdapter.sol`

Source

```
 ... 83 unchanged lines ...
         DEBT_ORACLE = AggregatorV3Interface(debtOracle_);
         COLLATERAL_ORACLE = AggregatorV3Interface(collateralOracle_);
+        // SECURITY: Derive decimals from ICToken/IBorrowableCToken .asset() via IERC20Metadata.decimals() and
+        // and ensure decimals <= 77 to avoid 10**n overflow in OracleHelpers.
         DEBT_DECIMALS = debtDecimals_;
         COLLATERAL_DECIMALS = collateralDecimals_;
 ... 222 unchanged lines ...
```

CurvanceCollateralAdapter.sol

File: `src/adapters/curvance/CurvanceCollateralAdapter.sol`

Source

```
 ... 90 unchanged lines ...
         COLLATERAL_ORACLE = AggregatorV3Interface(collateralOracle_);
```

```
         DEBT_ORACLE = AggregatorV3Interface(debtOracle_);
+        // SECURITY: Derive decimals from ICToken/IBorrowableCToken .asset() via IERC20Metadata.decimals() and
+        // and ensure decimals <= 77 to avoid 10**n overflow in OracleHelpers.
         COLLATERAL_DECIMALS = collateralDecimals_;
         DEBT_DECIMALS = debtDecimals_;
 ... 149 unchanged lines ...
```

EulerCollateralAdapter.sol

File: `src/adapters/euler/EulerCollateralAdapter.sol`

Source

```
 ... 102 unchanged lines ...
         COLLATERAL_ORACLE = AggregatorV3Interface(collateralOracle_);
         DEBT_ORACLE = AggregatorV3Interface(debtOracle_);
+        // SECURITY: Derive decimals from IEVault(asset()) via IERC20Metadata.decimals() and require match; bou
+        // and ensure decimals <= 77 to avoid 10**n overflow in OracleHelpers.
         COLLATERAL_DECIMALS = collateralDecimals_;
         DEBT_DECIMALS = debtDecimals_;
 ... 140 unchanged lines ...
```

Related findings

[Medium] Unchecked adapter constructor parameters in protocol adapters cause approval-based token drain and strategy liveness failures

Description

All four protocol adapters store critical wiring (protocol/oracle addresses, decimals, heartbeats) without validation. Misconfiguration can enable a malicious debt token to drain funds via unlimited approval during repay-all, or cause persistent reverts that block allocation/unwind flows. Although recoverable via adapter migration, the missing checks create a high-impact misconfiguration footgun with low likelihood.

The EulerCollateralAdapter, EulerDebtAdapter, CurvanceCollateralAdapter, and CurvanceDebtAdapter constructors assign addresses (protocol controllers/vaults/markets/oracles), token decimals, and oracle heartbeats without any validation (non-zero, isContract, heartbeat > 0, decimals bounds). BaseStrategy delegates state-changing calls to these adapters and relies on their view functions for auto-sizing and cooldown checks. Miswired addresses (e.g., Curvance MARKET_MANAGER) cause deallocation to revert in BaseStrategy._verifyCooldowns. In CurvanceDebtAdapter.repay, repay(amount == 0) sets unlimited ERC20 approval to DEBT_TOKEN and then calls repay; if DEBT_TOKEN is miswired to a malicious address, it can immediately drain the strategy's WMON via transferFrom. Heartbeat == 0 makes OracleHelpers.getPrice revert as stale for nearly all rounds, bricking adapter auto-sizing (getMaxBorrowAmount/getMaxSafeWithdrawal). Oversized decimals (>77) can overflow 10**decimals in price conversions, also causing reverts. While governance can migrate adapters to recover, missing constructor validation exposes the strategy to severe loss or operational DoS when misconfiguration occurs.

Severity

Impact Explanation: [High] Scenario 1 enables direct, material loss of principal funds via unlimited approval to a malicious debt token; Scenarios 2 and 3 cause significant availability loss of core allocation/unwind functionality.

Likelihood Explanation: [Low] All scenarios require trusted role (governance/admin) misconfiguration or compromise of adapter parameters; end users do not control these settings.

Exploitation

Exploitation Scenarios:

Scenario 1.

CurvanceDebtAdapter repay-all drains WMON: DEBT_TOKEN is miswired to an attacker contract; when the strategy calls repay(WMON, 0), the adapter grants unlimited allowance to DEBT_TOKEN and calls repay, after which the attacker uses

transferFrom to drain the strategy’s WMON.

Preconditions / Assumptions

- (a). CurvanceDebtAdapter’s DEBT_TOKEN address is misconfigured to an attacker-controlled contract
- (b). Strategy holds a non-zero WMON balance
- (c). A repay-all flow is triggered (repay with amount == 0) during unwind/termination

Scenario 2.

CurvanceCollateralAdapter deallocation DoS: MARKET_MANAGER is miswired (zero/EOA/wrong contract); BaseStrategy.deallocateFunds calls verifyCooldownPassed, which reverts when the adapter calls MARKET_MANAGER.accountAssets, blocking all deallocation/unwind steps.

Preconditions / Assumptions

- (a). CurvanceCollateralAdapter’s MARKET_MANAGER address is misconfigured (zero/EOA/wrong contract)
- (b). Vault invokes deallocateFunds (unwind) so BaseStrategy._verifyCooldowns runs

Scenario 3.

Heartbeat set to 0 bricks auto-sizing: An adapter is deployed with collateral/debt oracle heartbeat == 0; adapter view math (getMaxBorrowAmount/getMaxSafeWithdrawal) calls OracleHelpers.getPrice which reverts as stale, causing allocation/unwind commands that rely on amount == 0 auto-sizing to revert.

Preconditions / Assumptions

- (a). An Euler or Curvance adapter is deployed with an oracle heartbeat set to 0
- (b). Vault submits commands relying on adapter auto-sizing (amount == 0 for borrow/withdraw)

Proposed fix

CurvanceDebtAdapter.sol

File: `src/adapters/curvance/CurvanceDebtAdapter.sol`

Source

```
 ... 87 unchanged lines ...
         DEBT_ORACLE_HEARTBEAT = debtOracleHeartbeat_;
         COLLATERAL_ORACLE_HEARTBEAT = collateralOracleHeartbeat_;
+        require(ICurvanceMarketManager(marketManager_).isListed(debtToken_));
     }

 ... 218 unchanged lines ...
```

OracleHelpers.sol

File: `src/libraries/OracleHelpers.sol`

Source

```
 ... 125 unchanged lines ...
             // Casting to `uint256` is safe because negative values were already rejected above.
             // forge-lint: disable-next-line(unsafe-typecast)
-            return (uint256(priceInt), oracle.decimals());
+            uint8 dec = oracle.decimals();
+            if (dec > 77) revert InvalidOraclePrice();
+            return (uint256(priceInt), dec);
         } catch {
             // Surface a deterministic protocol error instead of the raw aggregator revert payload.
             revert OracleCallReverted();
         }
```

```
    }
}
```

BaseStrategy.sol

File: `src/strategies/BaseStrategy.sol`

Source

```
... 945 unchanged lines ...
            revert InvalidParams();
        }
+       if (!Address.isContract(primaryCollateralAdapter) || !Address.isContract(primaryDebtAdapter) || !Addres

        BaseStrategyStorageLib.BaseStrategyStorage storage $ = _getBaseStrategyStorage();
... 444 unchanged lines ...
```

[Medium] Missing adapter address validation in BaseStrategy.__BaseStrategy_init causes potential fund theft via delegatecall and operational DoS

Description

BaseStrategy.__BaseStrategy_init does not validate the four adapter addresses. Later, the strategy uses these addresses via delegatecall and functionStaticCall, enabling arbitrary code execution if malicious adapters are set, and causing misleading cooldown checks and DoS if zero/EOA addresses are set.

The strategy initializer stores primary/secondary collateral and debt adapters without checking for non-zero or contract code. Subsequent execution paths use these addresses with functionDelegateCall for mutating operations (deposit/withdraw/borrow/repay) and functionStaticCall for queries. If an adapter is a malicious contract, delegatecall executes arbitrary code in the strategy's context, allowing token exfiltration and storage corruption. If an adapter is zero/EOA, direct external cooldown calls may silently pass while any meaningful adapter interaction reverts, causing DoS. The migration helper only rejects zero addresses and still lacks isContract checks. Cooldown verification uses direct calls (not Address.functionStaticCall), permitting silent success to zero/EOA. Together, this creates risks of fund theft (with malicious adapters) and operational failures (with zero/EOA).

Severity

Impact Explanation: [High] Scenarios include direct, material loss of principal funds via delegatecall into malicious adapters and significant availability loss/DoS of core lifecycle operations due to zero/EOA misconfiguration.

Likelihood Explanation: [Low] Exploitation relies on rare/exceptional states and trusted-role/operator mistakes: an uninitialized-proxy race and subsequent vault adoption (Scenario 1), admin misconfiguration (Scenario 2), or malicious/compromised admin during migration (Scenario 3).

Exploitation

Exploitation Scenarios:

Scenario 1.

Uninitialized proxy race: An attacker front-runs initialization of a deployed strategy proxy, setting all adapters to a malicious contract with a fallback that transfers strategy-held tokens and returns empty data. The vault later integrates the proxy, approves the strategy, and calls allocateFunds (pullFromVault then depositCollateral). During depositCollateral, delegatecall enters the malicious adapter fallback, which drains the just-pulled assets from the strategy. The call appears successful due to empty return data.

Preconditions / Assumptions

- (a). A strategy proxy is deployed and remains uninitialized long enough for an attacker to call initialize first.
- (b). The attacker sets all four adapters to a malicious contract that returns empty data in fallback.
- (c). The vault subsequently adopts this proxy, grants the strategy allowance for the vault asset, and runs allocateFunds that includes pullFromVault and an adapter-mutating command.

Scenario 2.

Zero/EOA adapter misconfiguration: The strategy is initialized with one or more adapter addresses set to address(0) or an EOA. canWithdrawNow() passes because cooldown checks are direct external calls that silently succeed on zero/EOA. However, deallocateFunds or other adapter operations subsequently revert when functionStaticCall/functionDelegateCall detect non-contract targets, causing significant but temporary DoS of lifecycle operations until migration fixes the configuration.

Preconditions / Assumptions

- (a). Initializer was called with one or more adapter addresses set to address(0) or an EOA.
- (b). The vault adopts and uses this strategy for deallocation or other adapter-dependent operations.

Scenario 3.

Malicious adapter rewiring via upgrade: A privileged admin (or compromised key) invokes reinitializeUpgrade with a payload that rewires adapters to attacker-controlled contracts (non-zero). Later, normal vault operations (e.g., depositCollateral/borrow) delegatecall into the malicious adapters, executing arbitrary code that transfers strategy-held tokens to the attacker.

Preconditions / Assumptions

- (a). A privileged strategy admin (or compromised admin key) executes a migration to replace adapters with attacker-controlled contracts (non-zero).
- (b). The vault triggers normal strategy operations that invoke adapter delegatecalls.

Proposed fix

BaseStrategy.sol

File: `src/strategies/BaseStrategy.sol`

Source

```
 ... 164 unchanged lines ...
     {
         if (owner_ == address(0) || vault_ == address(0)) revert InvalidParams();
+        if (
+            primaryCollateralAdapter_ == address(0) || primaryDebtAdapter_ == address(0)
+                || secondaryCollateralAdapter_ == address(0) || secondaryDebtAdapter_ == address(0)
+                || !Address.isContract(primaryCollateralAdapter_) || !Address.isContract(primaryDebtAdapter_)
+                || !Address.isContract(secondaryCollateralAdapter_) || !Address.isContract(secondaryDebtAdapter
+        ) {
+            revert InvalidParams();
+        }
         if (maxPrimaryLtv_ > LTVCalculations.BPS || maxSecondaryLtv_ > LTVCalculations.BPS) {
             revert InvalidMaxLtv(maxPrimaryLtv_, maxSecondaryLtv_);
 ... 777 unchanged lines ...
             revert InvalidParams();
         }
+        if (
+            !Address.isContract(primaryCollateralAdapter) || !Address.isContract(primaryDebtAdapter)
+                || !Address.isContract(secondaryCollateralAdapter) || !Address.isContract(secondaryDebtAdapter)
+        ) {
+            revert InvalidParams();
+        }

         BaseStrategyStorageLib.BaseStrategyStorage storage $ = _getBaseStrategyStorage();
 ... 86 unchanged lines ...
     function _verifyCooldowns() internal view virtual {
         BaseStrategyStorageLib.BaseStrategyStorage storage $ = _getBaseStrategyStorage();
-        $.primaryCollateralAdapter.verifyCooldownPassed(address(this));
-        $.secondaryCollateralAdapter.verifyCooldownPassed(address(this));
+        address($.primaryCollateralAdapter).functionStaticCall(
```

```
+            abi.encodeWithSelector(ICollateralAdapter.verifyCooldownPassed.selector, address(this))
+        );
+        address($.secondaryCollateralAdapter).functionStaticCall(
+            abi.encodeWithSelector(ICollateralAdapter.verifyCooldownPassed.selector, address(this))
+        );
     }

 ... 352 unchanged lines ...
```

5. [Medium] Unenforced LTV ceilings in BaseStrategy allow over-leverage beyond governance limits causing potential liquidation losses

Status

Review status: Unresolved
Remediation status: Unremediated
Remediation note: Created by pipeline analysis

Description

BaseStrategy stores per-leg max LTV ceilings but never enforces them. Borrow/withdraw helpers pass caller-supplied targetLtv directly to adapters when auto-sizing (amount == 0) and bypass sizing entirely when explicit amounts are used (amount > 0). No pre- or post-action checks keep LTV within configured maxima, enabling batches to exceed governance risk limits or cause avoidable reverts.

The strategy initializes and exposes MAX_PRIMARY_LTV and MAX_SECONDARY_LTV but never applies them during execution. In _borrow and _withdrawCollateral, when amount == 0 the provided targetLtv is forwarded unchanged into adapter sizing (getMaxBorrowAmount/getMaxSafeWithdrawal). When amount > 0, the strategy directly delegatecalls adapter.borrow/withdraw without any LTV checks. Adapters compute using the provided targetLtv and protocol constraints and do not read the strategy's stored maxima (example usage in CurvanceDebtAdapter). There is no post-action verification that resulting LTVs are within the configured ceilings. Because only the vault can call, this is an operator/automation misuse surface: batches can over-leverage above governance limits (increasing liquidation risk) or issue explicit amounts that cause protocol-side reverts during allocate/deallocate flows.

Severity

Impact Explanation: [High] Exceeding governance-configured LTV ceilings can result in direct, material loss of principal through liquidation if adverse price moves occur while at higher-than-intended leverage.

Likelihood Explanation: [Low] Exploitation requires trusted role/operator misuse or compromise (only the vault can call) and, for principal loss, adverse market movement; per rules, trusted role misuse maps to low likelihood.

Exploitation

Exploitation Scenarios:

Scenario 1.

Over-leveraging via auto-sizing: The vault submits a batch with borrow(PRIMARY, WMON, targetLtv=8500 bps, amount=0). BaseStrategy forwards 8500 bps to the adapter, which sizes the borrow at ~85% LTV if allowed. The resulting position exceeds the strategy's configured governance LTV ceiling (e.g., 70%), leaving the strategy with reduced buffer and higher liquidation risk.

Preconditions / Assumptions

- (a). Strategy configured with conservative MAX_PRIMARY_LTV/MAX_SECONDARY_LTV (e.g., 7000 bps)
- (b). Underlying protocol permits higher LTV than the configured maxima under current oracles (e.g., 8500 bps)
- (c). Vault/operator composes batch with amount == 0 and elevated targetLtv
- (d). Strategy not paused

Scenario 2.

Over-withdrawing to a too-high target LTV during deallocation: The vault submits withdrawCollateral(PRIMARY, stablecoin, targetLtv=9000 bps, amount=0) after cooldowns. BaseStrategy forwards 9000 bps to the adapter, which sizes the withdrawal to leave the position near 90% LTV if allowed. The position ends riskier than intended by governance, vulnerable to small price moves and liquidation.

Preconditions / Assumptions

- (a). Active position with both collateral and debt; cooldowns passed
- (b). Underlying protocol allows withdrawal to the higher target LTV under current oracles
- (c). Vault/operator composes batch with amount == 0 and elevated targetLtv
- (d). Strategy not paused

Scenario 3.

Explicit-amount bypass causes avoidable reverts: The vault submits borrow/withdraw with amount > 0 that violates protocol constraints (e.g., Curvance MIN_LOAN_SIZE or health). BaseStrategy delegates the call without pre-checks, causing adapter/protocol reverts and operational delays in allocation/deallocation.

Preconditions / Assumptions

- (a). Vault/operator composes borrow/withdraw with explicit amount > 0 that conflicts with protocol constraints or safe sizing
- (b). Underlying protocol enforces constraints and reverts on invalid amounts
- (c). Trusted role/operator error or misuse

Proposed fix

BaseStrategy.sol

File: `src/strategies/BaseStrategy.sol`

Source

```
 ... 682 unchanged lines ...

     {
+        // SECURITY: Enforce per-leg MAX_*_LTV ceilings.
+        // - For amount == 0: call getMaxSafeWithdrawal with min(targetLtv, per-leg ceiling).
+        // - For amount > 0: compute allowed via getMaxSafeWithdrawal at the per-leg ceiling and revert if requ
+        //   amount (after applying reserveAmount cap for vault asset) exceeds allowed.
+        //   This prevents withdrawing to an LTV above governance-configured limits.
         if (amount == 0) {
             // Calculate safe withdrawal amount using staticcall
 ... 42 unchanged lines ...
         internal
     {
+        // SECURITY: Enforce per-leg MAX_*_LTV ceilings.
+        // - For amount == 0: call getMaxBorrowAmount with min(targetLtv, per-leg ceiling).
+        // - For amount > 0: compute allowed via getMaxBorrowAmount at the per-leg ceiling and revert if reques
+        //   amount exceeds allowed.
+        //   This prevents borrowing beyond governance-configured risk limits.
         IDebtAdapter adapter = _getDebtAdapter(position);
         if (amount == 0) {
 ... 661 unchanged lines ...
```

6. [Medium] Min-out enforcement trusts redeem() return value in SwapAdapter.unstakeShMonToMon, allowing under-delivery of MON and NAV loss

Status

Review status: Unresolved
Remediation status: Unremediated
Remediation note: Created by pipeline analysis

Description

The immediate shMON→MON redeem path enforces minMonOut against IERC4626.redeem's return value rather than the actual native MON received. A misbehaving or buggy shMON can report higher redeemed assets than it transfers, causing the min-out check to pass while the strategy receives less MON, undermining slippage protection and potentially causing losses or unwind failures.

In SwapAdapter.unstakeShMonToMon, the helper calls shMon.redeem(sharesToRedeem, address(this), address(this)) and enforces minMonOut using the returned value, without measuring the native MON balance delta. Other conversions in the codebase use delta-based checks, but this path does not. AusdStrategy._unstakeShMonToMon computes an enforcedMinOut and delegates to this helper, expecting min-out protection. If the external shMON misreports the redeemed amount or under-delivers MON, the check passes yet the strategy actually receives less. Downstream, _swapShMonToWMon wraps whatever MON arrived; it only checks for non-zero output and does not reconcile against min-out, so the shortfall is realized. During termination, minOut is set to zero (so the criticized guard is moot), but under-delivery still reduces available WMON and can cause InsufficientPrimaryDebtCoverage reverts when repaying primary debt.

Severity

Impact Explanation: [High] If shMON misreports and burns shares while transferring less MON, this causes direct, material loss of principal funds. In termination, under-delivery can also cause significant operational DoS of unwinds, but the primary impact is potential principal loss.

Likelihood Explanation: [Low] Exploitation requires the external shMON integration to misbehave (maliciously or via a bug) by misreporting redeem amounts or under-delivering assets.

Exploitation

Exploitation Scenarios:

Scenario 1.

Adversarial shMON misreports redeem: redeem returns an inflated amount (≥ minOut) while transferring less MON and burning the full shares. The min-out check passes, the strategy receives less MON than enforced, wraps it into WMON, and suffers direct NAV loss.

Preconditions / Assumptions

- (a). External shMON behaves maliciously or incorrectly by returning an inflated redeem value while transferring less MON
- (b). Strategy holds idle shMON shares
- (c). Vault/operator executes a convert(SHMON→WMON) with non-zero minConversionOut

Scenario 2.

Non-malicious bug or fee/rounding mismatch in shMON: redeem returns a higher value than actually transferred. Each conversion passes min-out but under-delivers MON, leading to repeated small losses that accumulate and erode NAV over time.

Preconditions / Assumptions

- (a). External shMON has a bug or fee/rounding mismatch causing redeem to return more than actually transferred
- (b). Strategy regularly performs immediate shMON→WMON conversions with non-zero minConversionOut

Scenario 3.

Termination unwind failure: termination uses minOut=0 for redeem; under-delivery of MON leads to insufficient WMON to repay primary debt. The strategy reverts with InsufficientPrimaryDebtCoverage, causing operational DoS of the unwind step.

Preconditions / Assumptions

- (a). External shMON under-delivers MON during redeem
- (b). Strategy is terminating positions with _swapShMonToWMon(shMonBalance, 0)

- (c). Primary WMON debt must be repaid; insufficient WMON after wrapping causes a revert

Proposed fix

SwapAdapter.sol

File: `src/common/SwapAdapter.sol`

Source

```
 ... 91 unchanged lines ...
         // Redeem directly into native MON and validate the received amount against the caller's
         // minimum-out guard.
-        uint256 monReceived = shMon.redeem(sharesToRedeem, address(this), address(this));
+        uint256 monBefore = address(this).balance;
+        shMon.redeem(sharesToRedeem, address(this), address(this));
+        uint256 monReceived = address(this).balance - monBefore;
         if (monReceived == 0) revert ZeroMonOutput();
         if (monReceived < minMonOut) revert InsufficientMonOutput(monReceived, minMonOut);
 ... 34 unchanged lines ...
```

Related findings

[Medium] Trusting unwrapper return value in WMON unwrap helpers causes silent loss of principal during certain conversions

Description

WMON unwrap helpers trust only the external unwrapper's returned amount and do not verify the strategy's native-balance delta. If the strategy admin repoints the unwrapper to a faulty/malicious contract that returns a positive value without forwarding MON, the unwrap step "succeeds" while the strategy has already transferred away its WMON and received no MON. In AusdStrategy's WMON->shMON conversion, the subsequent staking step silently no-ops and minOut is not enforced, resulting in value loss. Default unwrapper is safe; many Uniswap v4 swap paths would revert and roll back, but unwrap-only/convert paths can silently lose funds under an admin-installed bad unwrapper.

SwapAdapter.unwrapWMon, ShMonSwapAdapter._unwrapWMon, and Swapper._unwrapWMon all transfer WMON to an external unwrapper and then trust the unwrapper's returned monReceived without verifying the strategy's native-balance delta. BaseStrategy.setWMonUnwrapper allows the strategy admin to set any unwrapper address. The default WMonUnwrapper is safe (it measures its own MON balance delta and reverts on forward failure), but a malicious or faulty unwrapper can return a positive value without sending MON back. In AusdStrategy, the WMON->shMON convert path unwraps and then calls stakeMonToShMon with amount=0 (use all idle MON). If no MON was actually forwarded, stakeMonToShMon early-returns on zero input and does not enforce minOut, leaving the strategy with lost WMON and no revert. By contrast, Uniswap v4 swap flows that must settle native MON will revert on missing funds and roll back the earlier WMON transfer. Thus, the risk is a silent loss in unwrap-only or convert flows not immediately followed by a native settle, contingent on admin misconfiguration or malice.

Severity

Impact Explanation: [High] Direct, material loss of principal funds held by the strategy (WMON is transferred out and not recovered), reducing NAV for vault depositors.

Likelihood Explanation: [Low] Requires trusted role (strategy admin) misuse/compromise or severe misconfiguration, and a batch path that unwraps without immediately settling native MON. The default unwrapper is safe, and many swap flows would revert and roll back on missing MON.

Exploitation

Exploitation Scenarios:

Scenario 1.

Silent WMON loss in AusdStrategy WMON->shMON convert: Admin repoints the unwrapper to a malicious contract. Vault runs a batch that converts WMON to shMON. SwapAdapter.unwrapWMon transfers WMON to the malicious unwrapper and trusts a positive return value; no MON is actually forwarded. Then stakeMonToShMon(0, minOut) sees zero MON, early-returns, and minOut is not enforced. The batch succeeds while WMON is lost and no shMON is minted.

Preconditions / Assumptions

- (a). Strategy holds a positive WMON balance
- (b). Strategy admin sets the unwrapper to a malicious/faulty contract via setWMonUnwrapper
- (c). Vault executes a batch that converts WMON to shMON (uses unwrap then stake)
- (d). No immediate native settle step (e.g., Uniswap v4 settle) occurs after unwrap to force a revert

Scenario 2.

Silent WMON loss via explicit unwrapWMon command: Admin repoints the unwrapper to a malicious contract. Vault executes a batch that includes the granular unwrapWMon command without an immediate step that requires spending native MON. SwapAdapter.unwrapWMon transfers WMON to the malicious unwrapper, trusts a positive return, and the batch completes successfully with no MON forwarded and WMON lost.

Preconditions / Assumptions

- (a). Strategy holds a positive WMON balance
- (b). Strategy admin sets the unwrapper to a malicious/faulty contract via setWMonUnwrapper
- (c). Vault executes a batch that includes unwrapWMon without an immediate native settle requirement

Proposed fix

SwapAdapter.sol

File: `src/common/SwapAdapter.sol`

Source

```
 ... 123 unchanged lines ...

         // Measure received MON by native balance delta instead of trusting wrapper internals.
+        uint256 monBefore = address(this).balance;
         IERC20(address(wmon)).safeTransfer(wmonUnwrapper, wmonAmount);
-        uint256 monReceived = IWMonUnwrapper(wmonUnwrapper).unwrapToStrategy(wmonAmount);
-        if (monReceived == 0) revert ZeroMonOutput();
+        IWMonUnwrapper(wmonUnwrapper).unwrapToStrategy(wmonAmount);
+        if (address(this).balance - monBefore == 0) revert ZeroMonOutput();
     }
 }
```

ShMonSwapAdapter.sol

File: `src/common/ShMonSwapAdapter.sol`

Source

```
 ... 298 unchanged lines ...
         if (amount == 0) return;
         if (wmonUnwrapper == address(0)) revert WMonUnwrapperNotConfigured();
+        uint256 monBefore = address(this).balance;
         IERC20(wmon).safeTransfer(wmonUnwrapper, amount);
-        uint256 monReceived = IWMonUnwrapper(wmonUnwrapper).unwrapToStrategy(amount);
-        if (monReceived == 0) revert ZeroSwapOutput(wmon, NATIVE_MON);
+        IWMonUnwrapper(wmonUnwrapper).unwrapToStrategy(amount);
+        if (address(this).balance - monBefore == 0) revert ZeroSwapOutput(wmon, NATIVE_MON);
     }

     function _wrapMon(address wmon, uint256 amount) private {
         if (amount == 0) return;
         IWrappedToken(wmon).deposit{ value: amount }();
     }

     function _amount0(IUniswapV4.BalanceDelta delta) private pure returns (int128) {
         return int128(int256(IUniswapV4.BalanceDelta.unwrap(delta)) >> 128);
```

```
    }

    function _amount1(IUniswapV4.BalanceDelta delta) private pure returns (int128) {
        int256 v = IUniswapV4.BalanceDelta.unwrap(delta);
        return int128((v << 128) >> 128);
    }
}
```

7. [Medium] Lack of implementation binding in GatedVaultImpl timelock causes unexpected/malicious code installation

Status

Review status: Unresolved
Remediation status: Unremediated
Remediation note: Created by pipeline analysis

Description

GatedVaultImpl’s upgrade timelock commits only to version, calldata hash, and timing, not to the exact implementation address/bytecode. Because the ConcreteFactory is upgradeable and selects the implementation at execution, governance can change which code is installed after scheduling. This weakens the notice guarantee and can result in installing different or malicious code, including stripping timelock protections for subsequent upgrades.

GatedVaultImpl.scheduleUpgrade stores pendingUpgradeVersion, pendingUpgradeDataHash, and timing, but does not persist the target implementation address or code hash. The event includes the implementation resolved at scheduling time for visibility, but pendingUpgradeImplementation() re-reads the factory mapping dynamically. At execution, GatedVaultImpl._upgrade enforces only that timelock is enabled, a schedule exists, the requested newVersion matches the pending version, the calldata hash matches the pending hash, and the executable time has passed. It does not verify that the installed implementation address/bytecode is the same as at scheduling. The actual implementation is chosen by ConcreteFactory.upgrade at execution via getImplementationByVersion(newVersion) and installed through upgradeToAndCall. Since ConcreteFactory is UUPSUpgradeable (governance-upgradeable), governance can change behavior (including implementation resolution) during the notice window. This allows last-minute replacement of the reviewed code and, if the new implementation removes timelock checks in its own _upgrade, enables immediate follow-on upgrades without notice. The result is a non-binding timelock that relies on governance discipline and off-chain monitoring rather than on-chain enforcement.

Severity

Impact Explanation: [High] Potential direct, material loss of principal funds if a malicious implementation is installed; core protections (timelock) can be stripped, and core functionality can be broken.

Likelihood Explanation: [Low] Exploitation requires trusted-role (governance/admin) misuse, malice, mistake, or compromise to change factory behavior/mapping late in the notice window and to execute the upgrade.

Exploitation

Exploitation Scenarios:

Scenario 1.

Last-minute factory change installs different code than reviewed: Governance schedules an upgrade to version V with calldata hash H; users audit implementation A seen in the schedule event. Shortly before execution time T, governance upgrades the factory logic or behavior so getImplementationByVersion(V) returns B instead of A. At/after T, the vault owner calls upgrade(vault, V, data with keccak256(data)=H); the proxy installs B and calls the new implementation’s _upgrade. The vault runs B, not the audited A, enabling malicious or materially different behavior that can cause asset loss.

Preconditions / Assumptions

- (a). GatedVaultImpl timelock is enabled with a valid nonzero delay
- (b). A scheduled upgrade exists to version V with pendingUpgradeDataHash H and executableAt T

- (c). Governance (trusted role) controls the UUPSUpgradeable ConcreteFactory and can change its behavior
- (d). Vault owner can call the factory's upgrade at/after T

Scenario 2.

Timelock stripping via repointed target enabling immediate follow-on malicious upgrade: Governance repoints version V to implementation B that omits or weakens GatedVaultImpl's timelock checks. At/after T, the upgrade to B executes. Immediately afterward, governance upgrades again to version V2 (marked migratable) without any schedule/delay because B's _upgrade does not enforce the timelock. This enables rapid malicious changes (e.g., fund drain) with no effective notice.

Preconditions / Assumptions

- (a). All preconditions from Scenario 1
- (b). getImplementationByVersion(V) can be made to return an implementation B that omits or weakens timelock checks in its _upgrade
- (c). Factory migratability allows a subsequent upgrade to version V2

Scenario 3.

Functional drift without clear malice undermines notice: Governance repoints version V to implementation B that is economically or behaviorally different (e.g., fee logic, hook behavior) shortly before T. At/after T, the upgrade installs B, not A. Users who relied on the timelock's notice window for A have insufficient time to assess B, resulting in unexpected economic consequences (e.g., higher fees or altered behavior).

Preconditions / Assumptions

- (a). All preconditions from Scenario 1
- (b). getImplementationByVersion(V) can be made to return a functionally different implementation B close to T

Proposed fix

GatedVaultImpl.sol

File: `src/GatedVaultImpl.sol`

Source

```
 ... 61 unchanged lines ...
         /// @dev Earliest timestamp at which the timelock disable request may be finalized.
         uint64 upgradeTimelockDisableExecutableAt;
+        // SECURITY NOTE: To bind timelocked upgrades to exact code, add the following fields:
+        // address pendingUpgradeExpectedImplementation;
+        // bytes32 pendingUpgradeExpectedCodehash;
+        // These must be populated in scheduleUpgrade(), enforced in _upgrade(), and cleared in _clearPendingUp
     }

 ... 253 unchanged lines ...
         IConcreteFactory factory_ = IConcreteFactory(factory);
         address implementation = factory_.getImplementationByVersion(newVersion);
+        // SECURITY TODO: Snapshot and persist the expected implementation (and optionally its extcodehash) her
+        // $.pendingUpgradeExpectedImplementation = implementation;
+        // $.pendingUpgradeExpectedCodehash = extcodehash(implementation);
         // Blocked versions are rejected even if their implementation exists.
         if (factory_.isBlocked(newVersion)) revert UpgradeVersionBlocked(newVersion);
 ... 264 unchanged lines ...
         if (!$.upgradeTimelockEnabled) return;

+        // SECURITY TODO: Enforce binding to the scheduled implementation address (and optional codehash) befor
+        // address currentImpl = _getERC1967Implementation();
+        // require(currentImpl == $.pendingUpgradeExpectedImplementation, "Impl mismatch");
+        // (optional) require(extcodehash(currentImpl) == $.pendingUpgradeExpectedCodehash, "Codehash mismatch"
         // A timed upgrade can execute only if governance explicitly queued it earlier.
         if ($.pendingUpgradeVersion == 0) revert UpgradeNotScheduled();
```

```
 ... 28 unchanged lines ...
         $.upgradeScheduledAt = 0;
         $.upgradeExecutableAt = 0;
+        // SECURITY TODO: Also clear implementation binding fields when added:
+        // $.pendingUpgradeExpectedImplementation = address(0); $.pendingUpgradeExpectedCodehash = bytes32(0);
     }

 ... 126 unchanged lines ...
```

ConcreteFactory.sol

File: `dependencies/concrete-2.0.0/src/factory/ConcreteFactory.sol`

Source

```
 ... 268 unchanged lines ...
         require(!isBlocked(newVersion), ImplementationBlocked());

+        // SECURITY TODO: For timelock-enabled vaults, pre-validate the scheduled state and bind to the expecte
+        // - Read from the vault: pendingUpgradeVersion, pendingUpgradeDataHash, executableAt, expectedImplemen
+        // - Require: newVersion matches; keccak256(data) matches; block.timestamp >= executableAt
+        // - Use the expectedImplementation (and optionally assert its codehash) for the upgrade target below
+
         IVaultProxy(vault)
             .upgradeToAndCall(
                 getImplementationByVersion(newVersion), abi.encodeCall(IUpgradeableVault.upgrade, (newVersion,
             );

         emit Migrated(vault, newVersion);
     }

     function _authorizeUpgrade(address newImplementation) internal override onlyOwner {}
 }
```

Related findings

[Low] Split upgrade authority (timelock vs execution) in GatedVaultImpl with Concrete factory causes governance DoS of upgrades

Description

GatedVaultImpl gives VAULT_MANAGER exclusive control over the upgrade timelock and scheduling, while the vault owner (via the Concrete factory) executes upgrades. If these authorities are not aligned, VAULT_MANAGER can block or severely delay upgrades, creating a governance DoS risk.

GatedVaultImpl enforces an on-chain upgrade timelock only when enabled. All timelock controls (enable/disable with delay, setUpgradeDelay, scheduleUpgrade, cancelUpgrade) are restricted to VAULT_MANAGER. Actual upgrade execution is performed by the vault owner through the Concrete factory, which calls vault.upgrade. When the timelock is enabled, GatedVaultImpl._upgrade requires a prior schedule, matching version and calldata hash, and the delay to have elapsed; otherwise it reverts. If VAULT_MANAGER and owner are different and not coordinated, VAULT_MANAGER can enable the timelock and refuse to schedule (freezing upgrades) or set an excessively long delay (hindering agility). This is an intentional design choice but introduces a real governance/operational risk of upgrade DoS.

Severity

Impact Explanation: [Medium] Interferes with important governance functionality (upgrades), causing significant availability degradation (DoS of upgrade execution) but does not directly freeze user funds or break core user flows.

Likelihood Explanation: [Low] Requires trusted admin (VAULT_MANAGER) misuse, malice, misconfiguration, or unavailability; thus low likelihood under the rules.

Exploitation

Exploitation Scenarios:

Scenario 1.

VAULT_MANAGER enables the timelock and refuses to schedule any upgrade: The owner attempts to upgrade via the factory; GatedVaultImpl._upgrade reverts with UpgradeNotScheduled because upgradeTimelockEnabled is true and no schedule exists, effectively freezing upgrades.

Preconditions / Assumptions

- (a). VAULT_MANAGER and vault owner are distinct or not aligned under the same governance
- (b). Concrete factory behaves as documented: only owner can initiate upgrade; factory calls vault.upgrade
- (c). GatedVaultImpl timelock is enabled by VAULT_MANAGER
- (d). VAULT_MANAGER does not call scheduleUpgrade
- (e). Owner attempts to upgrade to a factory-approved, migratable version

Scenario 2.

VAULT_MANAGER sets an excessively long upgradeDelay and enables the timelock, then schedules upgrades only under that delay: Any upgrade requires waiting the full long delay before execution; urgent fixes cannot be applied promptly, significantly degrading governance responsiveness.

Preconditions / Assumptions

- (a). VAULT_MANAGER and vault owner are distinct or not aligned under the same governance
- (b). Concrete factory behaves as documented: only owner can initiate upgrade; factory calls vault.upgrade
- (c). VAULT_MANAGER sets upgradeDelay to a very large value (>= MIN_UPGRADE_DELAY) and enables the timelock
- (d). VAULT_MANAGER schedules the desired upgrade subject to the long delay
- (e). Owner or governance needs to execute the upgrade after the waiting period

Proposed fix

GatedVaultImpl.sol

File: `src/GatedVaultImpl.sol`

Source

```
 ... 288 unchanged lines ...
     /// @dev The function stores the version pair and calldata hash before emitting the schedule event so off-c
     ///      watchers can treat the emitted values as a complete snapshot of pending upgrade state.
-    function scheduleUpgrade(uint64 newVersion, bytes calldata data) external onlyRole(RolesLib.VAULT_MANAGER)
+    function scheduleUpgrade(uint64 newVersion, bytes calldata data) external {
+        if (_msgSender() != owner()) _checkRole(RolesLib.VAULT_MANAGER);
         GatedVaultStorage storage $ = _getGatedVaultStorage();
         // Scheduling is opt-in; deployments can leave the timelock disabled until governance is ready to use i
 ... 37 unchanged lines ...
     /// @notice Cancels the currently queued upgrade.
     /// @dev Cancellation clears all scheduling metadata so a later schedule starts from a clean slate.
-    function cancelUpgrade() external onlyRole(RolesLib.VAULT_MANAGER) {
+    function cancelUpgrade() external {
+        if (_msgSender() != owner()) _checkRole(RolesLib.VAULT_MANAGER);
         GatedVaultStorage storage $ = _getGatedVaultStorage();
         uint64 fromVersion = $.pendingUpgradeFromVersion;
 ... 413 unchanged lines ...
```

[Low] Legacy storage slot reuse in GatedVaultImpl upgrades causes unintended timelock activation and blocked upgrades

Description

GatedVaultImpl reuses a fixed ERC-7201 storage slot for new timelock state without a migration/wipe on upgrade. If upgrading from a legacy implementation that used the same slot for different fields with non-zero values, the new code

can misinterpret them as active timelock/pending-upgrade state, blocking immediate upgrades and requiring manager intervention.

GatedVaultImpl introduces a new ERC-7201 storage struct at a fixed slot (0x7ed7…b800) and initializes it only in _initialize(), which is not executed on upgrades. Tests model a legacy layout using the same slot for different fields (paused, pauseDelay, executableAt, pausedEpochID). If a real legacy proxy with any of these fields non-zero is upgraded to GatedVaultImpl, the new logic reinterprets them as upgradeTimelockEnabled/upgradeDelay/pendingUpgradeVersion, etc. When upgradeTimelockEnabled is accidentally true, the initial factory-driven upgrade requires a prior schedule and thus reverts, effectively forcing VAULT_MANAGER cooperation and blocking owner-only upgrades. Non-zero legacy values can also create phantom pending-upgrade state and unsafe (below-minimum) delays that prevent scheduling until the manager cancels/updates timelock state. Fresh deployments are unaffected; the risk is conditional on migrating from a legacy that used the same storage slot and had non-zero fields.

Severity

Impact Explanation: [Medium] Important governance functionality (upgrade path) is blocked or distorted until corrective actions are taken, but deposits/withdrawals and user funds are not directly affected.

Likelihood Explanation: [Low] Requires a real legacy implementation sharing the exact storage slot with non-zero legacy fields and, in the most severe case, trusted role (VAULT_MANAGER) refusal or delay to cooperate; these are rare/exceptional conditions.

Exploitation

Exploitation Scenarios:

Scenario 1.

Unintended timelock activation (legacy paused → upgradeTimelockEnabled) blocks the first upgrade: owner calls factory.upgrade, but _upgrade sees timelock enabled without a prior schedule and reverts, forcing VAULT_MANAGER to schedule/adjust timelock before any upgrade can proceed.

Preconditions / Assumptions

- (a). There exists a legacy implementation that used the same ERC-7201 storage slot (0x7ed7…b800) with different fields
- (b). Legacy state has paused == true at the time of upgrade
- (c). OWNER initiates the first upgrade to GatedVaultImpl via ConcreteFactory.upgrade
- (d). VAULT_MANAGER is distinct from OWNER and controls schedule/timelock functions

Scenario 2.

Phantom pending upgrade (legacy pausedEpochID spills into pendingUpgradeVersion) prevents scheduling: later scheduleUpgrade attempts revert with UpgradeAlreadyScheduled until VAULT_MANAGER calls cancelUpgrade to clear stale state.

Preconditions / Assumptions

- (a). There exists a legacy implementation that used the same ERC-7201 storage slot (0x7ed7…b800) with different fields
- (b). Legacy state has paused == false (so the initial upgrade succeeds)
- (c). Legacy pausedEpochID is non-zero with bits such that pendingUpgradeVersion decodes to non-zero after upgrade
- (d). VAULT_MANAGER later attempts to schedule an upgrade

Scenario 3.

Corrupted short/zero upgrade delay (legacy pauseDelay → upgradeDelay) prevents scheduling: with timelock enabled but upgradeDelay < MIN_UPGRADE_DELAY, scheduleUpgrade reverts with UpgradeDelayTooShort until VAULT_MANAGER sets a compliant delay.

Preconditions / Assumptions

- (a). There exists a legacy implementation that used the same ERC-7201 storage slot (0x7ed7...b800) with different fields
- (b). Legacy state has paused == true and pauseDelay < MIN_UPGRADE_DELAY
- (c). OWNER attempts the first upgrade; then VAULT_MANAGER attempts scheduleUpgrade

Proposed fix

GatedVaultImpl.sol

File: `src/GatedVaultImpl.sol`

Source

```
 ... 290 unchanged lines ...
     function scheduleUpgrade(uint64 newVersion, bytes calldata data) external onlyRole(RolesLib.VAULT_MANAGER)
         GatedVaultStorage storage $ = _getGatedVaultStorage();
+        // Sanitize inconsistent legacy-collision pending-upgrade state before scheduling.
+        if (
+            $.pendingUpgradeVersion != 0 && ($.upgradeScheduledAt == 0 || $.upgradeExecutableAt == 0)
+        ) {
+            _clearPendingUpgrade($);
+        }
         // Scheduling is opt-in; deployments can leave the timelock disabled until governance is ready to use i
         if (!$.upgradeTimelockEnabled) revert UpgradeTimelockDisabled();
 ... 289 unchanged lines ...
     function _upgrade(uint64, uint64 newVersion, bytes calldata data) internal virtual override {
         GatedVaultStorage storage $ = _getGatedVaultStorage();
+        // Sanitize legacy-collision state: disable timelock if configured with an invalid short delay.
+        if ($.upgradeTimelockEnabled && $.upgradeDelay < MIN_UPGRADE_DELAY) {
+            $.upgradeTimelockEnabled = false;
+            _clearPendingTimelockDisable($);
+            if ($.pendingUpgradeVersion != 0) _clearPendingUpgrade($);
+        }
+        // Drop inconsistent phantom pending-upgrade state from legacy slot collisions.
+        if (
+            $.pendingUpgradeVersion != 0 && ($.upgradeScheduledAt == 0 || $.upgradeExecutableAt == 0)
+        ) {
+            _clearPendingUpgrade($);
+        }
         // If governance has not enabled the timelock, the parent upgrade flow proceeds without extra checks.
         if (!$.upgradeTimelockEnabled) return;
 ... 161 unchanged lines ...
```

8. [Medium] Unsigned transfer recipient and upgradeable spender in GatedVaultImpl Permit2 deposit cause cross-upgrade signature reuse to steal user tokens

Status

Review status: Unresolved
Remediation status: Unremediated
Remediation note: Created by pipeline analysis

Description

GatedVaultImpl's Permit2-based deposit binds user authorization to an upgradeable proxy spender and a witness that omits the transfer recipient. Any still-valid Permit2 signatures can be reused after an upgrade to transfer user tokens to arbitrary addresses without minting shares. The upgrade timelock is disabled by default, increasing exposure.

In GatedVaultImpl.depositWithPermit2, the contract calls Permit2's permitWitnessTransferFrom from the vault proxy, making the proxy the authorized spender across upgrades. The signed witness commits only to {owner, assets, minShares}, while SignatureTransferDetails.to (the recipient) is chosen by the spender at execution and is not signed. Today's implementation sets to = address(this) and mints shares to the owner, but a future implementation at the same proxy address could redirect tokens to an attacker address and omit minting. Since the upgrade timelock starts disabled, a

privileged upgrader can immediately switch implementations and use any outstanding, valid user signatures (nonce unused, deadline not expired) to transfer tokens directly from user wallets, bypassing share minting.

Severity

Impact Explanation: [High] Direct, material loss of user principal as tokens are transferred from user wallets to attacker addresses without shares minted.

Likelihood Explanation: [Low] Exploitation requires trusted role misuse/compromise (upgrade governance and often relayer/signature custody) and valid outstanding signatures; per rules, admin misuse sets likelihood to Low.

Exploitation

Exploitation Scenarios:

Scenario 1.

Unified control over upgrades and signature collection; timelock disabled: The same governance controls the vault upgrade (and factory registration) and the relayer holding user signatures. They upgrade to a malicious implementation immediately and submit the collected signatures through Permit2 with to=attacker and no minting, draining users' wallets.

Preconditions / Assumptions

- (a). Upgrade authority (governance/VAULT_MANAGER) controls vault upgrades and factory version registration
- (b). The same entity controls relayer/WITHDRAWAL_MANAGER and possesses users' Permit2 signatures
- (c). upgradeTimelockEnabled is false (default) allowing immediate upgrade
- (d). User signatures are valid (deadlines in future, nonces unused)

Scenario 2.

Governance colludes with or compromises the relayer; timelock disabled: Upgrade authority gains access to user signatures via collusion/compromise of the relayer, upgrades immediately, and consumes signatures with to=attacker via Permit2, omitting share minting.

Preconditions / Assumptions

- (a). Upgrade authority (governance/VAULT_MANAGER) controls vault upgrades (and factory registration if needed)
- (b). Relayer is compromised or colludes, providing attackers with users' Permit2 signatures
- (c). upgradeTimelockEnabled is false allowing immediate upgrade
- (d). User signatures are valid (deadlines in future, nonces unused)

Scenario 3.

Timelock enabled but long deadlines and inattentive users: Governance schedules an upgrade to a malicious implementation. Users do not cancel nonces and signatures have long deadlines. After the delay, the upgrade executes and signatures are used with to=attacker, resulting in direct user wallet drains without shares.

Preconditions / Assumptions

- (a). upgradeTimelockEnabled is true with a non-zero delay
- (b). Upgrade authority controls both vault upgrades and factory registration
- (c). Attacker has access to users' Permit2 signatures (e.g., centralized relayer custody)
- (d). Signatures have long deadlines and users do not cancel nonces during the timelock window

Proposed fix

GatedVaultImpl.sol

File: `src/GatedVaultImpl.sol`

Source

```
... 110 unchanged lines ...
    uint64 public immutable MIN_UPGRADE_DELAY;
```

```
+    // IMPORTANT: Permit2's witness does not include the transfer recipient `to`, and the authorized spender is
+    // upgradeable proxy. To prevent cross-upgrade reuse of signatures to arbitrary recipients, route Permit2 c
+    // through a non-upgradeable, per-vault helper (spender) that hardcodes `to = vault` and is the address use
     /// @dev Permit2 witness typehash for deposit intents.
     bytes32 private constant _DEPOSIT_WITNESS_TYPEHASH =
 ... 519 unchanged lines ...

     /// @notice Deposit on behalf of a user using their Permit2 signature.
+    // TODO(security): Migrate this entrypoint to call a non-upgradeable Permit2Spender helper (fixed spender)
+    // that hardcodes `to = address(this)` and is only callable by this vault. Afterwards, deprecate and disabl
+    // accepting signatures that target the proxy as spender.
     /// @dev Pulls tokens from `owner` via Permit2, updates cached total assets, and mints shares to `owner`.
     ///      Enforces the same deposit limits and hooks as the standard ERC-4626 deposit path.
 ... 94 unchanged lines ...
         private
     {
+        // SECURITY NOTE: SignatureTransferDetails.to is selected by the spender at execution and is not signed
+        // Using the upgradeable proxy as spender allows future implementations to redirect funds. Use a dedica
+        // non-upgradeable spender that forces `to = address(this)` and is onlyVault callable.
         // Permit2 verifies the signature, nonce, deadline, allowance ceiling, and deposit witness.
         PERMIT2.permitWitnessTransferFrom(
             ISignatureTransfer.PermitTransferFrom({
                 permitted: ISignatureTransfer.TokenPermissions({ token: asset(), amount: assets }),
                 nonce: nonce,
                 deadline: deadline
             }),
             ISignatureTransfer.SignatureTransferDetails({ to: address(this), requestedAmount: assets }),
             owner,
             _hashDepositWitness(owner, assets, minShares),
             _DEPOSIT_WITNESS_TYPESTRING,
             signature
         );
     }
 }
```

Related findings

[Medium] Missing epoch/version binding in Permit2 deposit witness in GatedVaultImpl across upgrades causes cross-upgrade signature replay and potential asset theft

Description

Permit2 deposit signatures bind only owner, assets, and minShares and are scoped to the proxy (spender), not the implementation. Upgrades do not invalidate these signatures, so archived Permit2 deposit authorizations remain valid across upgrades. If a future implementation reuses the same witness but changes deposit semantics, old signatures can be executed under new rules.

GatedVaultImpl's depositWithPermit2 computes a Permit2 witness over {owner, assets, minShares} and calls permitWitnessTransferFrom. Permit2 binds signatures to the proxy (spender) and TokenPermissions(token, amount), but the witness does not include any implementation version/epoch. The upgrade path (scheduleUpgrade/_upgrade) does not rotate or invalidate signatures; it only enforces a timelock and calldata consistency. Consequently, any unconsumed Permit2 deposit signatures remain valid post-upgrade, provided asset() is unchanged and deadlines/nonces permit it. The current implementation mints shares to the owner and enforces a minShares floor, so immediate third-party abuse is limited. However, if a future implementation preserves the same witness type but changes deposit semantics (e.g., recipient or pricing), archived signatures may be executed under the new rules without fresh user consent, enabling direct theft (if recipient semantics change) or unexpected economic loss (if fees/pricing change). During gating, only the manager can execute; after gating is disabled, depositWithPermit2 becomes permissionless.

Severity

Impact Explanation: [High] Scenarios 1 and 2 can result in direct, material loss of principal by diverting minted shares after Permit2 pulls assets from the victim.

Likelihood Explanation: [Low] Exploitation requires governance to ship a semantics-changing implementation that preserves the same witness type and the existence of valid archived signatures; Scenario 2 additionally requires trusted role misuse. These multiple rare preconditions reduce likelihood.

Exploitation

Exploitation Scenarios:

Scenario 1.

After an upgrade that keeps the same DepositWitness type but changes deposit semantics to mint shares to msg.sender, a third party who has a user's archived valid signature calls depositWithPermit2 (post-gating, permissionless). Permit2 transfers the user's assets to the vault, and the new implementation mints shares to the attacker, resulting in direct theft of the user's principal.

Preconditions / Assumptions

- (a). Governance deploys a new implementation that reuses the same DepositWitness type (no epoch/version binding) and changes recipient semantics (e.g., mint to msg.sender).
- (b). asset() remains the same across the upgrade so TokenPermissions.token still matches.
- (c). Archived, valid user signatures exist (deadline not expired, nonce unused) and are accessible to an attacker.
- (d). Deposits are not gated (depositsGated == false), making depositWithPermit2 permissionless.

Scenario 2.

During the gated period, the WITHDRAWAL_MANAGER holds archived user signatures. Governance upgrades the implementation (same witness type) to redirect minted shares away from the owner (e.g., to msg.sender). The manager executes the old signature: Permit2 pulls the user's assets and the new implementation mints shares to the manager, stealing the user's principal.

Preconditions / Assumptions

- (a). depositsGated == true, so only WITHDRAWAL_MANAGER can call depositWithPermit2.
- (b). WITHDRAWAL_MANAGER possesses archived, valid user signatures.
- (c). Governance deploys an implementation that reuses the same DepositWitness type and changes recipient semantics away from owner.
- (d). asset() remains the same; signatures are still valid (deadline not expired, nonce unused).

Scenario 3.

Governance upgrades the implementation (same witness type) to introduce a deposit fee or altered pricing/rounding while asset() is unchanged. A stored, still-valid user signature with a loose minShares is executed (by manager during gating or anyone post-gating). Permit2 pulls assets and the user receives fewer shares than expected pre-upgrade (but >= minShares), causing unexpected economic loss due to the new policy.

Preconditions / Assumptions

- (a). Governance deploys an implementation that reuses the same DepositWitness type and changes deposit economics (e.g., adds a fee or different pricing/rounding).
- (b). asset() remains the same; signatures are still valid (deadline not expired, nonce unused).
- (c). A user previously signed with a loose minShares (or zero) allowing execution under changed economics.
- (d). Execution by manager during gating or anyone post-gating.

9. [Medium] Pre-deposit hook after asset snapshot in GatedVaultImpl.depositWithPermit2 causes overmint/dilution and deposit-cap bypass

Status

Review status: Unresolved
Remediation status: Unremediated
Remediation note: Created by pipeline analysis

Description

GatedVaultImpl.depositWithPermit2 snapshots cachedTotalAssets, then calls an external preDeposit hook before enforcing deposit bounds and pricing shares. A misconfigured, privileged hook can change strategy NAV during this window (e.g., by harvesting rewards), so the deposit mints against a stale, smaller asset base, overminting shares and potentially bypassing deposit limits.

In GatedVaultImpl.depositWithPermit2, withYieldAccrual updates cachedTotalAssets and the function snapshots totalAssetsBeforeDeposit. Next, an external preDeposit hook is invoked. Only after the hook returns does the function enforce deposit limits and compute shares using the earlier snapshot. Hooks are external and, under the assumptions, can be adversarial/misconfigured. If the hook also holds strategy roles (e.g., OPERATOR_ROLE/STRATEGY_ADMIN on AusdStrategy), it can change live NAV mid-call (for example, via RewardsAdapter's reward claim/processing), after the snapshot but before share pricing. The vault's nonReentrant guard does not block these strategy calls. As a result, deposits are priced with stale totalAssetsBeforeDeposit: a NAV increase yields overminted shares and may let assets + totalAssetsBeforeDeposit pass caps that assets + true_NAV would fail. Conversely, a NAV decrease can under-mint the depositor unless their minShares blocks it.

Severity

Impact Explanation: [High] Incorrect share minting against a stale asset base directly dilutes existing LPs' principal (overmint) and can bypass deposit limits; under certain actions it can under-mint depositors.

Likelihood Explanation: [Low] Exploitation requires a misconfigured pre-deposit hook to hold privileged strategy roles (OPERATOR_ROLE and/or STRATEGY_ADMIN). This is trusted-role misuse and thus low likelihood.

Exploitation

Exploitation Scenarios:

Scenario 1.

Overmint via reward harvest during preDeposit: A pre-deposit hook holding OPERATOR_ROLE on AusdStrategy calls claimAndProcessRewards (or claimRewards + processRewards to WMON/MON) during preDeposit, increasing NAV (Y) after the vault snapshots T. The vault then mints shares using T, overminting the depositor and potentially letting assets + T pass the cap where assets + (T + Y) would not. Existing LPs are diluted.

Preconditions / Assumptions

- (a). A pre-deposit hook is configured and enabled for the vault
- (b). The hook holds OPERATOR_ROLE on the target AusdStrategy
- (c). The strategy has harvestable rewards or reward balances to normalize to WMON/MON
- (d). Deposits are executed via depositWithPermit2 by a WITHDRAWAL_MANAGER
- (e). The attacker coordinates a user Permit2 deposit so the hook runs during that call

Scenario 2.

Overmint and cap bypass via MON->vault-asset swap during preDeposit (admin-configured): With RewardSwapConfig set by STRATEGY_ADMIN, the hook calls claimAndProcessRewards with outputAsset = VAULT_ASSET, unwrapping and swapping MON to the vault asset. NAV increases by Y in-stablecoin after the snapshot. Shares and cap checks still use T, causing overmint and easier cap bypass. Existing LPs are diluted.

Preconditions / Assumptions

- (a). A pre-deposit hook is configured and enabled for the vault
- (b). The hook (or its controller) has STRATEGY_ADMIN on AusdStrategy and RewardSwapConfig is set
- (c). The hook holds OPERATOR_ROLE on AusdStrategy
- (d). The strategy has harvestable rewards convertible to the vault asset
- (e). Deposits are executed via depositWithPermit2 by a WITHDRAWAL_MANAGER

Scenario 3.

Under-mint a depositor by reducing NAV during preDeposit: A pre-deposit hook with STRATEGY_ADMIN calls an accounting-reducing action (e.g., forceRemoveUnstakeRequest) after the snapshot, decreasing NAV by Y. The vault mints using stale T, so the depositor receives fewer shares than justified by post-hook NAV unless protected by a sufficiently high minShares.

Preconditions / Assumptions

- (a). A pre-deposit hook is configured and enabled for the vault
- (b). The hook holds STRATEGY_ADMIN on AusdStrategy
- (c). There exists an unstake request or similar accounting action that can reduce NAV post-snapshot
- (d). The depositor's minShares is low enough not to block the under-mint
- (e). Deposits are executed via depositWithPermit2 by a WITHDRAWAL_MANAGER

Proposed fix

GatedVaultImpl.sol

File: `src/GatedVaultImpl.sol`

Source

```
 ... 631 unchanged lines ...
                                  PERMIT2 DEPOSITS
           ////////////////////////////////////////////////////////////*/
+    function _syncYieldAccrual() private withYieldAccrual { }

     /// @notice Deposit on behalf of a user using their Permit2 signature.
 ... 33 unchanged lines ...
         }

+        _syncYieldAccrual();
+        totalAssetsBeforeDeposit = cachedTotalAssets();
+
         // Reuse the same min/max deposit guard enforced by the regular ERC-4626 path.
         _enforcePermit2DepositBounds(assets, totalAssetsBeforeDeposit);
 ... 76 unchanged lines ...
```

Related findings

[Medium] Missing exact-transfer enforcement in GatedVaultImpl deposit paths causes over-minting and principal loss

Description

Deposit paths compute shares and update cachedTotalAssets using the nominal assets input without verifying the actual token amount received. With fee-on-transfer or short-delivery assets, this leads to over-minting and inflated accounting, enabling withdrawals exceeding real deposits and diluting existing holders.

In GatedVaultImpl.depositWithPermit2, shares are priced from the user-supplied assets, Permit2 is called to transfer that amount, then cachedTotalAssets is incremented by assets and shares are minted—without checking the vault's real balance delta. Permit2 does not guarantee exact delivery for fee-on-transfer tokens. The standard ERC-4626 deposit/mint path inherited from the parent implementation follows the same accounting assumption. If the asset is short-delivery (e.g., fee-on-transfer), a deposit of A that actually delivers A' < A still mints shares and updates cachedTotalAssets as if A arrived. Withdrawals then pay out based on the overstated accounting, causing direct loss to the vault and dilution of existing holders. Yield accrual does not reconcile idle-balance shortfalls, so the drift persists and can inflate fee minting or distort deposit caps.

Severity

Impact Explanation: [High] Direct, material loss of principal funds and dilution of existing holders by over-minting shares and paying out withdrawals based on overstated accounting.

Likelihood Explanation: [Low] Requires an admin/governance misconfiguration (selection of a short-delivery asset or lack of protective hooks). Once present, exploitation is straightforward and profitable.

Exploitation

Exploitation Scenarios:

Scenario 1.

Public deposits open: An attacker deposits A of a fee-on-transfer asset; the vault receives only A' < A but mints shares and increments cachedTotalAssets using A. The attacker immediately withdraws the nominal amount A, extracting the difference (A − A') as profit. This can be repeated to drain liquidity.

Preconditions / Assumptions

- (a). The selected vault asset is short-delivery (e.g., fee-on-transfer or transfers less than requested).
- (b). Deposits are open (depositsGated() == false).
- (c). No protective hook is configured to enforce received == assets.
- (d). The vault maintains sufficient liquidity (idle or via deallocations) to honor withdrawals.
- (e). Permit2 and ERC20 function as expected without enforcing exact delivery.

Scenario 2.

Deposits gated: A malicious or compromised WITHDRAWAL_MANAGER submits an attacker's Permit2 deposit of A for a short-delivery asset; the vault receives A' < A but mints shares and accounts for A. The attacker then withdraws A, realizing the difference (A − A') at the expense of existing holders.

Preconditions / Assumptions

- (a). Deposits are gated (depositsGated() == true).
- (b). WITHDRAWAL_MANAGER is malicious or compromised and will submit the attacker's Permit2 deposit.
- (c). The selected vault asset is short-delivery.
- (d). No protective hook is configured to enforce received == assets.
- (e). The vault maintains sufficient liquidity to honor withdrawals.

Scenario 3.

Accounting drift and fee/cap distortion: Repeated short-delivery deposits (A_i delivering A_i' < A_i) cause cachedTotalAssets to drift upward relative to real balances, inflating fee share minting and prematurely tripping deposit caps, diluting holders and potentially causing later withdrawal shortfalls.

Preconditions / Assumptions

- (a). The selected vault asset is short-delivery.
- (b). Multiple deposits occur without a check that received == assets.
- (c). Management/performance fees are active or later applied based on cachedTotalAssets.
- (d). No protective hook is configured to enforce received == assets.
- (e). Deposit limits and AUM-dependent logic rely on cachedTotalAssets.

Proposed fix

GatedVaultImpl.sol

File: `src/GatedVaultImpl.sol`

Source

```
// SPDX-License-Identifier: AGPL-3.0
pragma solidity 0.8.28;

import { ConcreteAsyncVaultImpl } from "@concrete/implementation/ConcreteAsyncVaultImpl.sol";
import { ConcreteStandardVaultImpl } from "@concrete/implementation/ConcreteStandardVaultImpl.sol";
import { IConcreteFactory } from "@concrete/interface/IConcreteFactory.sol";
import { ConcreteV2ConversionLib as ConversionLib } from "@concrete/lib/Conversion.sol";
import { Hooks, HooksLibV1 as HooksLib } from "@concrete/lib/Hooks.sol";
import { ConcreteV2RolesLib as RolesLib } from "@concrete/lib/Roles.sol";
import {
    ConcreteCachedVaultStateStorageLib as CachedVaultStateLib
} from "@concrete/lib/storage/ConcreteCachedVaultStateStorageLib.sol";
import {
    ConcreteStandardVaultImplStorageLib as SVLib
} from "@concrete/lib/storage/ConcreteStandardVaultImplStorageLib.sol";
import { IERC4626 } from "@openzeppelin/contracts/interfaces/IERC4626.sol";
import { ERC4626Upgradeable } from "@openzeppelin/contracts-upgradeable/token/ERC20/extensions/ERC4626Upgradeab
```

```
+import { IERC20 } from "@openzeppelin/contracts/token/ERC20/IERC20.sol";
 import { Math } from "@openzeppelin/contracts/utils/math/Math.sol";
 import { ISignatureTransfer } from "@permit2/interfaces/ISignatureTransfer.sol";
 ... 110 unchanged lines ...
     /// @param minShares Minimum share amount encoded in the signed deposit intent.
     error Permit2SharesBelowMin(uint256 shares, uint256 minShares);
+    /// @notice Raised when the vault receives fewer tokens than requested during a deposit.
+    error ShortDelivery(uint256 requested, uint256 received);

     /// @notice Emitted whenever the direct ERC-4626 deposit path is opened or closed.
 ... 542 unchanged lines ...
         if (shares < minShares) revert Permit2SharesBelowMin(shares, minShares);

+        // Record balance before transfer to enforce exact-transfer invariant.
+        uint256 preBalance = IERC20(asset()).balanceOf(address(this));
         // Pull exactly `assets` from the user while binding the signature to the deposit action and share floo
         _permit2DepositAssetFromOwner(owner, assets, minShares, nonce, deadline, signature);
+        // Verify that the vault actually received exactly `assets`.
+        uint256 received = IERC20(asset()).balanceOf(address(this)) - preBalance;
+        if (received != assets) revert ShortDelivery(assets, received);

         // Update cached total assets to keep vault accounting in sync
 ... 67 unchanged lines ...
```

10. [Medium] Permissive rescueToken in BaseStrategy (under AusdStrategy multi-asset NAV) allows STRATEGY_ADMIN to drain NAV-critical tokens causing user principal loss

Status

Review status: Unresolved
Remediation status: Unremediated
Remediation note: Created by pipeline analysis

Description

BaseStrategy.rescueToken blocks only the vault asset, but AusdStrategy treats shMON and WMON as core NAV. A STRATEGY_ADMIN can transfer out shMON/WMON, reducing NAV that backs vault shares. Additionally, a misconfigured pre-deposit hook holding STRATEGY_ADMIN can drain NAV after the pre-deposit snapshot in depositWithPermit2, under-minting depositor shares. Admin draining can also impede termination.

AusdStrategy's NAV includes idle shMON, idle WMON, idle MON, idle stablecoin, shMON collateral, and pending unstake MON. However, BaseStrategy.rescueToken only forbids rescuing the vault's asset() (stablecoin), allowing STRATEGY_ADMIN to transfer out NAV-critical shMON/WMON directly. This creates a privileged centralization risk: NAV can be siphoned, harming existing depositors when yield accrual updates total assets. In GatedVaultImpl.depositWithPermit2, withYieldAccrual snapshots cachedTotalAssets, then a pre-deposit hook runs, and shares are calculated from the pre-hook snapshot. If a pre-deposit hook is misconfigured to also have STRATEGY_ADMIN on the strategy, it can rescue shMON/WMON after the snapshot but before share calculation, causing depositors to receive fewer shares than fair (under-mint). Separately, draining WMON can prevent full repayment of primary WMON debt during termination, reverting and delaying unwind.

Severity

Impact Explanation: [High] Direct, material loss of principal funds due to STRATEGY_ADMIN draining NAV-critical tokens; under-minting causes direct depositor loss; termination sabotage can significantly disrupt core functionality.

Likelihood Explanation: [Low] All scenarios require trusted-role (STRATEGY_ADMIN) misuse or a misconfiguration granting STRATEGY_ADMIN to a hook; these are privileged/operator failures.

Exploitation

Exploitation Scenarios:

Scenario 1.

Scenario 1 (Admin drain): A STRATEGY_ADMIN calls rescueToken on shMON or WMON while the strategy holds idle balances, transferring these tokens to the admin. Since shMON/WMON are included in NAV, this directly reduces the vault's total assets backing all shares; users realize principal loss on the next accrual/withdrawal.

Preconditions / Assumptions

- (a). Caller holds STRATEGY_ADMIN on the strategy
- (b). Strategy holds non-trivial idle shMON or WMON balances

Scenario 2.

Scenario 2 (Pre-deposit hook under-mint due to misconfiguration): A pre-deposit hook is configured and also holds STRATEGY_ADMIN on the strategy. During depositWithPermit2, after withYieldAccrual snapshots total assets, the hook calls rescueToken on shMON/WMON. Shares are then computed from the stale, higher pre-hook snapshot, so the depositor mints fewer shares than fair for the same assets.

Preconditions / Assumptions

- (a). A pre-deposit hook is configured on the vault
- (b). The pre-deposit hook (or a callee) also holds STRATEGY_ADMIN on the strategy (misconfiguration)
- (c). The strategy holds idle shMON/WMON at deposit time
- (d). A depositWithPermit2 call is executed

Scenario 3.

Scenario 3 (Termination sabotage): With primary WMON debt open, STRATEGY_ADMIN rescues idle WMON needed for repayment. In _terminatePositions, the strategy detects insufficient WMON to fully repay and reverts, blocking the unwind, potentially delaying user liquidity.

Preconditions / Assumptions

- (a). Primary WMON debt is open
- (b). Idle WMON (or expected proceeds) is needed to fully repay
- (c). Caller holding STRATEGY_ADMIN rescues WMON before termination

Proposed fix

AusdStrategy.sol

File: `src/strategies/AusdStrategy.sol`

Source

```
 ... 863 unchanged lines ...
     }

+    /// @notice Restrict rescue of NAV-critical tokens (stablecoin, shMON, WMON).
+    function rescueToken(address token, uint256 amount) external override onlyStrategyAdmin {
+        if (token == address(STABLECOIN) || token == address(SHMON) || token == address(WMON)) {
+            revert CannotRescueCoreAsset();
+        }
+        uint256 bal = IERC20(token).balanceOf(address(this));
+        uint256 toSend = amount == 0 ? bal : amount;
+        address admin = strategyAdmin();
+        if (admin == address(0)) revert InvalidParams();
+        IERC20(token).safeTransfer(admin, toSend);
+    }
     function _deployMigratedWMonUnwrapper() internal override returns (address) {
         return address(new WMonUnwrapper(address(this), address(WMON)));
 ... 321 unchanged lines ...
```

11. [Medium] Unbounded gas forwarding during ERC20 calls in VaultFeeAllocator causes token-level distribution DoS and frozen funds

Status

Review status: Unresolved
Remediation status: Unremediated
Remediation note: Created by pipeline analysis

Description

VaultFeeAllocator’s distribution and emergency-withdraw flows can be DoSed for a given token because ERC20 calls forward nearly all gas. A malicious or buggy token can consume the forwarded gas in balanceOf/transfer, leaving insufficient gas for the allocator’s post-call storage updates and events, causing out-of-gas reverts that prevent payouts and block rescue of that token.

VaultFeeAllocator computes distributable balances via _availableBalance(token) which calls token.balanceOf(this), then iterates partners and uses _tryTransfer -> SafeERC20.trySafeTransfer for each payout. trySafeTransfer forwards nearly all gas to token.transfer and returns a boolean on failure. A malicious/broken ERC20 can consume this gas (gas-grief), causing the allocator to return with too little gas to complete _reservePendingFees (SSTOREs) and event emission, or to proceed to subsequent transfers (e.g., next partner or owner residual). The function then runs out of gas and reverts the entire distribution, preventing any payouts for that token. Similarly, emergencyWithdraw first calls balanceOf and then safeTransfer, both of which can be blocked by gas-grief or reverts, trapping funds. The effect is token-scoped fund freezing with no practical on-chain workaround, contradicting the intended fault tolerance of reserving failed transfers.

Severity

Impact Explanation: [High] For the affected token, distributions and emergency withdrawals can be blocked indefinitely, freezing funds with no practical on-chain workaround.

Likelihood Explanation: [Low] Exploitation requires an integrated ERC20 to misbehave or be malicious (e.g., via upgrade or nonstandard implementation), which is an external integration failure and provides no direct profit (griefing).

Exploitation

Exploitation Scenarios:

Scenario 1.

Integrated fee token becomes malicious and gas-griefs during balanceOf and transfer; distributeFees(token) reverts due to insufficient post-call gas to update pending reservations or complete the loop, freezing that token’s payouts.

Preconditions / Assumptions

- (a). At least one active partner exists in partners[]
- (b). Allocator holds a nonzero balance of an integrated ERC20 used for fees
- (c). That ERC20 behaves maliciously/buggily by consuming most forwarded gas in balanceOf and/or transfer and failing transfers (revert or false)
- (d). Someone calls distributeFees(token)

Scenario 2.

With multiple recipients (partners and/or owner residual), the first gas-griefing transfer leaves limited gas; a second transfer attempt reduces gas further, causing out-of-gas during subsequent storage updates and reverting the entire distribution for that token.

Preconditions / Assumptions

- (a). Multiple active partners or totalAllocatedBps < 10_000 (nonzero owner residual)
- (b). Allocator holds a nonzero balance of an integrated ERC20 used for fees
- (c). That ERC20 behaves maliciously/buggily by consuming most forwarded gas in transfer and failing transfers (revert or false)
- (d). Someone calls distributeFees(token)

Scenario 3.

Owner attempts emergencyWithdraw(token, recipient, amount); the malicious token gas-griefs or reverts in balanceOf and/or transfer, causing the rescue transaction to revert and leaving the token's funds stuck.

Preconditions / Assumptions

- (a). Allocator holds a nonzero balance of an integrated ERC20
- (b). That ERC20 behaves maliciously/buggily by gas-griefing or reverting in balanceOf and/or transfer
- (c). Owner calls emergencyWithdraw(token, recipient, amount)

Proposed fix

VaultFeeAllocator.sol

File: `src/fee/VaultFeeAllocator.sol`

Source

```
 ... 536 unchanged lines ...
     /// @return success Whether the transfer call reported success.
     function _tryTransfer(address token, address recipient, uint256 amount) private returns (bool) {
-        return IERC20(token).trySafeTransfer(recipient, amount);
+        (bool success, bytes memory data) =
+            token.call{ gas: 100000 }(abi.encodeWithSelector(IERC20.transfer.selector, recipient, amount));
+        return success && (data.length == 0 || (data.length == 32 && abi.decode(data, (bool))));
     }

 ... 49 unchanged lines ...
     /// @return availableBalance Token balance minus all reserved pending-fee obligations.
     function _availableBalance(address token) private view returns (uint256) {
-        uint256 totalBalance = IERC20(token).balanceOf(address(this));
+        (bool s, bytes memory d) =
+            address(token).staticcall{ gas: 50000 }(abi.encodeWithSelector(IERC20.balanceOf.selector, address(t
+        uint256 totalBalance = s && d.length >= 32 ? abi.decode(d, (uint256)) : 0;
         uint256 reservedBalance = totalPendingPartnerFees[token];

         // If reservations fully consume the token balance, nothing is currently distributable.
         if (totalBalance <= reservedBalance) {
             return 0;
         }

         return totalBalance - reservedBalance;
     }

     /**
      * @notice Authorize an upgrade to a new implementation.
      * @dev Only the owner can approve upgrades, as required by `UUPSUpgradeable`.
      * @param newImplementation Address of the proposed implementation.
      */
     function _authorizeUpgrade(address newImplementation) internal view override onlyOwner {
         newImplementation;
     }
 }
```

Related findings

[Low] Unbounded partner iteration in VaultFeeAllocator fee distribution causes gas DoS preventing on-chain distributions

Description

VaultFeeAllocator's fee distribution loops over an append-only partners array and attempts ERC20 transfers for each active partner. As the array grows (active or inactive entries), gas costs can exceed block limits, causing distributions to revert.

This is an admin-controlled scalability DoS, not attacker-exploitable.

VaultFeeAllocator.distributeFees and distributeFeesAmount iterate over the entire partners array and, for each active partner, calculate a share and attempt an ERC20 transfer. The partners array is append-only: addPartner pushes new entries, and removePartner only marks entries inactive without shrinking. Consequently, the loop cost scales with partners.length, and for many active partners the cumulative external transfer attempts further increase gas usage. With sufficiently large arrays (either many active partners or many historical inactive entries), distribution calls can exceed the block gas limit and revert, preventing on-chain fee distributions. While tokens are not permanently stuck (the owner can use emergencyWithdraw or upgrade via UUPS), this still breaks the intended on-chain, pro-rata distribution process and can delay partner payouts until the configuration is addressed.

Severity

Impact Explanation: [Medium] On-chain fee distribution (an important non-core function) can be effectively disabled by gas exhaustion, delaying partner payouts and breaking intended distribution flows. Funds are not permanently frozen due to emergencyWithdraw and upgradeability, so impact does not rise to high.

Likelihood Explanation: [Low] All scenarios require admin-controlled configuration (many active partners or heavy churn creating many inactive entries). Attackers cannot bloat the array or meaningfully increase likelihood; this hinges on trusted-role misconfiguration.

Exploitation

Exploitation Scenarios:

Scenario 1.

Many active partners: The owner has configured hundreds or thousands of active partners; distributeFees(token) iterates the array and attempts a transfer per active partner. The transaction exceeds block gas and reverts, preventing any on-chain distribution.

Preconditions / Assumptions

- (a). Owner configured a large number of active partners (each with basisPoints > 0) with totalAllocatedBps ≤ 10,000
- (b). VaultFeeAllocator holds a nonzero balance of the fee token to distribute
- (c). Standard block gas limits apply; no special gas subsidies or batching exist

Scenario 2.

Many inactive (historical) entries: Over time, the owner added and removed many partners, producing a large partners.length with few active entries. distributeFees(token) still scans the entire array, and the read/branch overhead leads to out-of-gas and reverts before completing distribution.

Preconditions / Assumptions

- (a). Owner previously added and removed many partners, leaving a very large partners.length with many inactive entries
- (b). Only a small subset of partners are active at distribution time
- (c). VaultFeeAllocator holds a nonzero balance of the fee token to distribute

Scenario 3.

Near 1-bps-per-partner upper bound: The owner configures close to per-bps cap (~10,000 active entries); distributeFees(token) attempts thousands of external transfers in one call, exceeding block gas and reverting.

Preconditions / Assumptions

- (a). Owner configured an extreme number of active partners approaching the per-bps cap (~10,000 active entries)
- (b). VaultFeeAllocator holds a nonzero balance of the fee token to distribute
- (c). Standard block gas limits apply

Proposed fix

VaultFeeAllocator.sol

File: `src/fee/VaultFeeAllocator.sol`

Source

```
 ... 53 unchanged lines ...
     /// @notice Append-only list of current and retired partners.
     Partner[] public partners;
+    // NOTE: partners[] is append-only; removePartner marks inactive without shrinking the array.
+    // To avoid gas blowups when iterating, maintain a compact activePartnerIndices list + mapping for iteratio

     /// @notice Sum of basis points across all active partners.
 ... 196 unchanged lines ...
      * @dev Reserved pending balances are excluded so previously failed payouts cannot be double-spent.
      * @param token Token to distribute, whether vault shares or already-realized underlying.
+    // WARNING: Iterates over partners[], which can be large due to append-only storage.
+    // Recommended: implement snapshot-based, paginated distribution rounds with a cursor over activePartnerInd
      */
     function distributeFees(address token) external nonReentrant {
 ... 37 unchanged lines ...
      * @param token Token to distribute.
      * @param amount Amount to distribute.
+    // WARNING: Same iteration pattern as distributeFees; should use the same paginated round mechanism.
      */
     function distributeFeesAmount(address token, uint256 amount) external nonReentrant onlyOwner {
 ... 314 unchanged lines ...
```

12. [Medium] Unchecked slippage parameters in Uniswap v3/v4 shMON/WMON swaps in AusdStrategy cause principal loss via MEV

Status

Review status: Unresolved
Remediation status: Unremediated
Remediation note: Created by pipeline analysis

Description

AusdStrategy's Uniswap v3/v4 shMON/WMON swap helpers forward caller-supplied minAmountOut and price limits without enforcing a strategy-level floor. When operators submit batches with zero or overly lax bounds, swaps can be sandwiched or otherwise manipulated, resulting in material value loss to the strategy and vault depositors.

The functions AusdStrategy._swapViaUniswapV3 and AusdStrategy._swapViaUniswapV4 decode ABI-encoded swap parameters and forward them directly to ShMonSwapAdapter without adding any minimum-output or price-limit floor. In ShMonSwapAdapter._swapV3, Uniswap v3's exactInputSingle is called with amountOutMinimum set to the provided minAmountOut; the adapter only rejects zero output. In ShMonSwapAdapter._swapV4, the unlockCallback enforces only that the received amount >= minAmountOut; if sqrtPriceLimitX96 is zero, the adapter resolves an extremely permissive default bound. Although only the vault can trigger these swaps (onlySelf), MEV adversaries can front-run/back-run when operators submit weak slippage parameters. The basis guard in AusdStrategy gates allocation/deallocation by comparing oracle vs implied shMON PPS but does not constrain per-trade DEX execution, so it does not mitigate sandwich risk for these swaps.

Severity

Impact Explanation: [High] Swaps can execute at materially worse-than-market prices, causing direct principal loss to strategy assets and vault depositors; magnitude scales with trade size.

Likelihood Explanation: [Low] Exploitation requires the trusted operator to supply zero or overly permissive slippage/price limits (or a wrong router), which is a trusted-role misconfiguration, though MEV conditions are common.

Exploitation

Exploitation Scenarios:

Scenario 1.

Uniswap v3 sandwich: The vault operator executes a batch including swapViaUniswapV3 with a large amountIn and minAmountOut=0 (or very small). MEV actors front-run to worsen the price, the strategy's trade clears at a poor price due to zero/lax minOut, then the attacker back-runs to restore the price, extracting value. The adapter accepts any non-zero output, causing strategy NAV loss.

Preconditions / Assumptions

- (a). A liquid shMON/WMON Uniswap v3 pool exists
- (b). Vault/operator includes swapViaUniswapV3 in a batch with large amountIn
- (c). minAmountOut is zero or overly lax; sqrtPriceLimitX96 is not constraining
- (d). Public mempool with active MEV/front-running
- (e). OnlyVault and onlySelf gates allow the batch to execute (operator-triggered)

Scenario 2.

Uniswap v4 sandwich: The vault operator executes a batch including swapViaUniswapV4 with a large amountIn, minAmountOut=0 (or very small), and sqrtPriceLimitX96=0 (permissive default). MEV actors front-run to push price adversely before the swap, the trade executes with effectively no meaningful slippage bound, and the attacker back-runs to profit. The strategy realizes a material loss.

Preconditions / Assumptions

- (a). A native MON/shMON Uniswap v4 pool exists with the trusted PoolManager
- (b). Vault/operator includes swapViaUniswapV4 in a batch with large amountIn
- (c). minAmountOut is zero or overly lax; sqrtPriceLimitX96=0 resolving to permissive bound
- (d). Public mempool with active MEV/front-running
- (e). OnlyVault and onlySelf gates allow the batch to execute (operator-triggered)

Scenario 3.

Malicious v3 router misconfiguration: The vault operator misconfigures V3SwapParams.router to a malicious address and sets minAmountOut tiny or zero. The adapter approves exactly amountIn, the malicious router transfers in the full allowance and returns dust output (>0), passing the adapter's non-zero output check. The strategy loses nearly the entire input.

Preconditions / Assumptions

- (a). Operator sets V3SwapParams.router to a non-canonical, malicious router
- (b). minAmountOut is zero or dust-level
- (c). Strategy holds sufficient input tokens and approves exactly amountIn to the router
- (d). OnlyVault and onlySelf gates allow the batch to execute (operator-triggered)

Proposed fix

AusdStrategy.sol

File: `src/strategies/AusdStrategy.sol`

Source

```
 ... 477 unchanged lines ...
     function _swapViaUniswapV3(bytes memory data) internal override {
         ShMonSwapAdapter.V3SwapParams memory params = abi.decode(data, (ShMonSwapAdapter.V3SwapParams));
+        { uint256 b = IERC20(params.tokenIn).balanceOf(address(this)); uint256 a = (params.amountIn == 0 || par
         ShMonSwapAdapter.swapViaUniswapV3(address(SHMON), address(WMON), params);
     }

     /// @dev Execute a dedicated Uniswap v4 WMON/shMON swap using the trusted PoolManager unlock flow.
     function _swapViaUniswapV4(bytes memory data) internal override {
         ShMonSwapAdapter.V4SwapParams memory params = abi.decode(data, (ShMonSwapAdapter.V4SwapParams));
+        { uint256 b = IERC20(params.tokenIn).balanceOf(address(this)); uint256 a = (params.amountIn == 0 || par
         _activateShMonSwapV4UnlockContext();
```

```
        ShMonSwapAdapter.swapViaUniswapV4(address(SHMON), address(WMON), _requireWMonUnwrapper(), params);
... 701 unchanged lines ...
```

Related findings

[Medium] Untrusted router parameter and missing minOut enforcement in ShMonSwapAdapter.swapViaUniswapV3 cause privileged principal loss

Description

The Uniswap v3 shMON/WMON swap path approves and calls a caller-supplied router and only checks for non-zero output, never re-enforcing minAmountOut. A malicious or misconfigured router can pull the full input with the temporary allowance and return dust, enabling privileged exfiltration or severe slippage. Although only the vault can trigger this path, it creates a real loss vector under operator compromise or misconfiguration.

In ShMonSwapAdapter._swapV3, the strategy resolves amountIn (using the full balance when params.amountIn == 0), force-approves params.router for tokenIn, and calls IUniswapV3SwapRouter(params.router).exactInputSingle. Afterward, it only checks that the router returned a non-zero amount and that tokenOut balance increased by a non-zero amount. It does not whitelist/pin the router address and does not re-validate that the actual balance delta meets params.minAmountOut. AusdStrategy._swapViaUniswapV3 decodes V3SwapParams directly from batch data and forwards unchanged. BaseStrategy.swapViaUniswapV3 is onlySelf and can only be reached via allocateFunds/deallocateFunds, which are onlyVault (allocateFunds). With the canonical Uniswap v3 router, minOut is enforced at the router and this path is safe; however, because the router is caller-supplied, a malicious or misconfigured router can ignore minOut, use the temporary allowance to transferFrom the full input, and return dust to pass the adapter's non-zero checks. This enables a privileged exfiltration vector and defeats slippage guarantees. In contrast, the v4 path is hard-pinned to a trusted PoolManager and enforces minOut on the actual BalanceDelta within the unlock callback.

Severity

Impact Explanation: [High] The scenarios cause direct, material loss of principal funds held by the strategy (WMON/shMON), including complete drainage of the tokenIn balance.

Likelihood Explanation: [Low] Exploitation requires trusted-role (vault/onlyVault) misuse, compromise, or operator misconfiguration of the router address; end users cannot trigger this path.

Exploitation

Exploitation Scenarios:

Scenario 1.

Compromised or malicious vault operator submits a batch that calls swapViaUniswapV3 with router set to an attacker contract. The adapter approves the router for the full tokenIn balance (amountIn=0). The attacker router pulls all tokenIn via transferFrom during exactInputSingle and sends back 1 unit of tokenOut, returning 1 as amountOut. The adapter's non-zero checks pass and funds are siphoned to the attacker.

Preconditions / Assumptions

- (a). Vault (onlyVault) control is malicious or compromised (able to execute allocateFunds/deallocateFunds batches).
- (b). Strategy holds a non-zero balance of WMON or shMON.
- (c). Attacker deploys a router-like contract implementing exactInputSingle that can transferFrom tokenIn and send dust tokenOut.
- (d). Pair is shMON <-> WMON (validated by the adapter).

Scenario 2.

An honest operator misconfigures the router address to a non-canonical router that does not enforce amountOutMinimum. The batch sets a high minAmountOut expecting slippage protection, but the non-canonical router performs a poor swap or ignores minOut, returning dust. The adapter accepts the result because it only checks for non-zero outputs, causing severe slippage loss to the strategy.

Preconditions / Assumptions

- (a). Vault (onlyVault) operator is honest but misconfigures the router address to a non-canonical implementation.
- (b). Strategy holds a non-zero balance of WMON or shMON.
- (c). The non-canonical router ignores or misapplies amountOutMinimum and returns non-zero but insufficient output.
- (d). Pair is shMON <-> WMON.

Scenario 3.

A vault-only adversary uses swapViaUniswapV3 with a malicious router to exfiltrate strategy-held WMON/shMON to arbitrary EOAs during the router call, bypassing destination constraints present in other commands (e.g., returnIdleToVault, withdrawToOwner), even if the strategy admin itself is uncompromised.

Preconditions / Assumptions

- (a). Vault (onlyVault) privileges are compromised or malicious while the strategy admin may remain uncompromised.
- (b). Strategy holds a non-zero balance of WMON or shMON.
- (c). Attacker supplies a malicious router that can transferFrom during exactInputSingle and send dust tokenOut.
- (d). Pair is shMON <-> WMON.

Proposed fix

ShMonSwapAdapter.sol

File: `src/common/ShMonSwapAdapter.sol`

Source

```
 ... 205 unchanged lines ...
         uint256 actualAmountOut = IERC20(params.tokenOut).balanceOf(address(this)) - outputBalanceBefore;
         if (amountOut == 0 || actualAmountOut == 0) revert ZeroSwapOutput(params.tokenIn, params.tokenOut);
+        // Enforce caller's slippage bound against the actual balance delta so arbitrary/misconfigured routers
+        // cannot bypass minAmountOut and exfiltrate via temporary allowance.
+        if (actualAmountOut < params.minAmountOut) {
+            revert InsufficientShMonSwapOutput(actualAmountOut, params.minAmountOut);
+        }
     }

 ... 110 unchanged lines ...
```

[Medium] Missing input-cap with hook-enabled pools in ShMonSwapAdapter v4 causes over-consumption of input balances

Description

ShMonSwapAdapter's Uniswap v4 shMON/WMON swap path accepts hook-enabled pools and pays the swap's settled input derived from BalanceDelta without enforcing settleAmount <= amountIn. A beforeSwap hook can increase the specified-currency delta, causing the strategy to pay more than the requested exact input by consuming unrelated idle MON or shMON, or otherwise to revert unexpectedly.

In ShMonSwapAdapter's Uniswap v4 path, _validatePool only checks that pool currencies match shMON and MON, allowing nonzero hooks and arbitrary hookData. During unlockCallback, the adapter executes an exact-input style swap (amountSpecified = -amountIn), then computes settleAmount directly from the returned BalanceDelta and pays it. There is no check that settleAmount <= amountIn. Under canonical Uniswap v4 semantics, a beforeSwap hook can return a positive specified-currency delta, making the net owed input exceed amountIn. If tokenIn is WMON, the adapter unwraps exactly amountIn to native MON first, then calls settle{value: settleAmount}; if settleAmount is larger and the strategy holds idle native MON, it is over-consumed. If tokenIn is shMON (ERC20), the adapter syncs and transfers settleAmount of shMON; if extra idle shMON exists, it is over-consumed. Otherwise, the swap reverts unexpectedly. The output is only floor-bounded by minAmountOut, so excess input can be extracted as unintended fees when a hook-enabled pool is used. Only the vault/operator can trigger this path and choose the poolKey/hookData, making this an integration-hardening issue rather than a public attacker exploit.

Severity

Impact Explanation: [High] Excess input beyond amountIn can be transferred from the strategy when hooks are enabled and idle balances exist, resulting in direct, material loss of principal funds.

Likelihood Explanation: [Low] Exploitation requires a trusted operator to configure or select a hook-enabled pool and the strategy to have specific idle balances at execution time; end users cannot trigger the path.

Exploitation

Exploitation Scenarios:

Scenario 1.

Over-consumption of idle native MON on a WMON->shMON v4 swap: operator selects a hook-enabled MON/shMON pool; the hook increases the specified-currency delta so settleAmount > amountIn; the adapter unwraps amountIn WMON to MON then pays settle{value: settleAmount}, consuming additional idle MON and transferring more than intended while output meets minAmountOut.

Preconditions / Assumptions

- (a). Only the vault/operator can execute swapViaUniswapV4 and supply poolKey/hookData
- (b). A Uniswap v4 MON/shMON pool with a nonzero hooks address is selected
- (c). The hook returns a positive specified-currency delta (per canonical v4 semantics)
- (d). The strategy holds idle native MON beyond the freshly unwrapped amountIn
- (e). minAmountOut is satisfied for the swap output

Scenario 2.

Over-consumption of idle shMON on a shMON->WMON v4 swap: operator selects a hook-enabled MON/shMON pool; the hook increases the specified-currency delta so settleAmount > amountIn; the adapter transfers settleAmount of shMON (after sync), consuming extra idle shMON beyond amountIn while output meets minAmountOut.

Preconditions / Assumptions

- (a). Only the vault/operator can execute swapViaUniswapV4 and supply poolKey/hookData
- (b). A Uniswap v4 MON/shMON pool with a nonzero hooks address is selected
- (c). The hook returns a positive specified-currency delta (per canonical v4 semantics)
- (d). The strategy holds idle shMON beyond amountIn
- (e). minAmountOut is satisfied for the swap output

Scenario 3.

Unexpected revert (DoS) when no spare idle balances exist: operator uses a hook-enabled pool; the hook increases specified-currency delta so settleAmount > amountIn; with no extra idle MON (WMON input) or shMON (shMON input), the settle/transfer fails and the batch reverts.

Preconditions / Assumptions

- (a). Only the vault/operator can execute swapViaUniswapV4 and supply poolKey/hookData
- (b). A Uniswap v4 MON/shMON pool with a nonzero hooks address is selected
- (c). The hook returns a positive specified-currency delta (per canonical v4 semantics)
- (d). The strategy does not hold idle native MON (for WMON input) or idle shMON (for shMON input)

Proposed fix

ShMonSwapAdapter.sol

File: `src/common/ShMonSwapAdapter.sol`

Source

```
 ... 67 unchanged lines ...
     /// @param currency1 Actual `currency1` address resolved from the pool key.
     error InvalidShMonSwapPool(address currency0, address currency1);
+    error UnexpectedShMonSwapHooks(address hooks);
     /// @notice Revert when the supplied token pair is not the supported shMON/WMON market.
     /// @param tokenIn Requested input token.
 ... 206 unchanged lines ...
```

```
        pure
    {
+       if (address(poolKey.hooks) != address(0)) revert UnexpectedShMonSwapHooks(address(poolKey.hooks));
+
        address currency0 = IUniswapV4.Currency.unwrap(poolKey.currency0);
        address currency1 = IUniswapV4.Currency.unwrap(poolKey.currency1);
 ... 38 unchanged lines ...
```

13. [Medium] Unchecked forceRemoveUnstakeRequest in AusdStrategy causes share-pricing manipulation and LP dilution

Status

Review status: Unresolved
Remediation status: Unremediated
Remediation note: Created by pipeline analysis

Description

AusdStrategy lets STRATEGY_ADMIN remove active pending-unstake value from NAV without checks. Vault accrual then books a loss (lower PPS), allowing mispriced deposits; later recovery books a gain that can mint performance fees, turning a liveness tool into a privileged share-pricing lever.

Pending-unstake MON is explicitly included in strategy NAV via totalPendingUnstakeMon. forceRemoveUnstakeRequest deducts a request's expectedAmountMon from this total and marks it inactive without any maturity/liveness/economic-loss checks, immediately lowering the strategy's totalAllocatedValue. The next vault entrypoint that runs withYieldAccrual books this as negative yield into cachedTotalAssets, reducing PPS and minting more shares per asset for deposits. Later, recoverForceRemovedUnstakeRequest returns the MON to the strategy as idle; a subsequent accrual recognizes positive yield and can mint performance fees on this rebound. When deposits are open, STRATEGY_ADMIN can self-deposit at the depressed PPS (diluting existing LPs); when deposits are gated, collusion with WITHDRAWAL_MANAGER can execute Permit2 deposits at the dip. All actions are logged but not constrained by checks, making this a privileged, discretionary share-pricing and fee-minting control surface rather than a pure liveness tool.

Severity

Impact Explanation: [High] Admin-induced PPS depression enables mispriced share minting that dilutes existing LPs (material principal loss), and later rebound can mint performance fees (material loss of yield/fees).

Likelihood Explanation: [Low] Exploitation requires misuse or collusion of trusted roles (STRATEGY_ADMIN, and WITHDRAWAL_MANAGER when gated) plus operational setup to create sizable pending-unstake value.

Exploitation

Exploitation Scenarios:

Scenario 1.

Open deposits: STRATEGY_ADMIN builds a large pending-unstake pool, force-removes an active request to depress NAV, immediately deposits via ERC-4626 at the lower PPS (diluting existing LPs), then later recovers the request and triggers accrual to mint performance fees on the rebound.

Preconditions / Assumptions

- (a). Deposits are not gated (public ERC-4626 deposit/mint enabled)
- (b). Performance fee configured (>0)
- (c). Sufficient totalPendingUnstakeMon exists (or can be created) to move NAV
- (d). STRATEGY_ADMIN controls AusdStrategy and can call forceRemoveUnstakeRequest/recoverForceRemovedUnstakeRequest
- (e). withYieldAccrual runs on deposit (standard ERC-4626 semantics)

Scenario 2.

Fee rebound without deposit timing: STRATEGY_ADMIN force-removes to book a loss, triggers accrual, then later recovers the request and triggers accrual again to book positive yield; performance fees mint on the rebound despite no net economic gain across both accruals.

Preconditions / Assumptions

- (a). Performance fee configured (>0)
- (b). STRATEGY_ADMIN can force-remove and later recover pending unstake
- (c). Sufficient totalPendingUnstakeMon exists to move NAV
- (d). Anyone can call accrueYield to book loss and later gain
- (e). Fee recipient is aligned with admin interests or otherwise benefits

Scenario 3.

Gated deposits with collusion: With deposits gated, STRATEGY_ADMIN force-removes to depress NAV and WITHDRAWAL_MANAGER executes Permit2 deposits (using low-minShares signatures or self-signed) at the dip; later recovery triggers performance-fee minting on the rebound, harming existing LPs.

Preconditions / Assumptions

- (a). Deposits are gated (standard ERC-4626 deposit/mint disabled)
- (b). WITHDRAWAL_MANAGER is cooperative or controlled by the same operator
- (c). Permit2 deposit signatures with low minShares exist or are self-signed
- (d). Performance fee configured (>0)
- (e). Sufficient totalPendingUnstakeMon exists to move NAV
- (f). STRATEGY_ADMIN can force-remove and recover

Proposed fix

StrategyStorageLib.sol

File: `src/libraries/StrategyStorageLib.sol`

Source

```
 ... 95 unchanged lines ...
         /// @notice Aggregate pending MON amount across all active unstake requests.
         uint256 totalPendingUnstakeMon;
+        uint256 totalSuspendedUnstakeMon;
         /// @notice Dense list of currently active unstake request ids.
         uint256[] activeUnstakeRequestIds;
         /// @notice Request metadata keyed by unstake request id.
         mapping(uint256 => UnstakeRequest) unstakeRequests;
     }

     /// @notice Load the AUSD strategy storage pointer.
     /// @return $ Storage pointer rooted at the ERC-7201 namespace.
     /// @dev This mirrors `BaseStrategyStorageLib.fetch` but targets the AUSD-only namespace.
     function fetch() internal pure returns (AusdStrategyStorage storage $) {
         bytes32 slot = STORAGE_LOCATION;
         assembly {
             // Point the returned storage reference at the fixed ERC-7201 namespace slot.
             $.slot := slot
         }
     }
 }
```

AusdStrategy.sol

File: `src/strategies/AusdStrategy.sol`

Source

```
 ... 764 unchanged lines ...
             total += _wmonToUsd(pendingMon);
         }
+        uint256 suspendedMon = AusdStrategyStorageLib.fetch().totalSuspendedUnstakeMon;
+        if (suspendedMon > 0) {
+            total += _wmonToUsd(suspendedMon);
+        }
     }

 ... 109 unchanged lines ...
         uint256 index = _findActiveUnstakeRequestIndex($, requestId);
         $.totalPendingUnstakeMon -= request.expectedAmountMon;
+        $.totalSuspendedUnstakeMon += request.expectedAmountMon;
         request.active = false;
         _removeActiveUnstakeRequestAtIndex($, index);

         emit ShMonUnstakeForceRemoved(requestId, request.worker, request.expectedAmountMon);
     }

     /**
      * @notice Complete a previously force-removed worker if its underlying request later becomes claimable.
      * @param requestId Strategy-local unstake request id that was already marked inactive.
      * @return amountReceived MON amount forwarded back from the worker.
      */
     function recoverForceRemovedUnstakeRequest(uint256 requestId)
         external
         onlyStrategyAdmin
         returns (uint256 amountReceived)
     {
         AusdStrategyStorageLib.UnstakeRequest storage request =
             AusdStrategyStorageLib.fetch().unstakeRequests[requestId];
         if (request.active) revert ActiveUnstakeRequest(requestId);
         if (request.worker == address(0)) revert InactiveUnstakeRequest(requestId);

         amountReceived = AusdStrategyUnstakeWorker(payable(request.worker)).completeUnstake();
+        AusdStrategyStorageLib.AusdStrategyStorage storage $ = AusdStrategyStorageLib.fetch();
+        uint256 dec = amountReceived < request.expectedAmountMon ? amountReceived : request.expectedAmountMon;
+        if (dec > 0) { $.totalSuspendedUnstakeMon -= dec; }
         emit ShMonUnstakeRecovered(requestId, request.worker, amountReceived);
     }
 ... 285 unchanged lines ...
```

Related findings

[Medium] Pending shMON unstake face-value accounting in AusdStrategy causes time-dependent NAV overstatement and unfair pricing

Description

AusdStrategy counts pending shMON unstakes at full face value (stored "locked" amount) in NAV without mid-flight refresh. On completion it tolerates receiving less MON. This can overstate NAV until completion, enabling unfair withdrawal pricing or fee accrual, with the shortfall later borne by remaining LPs.

When requesting an async shMON unstake, AusdStrategy records the ShMonad-reported locked MON amount in request.expectedAmountMon and increments totalPendingUnstakeMon. NAV then includes this pendingMon valued at the WMON oracle. The strategy never refreshes these amounts while pending; only on completion does it subtract the stored expectedAmountMon and add the actually received MON to idle balances. If ShMonad’s effective payout is reduced before completion (e.g., haircut, liability rebalancing), NAV remains overstated during the waiting window. Vault processes (withdrawal epoch pricing, fee accrual) that rely on the strategy’s current value can overpay exiting users or over-collect fees, pushing the realized loss onto remaining LPs when completion occurs. MAX_ACTIVE_UNSTAKE_REQUESTS=7 keeps worker count small but the code still does not re-read current locked amounts or discount pendingMon. Completion is operator-driven, so the window exists until finalize is called.

Severity

Impact Explanation: [High] Exiting users can be overpaid and remaining LPs absorb a direct, material loss of principal when the shortfall is realized; fee accrual can also be overstated, reducing LP returns.

Likelihood Explanation: [Low] Requires uncommon external conditions (haircut or payout reduction) and timing of vault snapshots relative to operator-driven completion; multiple preconditions outside attacker control reduce frequency.

Exploitation

Exploitation Scenarios:

Scenario 1.

Async withdrawal epoch overpayment: A haircut reduces the pending unstake payout below the stored expected amount. The vault snapshots withdrawal pricing before completion, using NAV that counts pendingMon at face value. Exiting users are overpaid; later completion realizes the shortfall, harming remaining LPs.

Preconditions / Assumptions

- (a). Strategy has nonzero pending unstakes (totalPendingUnstakeMon > 0)
- (b). ShMonad can reduce the effective payout below the previously recorded locked amount before completion
- (c). Vault snapshot/epoch pricing includes strategy NAV with pendingMon at face value
- (d). Operator has not completed the pending unstake before the snapshot

Scenario 2.

Performance fees on overstated NAV: The operator requests an unstake so pendingMon inflates NAV, then delays completion until after a fee snapshot/accrual. Fees are taken on the inflated base; completion later realizes the shortfall, harming LPs' net returns.

Preconditions / Assumptions

- (a). Vault or fee module accrues performance fees based on strategy-reported NAV before receivables are realized
- (b). Strategy has nonzero pending unstakes in NAV
- (c). Operator delays completion past a fee snapshot
- (d). A material reduction in payout occurs before completion (otherwise no harm even if delayed)

Scenario 3.

Matured-but-not-finalized window: A pending unstake has matured but is not yet completed. If an adjustment reduces the effective payout, a withdrawal pricing snapshot in this window still includes pendingMon at face value, overpaying exits. The shortfall is realized when the operator completes the unstake.

Preconditions / Assumptions

- (a). A pending unstake is matured but not yet completed
- (b). A reduction in effective payout occurs at or before completion
- (c). A withdrawal pricing snapshot occurs in the matured-but-not-finalized window

Proposed fix

AusdStrategy.sol

File: `src/strategies/AusdStrategy.sol`

Source

```
 ... 760 unchanged lines ...
         total += _stablecoinToUsd(STABLECOIN.balanceOf(address(this)));

-        uint256 pendingMon = AusdStrategyStorageLib.fetch().totalPendingUnstakeMon;
-        if (pendingMon > 0) {
-            total += _wmonToUsd(pendingMon);
```

```
-         }
+         AusdStrategyStorageLib.AusdStrategyStorage storage $ = AusdStrategyStorageLib.fetch();
+         uint256 p; uint256 l = $.activeUnstakeRequestIds.length;
+         for (uint256 i; i < l; ++i) { AusdStrategyStorageLib.UnstakeRequest storage r = $.unstakeRequests[$.act
+         if (p > 0) total += _wmonToUsd(p);
      }

 ... 420 unchanged lines ...
```

[Medium] MON-based bucket prediction using previewUnstake in AusdStrategy requestUnstake classifier causes async-unstake DoS via worker-duplication and cap exhaustion

Description

AusdStrategy classifies shMON unstake requests into base/extended buckets using previewUnstake(shares) to estimate MON and predict the completion epoch. If ShMon's oversize gating uses a different measure (e.g., shares or gross MON), predictions can be wrong, leading to duplicate workers per true epoch, eventual exhaustion of the 7-worker cap (blocking new async requests), and frequent top-up reverts.

AusdStrategy._requestUnstakeShMon computes expectedAmountMon = IShMon.previewUnstake(shares) and predicts a fresh completion epoch via _quoteFreshStandaloneUnstakeCompletionEpoch(expectedAmountMon). It then attempts to top up an existing worker in that predicted bucket or creates a new worker. However, the actual completion epoch is determined by ShMon's requestUnstake(shares), which may key its +2-epoch oversize rule to a measure different from previewUnstake(shares) (e.g., shares or gross amounts with different rounding/fees). When predictions are wrong, top-ups revert with UnexpectedTopUpCompletionEpoch, or new workers are created whose actual epoch equals an existing worker's epoch, causing duplicate workers. Repeating this can fill the hard cap of MAX_ACTIVE_UNSTAKE_REQUESTS = 7, which reverts further async requests with TooManyActiveUnstakeRequests, effectively DoS-ing new async unstakes until existing workers mature or admins force-remove requests. Using forceRemoveUnstakeRequest to recover liveness temporarily reduces reported pending value (totalPendingUnstakeMon) until manual recovery via recoverForceRemovedUnstakeRequest.

Severity

Impact Explanation: [Medium] Significant but temporary availability loss/DoS of an important strategy function (asynchronous unstake requests) due to worker-duplication and cap exhaustion. Workarounds (immediate redemption/swaps, admin force-removal) exist, so not a total break or principal loss.

Likelihood Explanation: [Medium] Requires a realistic mismatch between ShMon's oversize gating measure and previewUnstake(shares) (e.g., shares-based gating or net-vs-gross differences) and requests near capacity; plausible during operational unwinds and not dependent on attacker behavior.

Exploitation

Exploitation Scenarios:

Scenario 1.

Duplicate workers exhaust the 7-worker cap: The strategy predicts a base bucket using expected MON from previewUnstake(shares), but ShMon returns an extended epoch keyed to shares. Each request creates a new extended-epoch worker instead of aggregating, eventually filling the 7-worker cap and blocking new async unstakes until maturities or admin intervention.

Preconditions / Assumptions

- (a). ShMon's oversize gating for +2 epochs differs from previewUnstake-based MON amount (e.g., keyed to shares or gross values).
- (b). Requests are sized near the capacity threshold so misclassification occurs.
- (c). Multiple async requests are made before earlier ones mature, allowing duplicate-epoch workers to accumulate.

Scenario 2.

Top-up reverts repeatedly: The strategy predicts an epoch (e.g., extended) by expected MON and finds a matching worker to top up. ShMon returns a different actual epoch for the added shares (e.g., base), causing

UnexpectedTopUpCompletionEpoch and batch failure until the operator adapts (e.g., avoid top-ups).

Preconditions / Assumptions

- (a). ShMon's epoch gating differs from previewUnstake(shares) semantics (shares vs MON or gross vs net).
- (b). A top-up is attempted into a worker matched by predicted epoch, but actual epoch returned by ShMon for added shares differs.

Scenario 3.

Force-remove undervaluation: To recover from cap exhaustion, admin calls forceRemoveUnstakeRequest to reduce active requests, which decreases totalPendingUnstakeMon and temporarily understates NAV until recoverForceRemovedUnstakeRequest later claims the proceeds.

Preconditions / Assumptions

- (a). Admin chooses to call forceRemoveUnstakeRequest to free active slots after duplicates accumulated.
- (b). Underlying ShMon request later becomes completable and is claimed via recoverForceRemovedUnstakeRequest.

Proposed fix

AusdStrategy.sol

File: `src/strategies/AusdStrategy.sol`

Source

```
 ... 616 unchanged lines ...
         _finalizeMaturedUnstakes();
         AusdStrategyStorageLib.AusdStrategyStorage storage $ = AusdStrategyStorageLib.fetch();
-        uint64 standaloneCompletionEpoch = _quoteFreshStandaloneUnstakeCompletionEpoch(expectedAmountMon);
-        (bool foundMatchingRequest, uint256 matchingRequestId) =
-            _findMatchingActiveUnstakeRequestId($, standaloneCompletionEpoch);
-
-        if (foundMatchingRequest) {
+        if ($.activeUnstakeRequestIds.length > 0) {
+            uint256 matchingRequestId = $.activeUnstakeRequestIds[0];
             AusdStrategyStorageLib.UnstakeRequest storage request = $.unstakeRequests[matchingRequestId];
             uint256 previousExpectedAmountMon = request.expectedAmountMon;

             IERC20(address(SHMON)).safeTransfer(request.worker, toRequest);
             uint64 actualCompletionEpoch = AusdStrategyUnstakeWorker(payable(request.worker)).requestUnstake(to
-            if (actualCompletionEpoch != request.completionEpoch) {
+            if (actualCompletionEpoch < request.completionEpoch) {
                 revert UnexpectedTopUpCompletionEpoch(matchingRequestId, request.completionEpoch, actualComplet
             }
+            if (actualCompletionEpoch > request.completionEpoch) {
+                request.completionEpoch = actualCompletionEpoch;
+            }

             (uint128 updatedLockedAmountMon,) = IShMon(address(SHMON)).getUnstakeRequest(request.worker);
 ... 555 unchanged lines ...
```

[Low] Matured-but-failing unstake workers counted in cap in AusdStrategy async-unstake flow causes temporary liveness/DoS on new requests and termination

Description

Matured shMON unstake workers that fail to complete remain active and still count toward the fixed 7-worker cap. Because new requests only bucket into currentEpoch+5 or +7 and matured workers cannot be topped up, this can block creation of needed new workers and also prevent full termination until operators intervene.

In AusdStrategy, _finalizeMaturedUnstakes() attempts to complete all matured async shMON unstake workers. If a completion reverts, the worker remains active and continues to occupy a slot in activeUnstakeRequestIds. In

_requestUnstakeShMon(), after attempting to finalize matured workers, the strategy computes the fresh standalone completion epoch (only currentEpoch+5 or +7) and either tops up a matching active worker or creates a new one. If no matching bucket exists and activeUnstakeRequestIds.length is already 7 (including any matured-but-failing worker), the function reverts with TooManyActiveUnstakeRequests. Additionally, _terminatePositions() refuses to proceed if any async requests remain active, so a matured-but-failing worker can also block full termination. While this is an intentional design tradeoff (with admin forceRemove and later recover hooks), it creates a temporary liveness/DoS risk for the async path and termination until operators act or retries succeed.

Severity

Impact Explanation: [Low] Temporary, operator-resolvable unavailability of the async-unstake path and termination; no direct principal loss; multiple workarounds exist (top-up +5, immediate redemption, swaps, admin forceRemove and later recover).

Likelihood Explanation: [Low] Requires uncommon internal states (cap fully used, at least one matured-but-failing worker, specific bucket distribution and amount sizing). Operators are expected to be competent and can mitigate promptly.

Exploitation

Exploitation Scenarios:

Scenario 1.

Cap blocks opening a needed +7 bucket during unwind: With 7 active requests (including 1 matured-but-failing worker) and no existing +7 bucket, a new large unstake request that buckets into currentEpoch+7 cannot create a new worker and reverts with TooManyActiveUnstakeRequests, halting that deallocation batch.

Preconditions / Assumptions

- (a). activeUnstakeRequestIds.length == 7
- (b). At least one worker is matured (completionEpoch <= currentEpoch) but completeUnstake() currently fails
- (c). No active worker for completionEpoch == currentEpoch+7 exists
- (d). A new request amount that triggers standalone completion at currentEpoch+7

Scenario 2.

Full termination blocked by a matured-but-failing worker: A matured worker repeatedly fails completeUnstake(), remains active, and _assertNoPendingShMonadUnstakes() reverts termination with TerminationPendingShMonadUnstake until operators forceRemove or a later retry succeeds.

Preconditions / Assumptions

- (a). At least one matured worker fails completeUnstake() and remains active
- (b). Operator invokes terminatePositions before force-removing or successfully completing the worker

Scenario 3.

Multiple matured completions fail; async path effectively frozen: Several matured workers fail to complete, consuming the 7-worker cap and preventing creation of new +5/+7 buckets as needed. Async-unstake progress halts until operators free capacity (forceRemove) or completions later succeed.

Preconditions / Assumptions

- (a). Multiple matured workers fail completeUnstake() and remain active
- (b). 7-worker cap consumed by matured-but-failing (and/or non-matching) buckets
- (c). Need to open new +5/+7 buckets to continue async unwind

Proposed fix

AusdStrategy.sol

File: `src/strategies/AusdStrategy.sol`

Source

```
 ... 649 unchanged lines ...
         }

-        if ($.activeUnstakeRequestIds.length >= MAX_ACTIVE_UNSTAKE_REQUESTS) {
-            revert TooManyActiveUnstakeRequests($.activeUnstakeRequestIds.length);
+        // Enforce cap against future-bucket workers only; ignore matured-but-failing ones.
+        uint64 currentEpoch = IShMon(address(SHMON)).getInternalEpoch();
+        uint256 futureActiveCount;
+        uint256 idsLen = $.activeUnstakeRequestIds.length;
+        for (uint256 i = 0; i < idsLen; ++i) {
+            uint256 id = $.activeUnstakeRequestIds[i];
+            AusdStrategyStorageLib.UnstakeRequest storage r = $.unstakeRequests[id];
+            if (r.active && r.completionEpoch > currentEpoch) {
+                unchecked { ++futureActiveCount; }
+            }
+        }
+        if (futureActiveCount >= MAX_ACTIVE_UNSTAKE_REQUESTS) revert TooManyActiveUnstakeRequests(futureActiveC

         AusdStrategyUnstakeWorker worker = new AusdStrategyUnstakeWorker(address(this), address(SHMON));
 ... 532 unchanged lines ...
```

[Informational] Unconditional external epoch read in AusdStrategy._finalizeMaturedUnstakes causes avoidable liveness failures in empty-state flows

Description

AusdStrategy._finalizeMaturedUnstakes() unconditionally calls ShMon.getInternalEpoch() before checking for active unstake requests. If the external ShMon view reverts, calls to completeUnstakeShMon(), requestUnstakeShMon() (preemptive sweep), and terminatePositions() can revert even when there are no active requests, creating unnecessary liveness fragility. An early return when no active requests exist would remove this dependency.

Inside AusdStrategy._finalizeMaturedUnstakes(), the first operation is an external call to IShMon(address(SHMON)).getInternalEpoch() before reading the activeUnstakeRequestIds array length. As a result, even when there are zero active requests, this function will revert if the ShMon epoch view fails. Because _finalizeMaturedUnstakes() is unconditionally invoked by completeUnstakeShMon(), requestUnstakeShMon() (as a preemptive sweep), and terminatePositions() (before asserting no pending requests), those calls can all fail in the empty state solely due to an external read. This widens the external liveness dependency unnecessarily and blocks the convenience termination path in an edge case. The issue does not cause loss of funds, and manual unwind using granular commands remains possible. A trivial fix is to early-return when activeUnstakeRequestIds.length == 0 before calling getInternalEpoch().

Severity

Impact Explanation: [Informational] No loss of funds or core functionality; only convenience/UX and minor liveness friction in empty-state paths. Manual unwind via granular commands remains available.

Likelihood Explanation: [Low] Requires an external integration (ShMon) view to fail; such outages are plausible but not expected under normal operation.

Exploitation

Exploitation Scenarios:

Scenario 1.

Termination blocked in empty state: With zero active unstake requests, terminatePositions() reverts because _finalizeMaturedUnstakes() calls ShMon.getInternalEpoch() and it fails, even though there is nothing to finalize.

Preconditions / Assumptions

- (a). No active unstake requests (activeUnstakeRequestIds.length == 0 and totalPendingUnstakeMon == 0)
- (b). ShMon.getInternalEpoch() reverts

- (c). Admin invokes terminatePositions()

Scenario 2.

Housekeeping completion call fails in empty state: With zero active unstake requests, completeUnstakeShMon() reverts due to the unconditional epoch read in _finalizeMaturedUnstakes(), despite having no work to do.

Preconditions / Assumptions

- (a). No active unstake requests (activeUnstakeRequestIds.length == 0)
- (b). ShMon.getInternalEpoch() reverts
- (c). Operator/automation invokes completeUnstakeShMon()

Scenario 3.

Preemptive finalize causes earlier-than-necessary revert when opening a new request: requestUnstakeShMon() calls _finalizeMaturedUnstakes() first; if ShMon.getInternalEpoch() reverts (even if previewUnstake succeeds), the call fails earlier than necessary.

Preconditions / Assumptions

- (a). No active unstake requests (activeUnstakeRequestIds.length == 0)
- (b). ShMon.getInternalEpoch() reverts (previewUnstake may still succeed)
- (c). Operator invokes requestUnstakeShMon(shares)

Proposed fix

AusdStrategy.sol

File: `src/strategies/AusdStrategy.sol`

Source

```
 ... 1015 unchanged lines ...
     function _finalizeMaturedUnstakes() internal {
         AusdStrategyStorageLib.AusdStrategyStorage storage $ = AusdStrategyStorageLib.fetch();
+        uint256 length = $.activeUnstakeRequestIds.length;
+        if (length == 0) return;
         uint64 currentEpoch = IShMon(address(SHMON)).getInternalEpoch();

         uint256 i = 0;
-        uint256 length = $.activeUnstakeRequestIds.length;
         while (i < length) {
             uint256 requestId = $.activeUnstakeRequestIds[i];
 ... 165 unchanged lines ...
```

[Low] Strict reserved-amount increase check with no fallback in AusdStrategy._requestUnstakeShMon top-ups causes liveness failures and NAV overstatement

Description

AusdStrategy’s top-up path for existing shMON unstake workers reverts unless ShMon’s live reserved MON strictly increases and the epoch stays unchanged, with no fallback to create a new worker. If the live reserved amount decreases or remains unchanged before completion, or aggregation changes the epoch, top-ups revert and accounting can remain stale, leading to liveness issues and potential NAV overstatement until resolution.

When _requestUnstakeShMon finds a matching completion-epoch bucket, it transfers shares to the worker, requests an unstake, and then requires (1) the actual completionEpoch equals the stored epoch and (2) the live reserved MON (IShMon.getUnstakeRequest(worker).amountMon) strictly exceeds the previously stored expectedAmountMon. If either condition fails, the call reverts and there is no fallback to open a new worker in this branch. The strategy also does not reconcile expectedAmountMon or totalPendingUnstakeMon downward if ShMon’s live reserved amount has changed since the snapshot. As a result, small or rounding-no-op top-ups, epoch changes on aggregation, or any mid-flight downward adjustments to reserved amounts can cause repeated reverts, prevent top-ups into the existing bucket, and leave pending accounting stale, overstating NAV until the request completes or is force-removed.

Severity

Impact Explanation: [Medium] Top-ups into existing buckets can be blocked and NAV can be overstated until completion or force-removal. This is a significant but temporary availability and accuracy issue; there is no direct principal loss and operator/admin workarounds exist.

Likelihood Explanation: [Low] The most impactful scenario requires conditions not guaranteed in all deployments (e.g., per-account reserved amount decreasing or not increasing enough mid-flight). While rounding and threshold-crossing are plausible, combining conditions that produce meaningful impact is less frequent.

Exploitation

Exploitation Scenarios:

Scenario 1.

Mid-flight decrease in the worker's live reserved MON below the stored snapshot: a moderate top-up fails to push the live reserved amount above the stale snapshot, causing the top-up to revert with no fallback. The matching bucket remains hard to top up until an oversized amount is used or the request is force-removed. Meanwhile, totalPendingUnstakeMon remains overstated, inflating NAV until completion/removal.

Preconditions / Assumptions

- (a). An active unstake worker exists with stored expectedAmountMon = X and completionEpoch = E
- (b). ShMon's live reserved amount for that worker has drifted to Y < X before completion (or does not increase enough on a modest top-up to exceed X)
- (c). Operator attempts a top-up whose standalone quote still maps to E (matching-bucket path is chosen)
- (d). The code reads updatedLockedAmountMon <= X after the top-up, triggering a revert and leaving accounting at X

Scenario 2.

Small top-up results in no net increase (rounding/no-op) so updatedLockedAmountMon equals the previous stored value and the call reverts. Batches containing such top-ups fail, and operators must use larger increments to proceed.

Preconditions / Assumptions

- (a). An active unstake worker exists with stored expectedAmountMon = X and completionEpoch = E
- (b). Operator submits a small top-up amount
- (c). ShMon's accounting rounds such that updatedLockedAmountMon == X after the request
- (d). The strict updated > previous check triggers a revert, aborting the batch

Scenario 3.

Aggregation on top-up changes the completion epoch so actualCompletionEpoch differs from the stored epoch, triggering a revert. There is no fallback to create a new worker for the new epoch from this path, forcing operators to retry with a different amount.

Preconditions / Assumptions

- (a). An active unstake worker exists with stored completionEpoch = E
- (b). A toRequest amount has a standalone quote for E but, when aggregated with existing pending, ShMon assigns a different epoch E'
- (c). The code detects actualCompletionEpoch != stored epoch and reverts without falling back to a new worker

Proposed fix

AusdStrategy.sol

File: `src/strategies/AusdStrategy.sol`

Source

```
 ... 631 unchanged lines ...

             (uint128 updatedLockedAmountMon,) = IShMon(address(SHMON)).getUnstakeRequest(request.worker);
-            if (updatedLockedAmountMon <= previousExpectedAmountMon) revert ZeroExpectedShMonUnstake(toRequest)
-
             uint256 newExpectedAmountMon = uint256(updatedLockedAmountMon);
+            if (newExpectedAmountMon <= previousExpectedAmountMon) {
+                // No-op or downward adjustment: keep accounting in sync and avoid revert.
+                if (newExpectedAmountMon < previousExpectedAmountMon) {
+                    $.totalPendingUnstakeMon -= (previousExpectedAmountMon - newExpectedAmountMon);
+                    request.expectedAmountMon = newExpectedAmountMon;
+                }
+                return;
+            }
             uint256 addedAmountMon = newExpectedAmountMon - previousExpectedAmountMon;
             request.expectedAmountMon = newExpectedAmountMon;
 ... 550 unchanged lines ...
```

14. [Medium] Unverified PoolManager callback data in Swapper.unlockCallback causes redirected reward proceeds and unintended MON overspend

Status

Review status: Unresolved
Remediation status: Unremediated
Remediation note: Created by pipeline analysis

Description

Swapper.unlockCallback trusts recipient and monAmount from PoolManager-provided callback bytes without independent verification, enabling redirected stable-asset proceeds or inflated MON input if the PoolManager deviates from canonical behavior.

During reward swaps to the vault asset, Swapper.processRewards unwraps WMON to MON, builds UnlockCallbackData (including recipient and monAmount), and calls PoolManager.unlock. RewardsAdapter gates the callback but does not bind the payload. Swapper.unlockCallback decodes the callback data, uses monAmount as the exact-input for swap, derives the negative MON delta to settle, and then calls take to the decoded recipient. It does not verify the callback data matches what was sent nor check settle()’s returned value. If the PoolManager (or a wrapper/proxy) alters the callback bytes, it can redirect output to an attacker or inflate monAmount to consume extra MON. This is an integration brittleness beyond the IPoolManager interface guarantees. The shMON v4 path demonstrates stronger defenses (hardcoded recipient and settle return check), which are missing here.

Severity

Impact Explanation: [High] In the worst case, stable-asset proceeds are redirected to an attacker and extra native MON is consumed, causing direct, material loss of principal funds.

Likelihood Explanation: [Low] Exploitation requires the external PoolManager integration to deviate from canonical behavior (e.g., proxy misconfiguration or wrapper altering callback bytes), plus additional state conditions for some scenarios.

Exploitation

Exploitation Scenarios:

Scenario 1.

Recipient redirection: The PoolManager alters the callback payload to replace recipient with an attacker address before invoking unlockCallback. The strategy settles the MON delta, but take sends the stable-asset output to the attacker. The post-unlock balance delta at the strategy is zero, so the function returns without reverting, resulting in stolen reward proceeds.

Preconditions / Assumptions

- (a). PoolManager deviates from canonical behavior and alters callback bytes to change recipient before invoking the strategy’s unlockCallback
- (b). Operator triggers processRewards with outputAsset = VAULT_ASSET and valid RewardSwapConfig
- (c). Swap executes within the specified price bound and produces amount1 >= minAmountOut

Scenario 2.

Overconsumption of MON: The PoolManager inflates monAmount in the callback payload. Swapper.unlockCallback executes a larger exact-input swap and settles a larger negative MON delta using any extra idle MON held by the strategy. The strategy unintentionally spends more MON than intended, depleting idle MON balances.

Preconditions / Assumptions

- (a). PoolManager deviates from canonical behavior and alters callback bytes to inflate monAmount before invoking the strategy’s unlockCallback
- (b). Strategy holds extra idle native MON beyond the just-unwrapped amount
- (c). Swap executes within the specified price bound so the larger exact-input is consumed

Scenario 3.

Combined attack: The PoolManager both inflates monAmount and redirects recipient. The strategy pays extra MON to settle the inflated input while the stable-asset output is delivered to the attacker, causing principal loss of both native MON and stable-asset rewards.

Preconditions / Assumptions

- (a). PoolManager deviates from canonical behavior and alters callback bytes to inflate monAmount and change recipient
- (b). Strategy holds extra idle native MON sufficient to cover the inflated negative delta
- (c). Swap executes within the specified price bound and produces amount1 >= minAmountOut

Proposed fix

Swapper.sol

File: `src/adapters/rewards/Swapper.sol`

Source

```
 ... 233 unchanged lines ...

     /// @notice Execute the trusted Uniswap v4 unlock callback for reward swaps.
+        // SECURITY: Enforce local receipt: require outputAmount >= params.minAmountOut to ensure proceeds
+        // were delivered to this contract; revert if below to avoid silent non-delivery despite callback delta
     /// @dev The callback always performs a native MON -> vault-asset swap where currency0 is MON
     ///      and currency1 is the vault asset. The pool key is validated again here so the callback
 ... 30 unchanged lines ...
         uint256 settleAmount = uint256(uint128(-raw0));
         // Settle the MON that the pool manager is owed for the swap.
+        // SECURITY: Capture and assert the return value of settle() equals settleAmount (mirror ShMonSwapAdapt
+        // to detect PoolManager anomalies early and revert atomically on mismatch.
         IPoolManager(TRUSTED_POOL_MANAGER).settle{ value: settleAmount }();

         int128 raw1 = delta.amount1();
         if (raw1 <= 0) revert UnexpectedAmount1Delta(raw1);

         // casting to 'uint128' is safe because a positive 'int128' value cannot exceed the max 'uint128' value
         // forge-lint: disable-next-line(unsafe-typecast)
         uint256 takeAmount = uint256(uint128(raw1));
         if (takeAmount < callbackData.minAmountOut) {
             revert InsufficientRewardOutput(takeAmount, callbackData.minAmountOut);
         }
```

```
        // Pull the swapped vault asset out of the pool manager into the designated strategy recipient.
+       // SECURITY: Consider ignoring callbackData.recipient and using address(this) to prevent redirected pro
        IPoolManager(TRUSTED_POOL_MANAGER)
            .take(IUniswapV4.Currency.wrap(callbackData.stableAsset), callbackData.recipient, takeAmount);
 ... 36 unchanged lines ...
```

RewardsAdapter.sol

File: `src/adapters/rewards/RewardsAdapter.sol`

Source

```
 ... 186 unchanged lines ...
        emit RewardUnlockStatusChanged(false);
        if (unlockContext == UNLOCK_CONTEXT_REWARD_SWAP) {
+           // SECURITY: Consider binding the callback payload by storing a keccak256 commitment before unlock
+           // and asserting equality here to prevent mutated recipient/monAmount or other parameters.
            return Swapper.unlockCallback(data);
        }
 ... 66 unchanged lines ...
```

Related findings

[Low] Trusting callback-supplied recipient/poolKey and missing minOut recheck in Swapper unlock path causes diverted or under-delivered reward outputs

Description

Swapper.unlockCallback uses recipient and pool/payload fields decoded from PoolManager callback bytes without re-asserting intended recipient or configured poolKey, and Swapper.processRewards does not enforce minAmountOut on the actually received tokens. A deviating PoolManager can redirect rewards or alter pool/params, and fee-on-transfer tokens can under-deliver without revert.

In reward swaps to the vault asset, Swapper.processRewards encodes UnlockCallbackData with recipient = address(this) and calls PoolManager.unlock. In Swapper.unlockCallback, the data is decoded and pool currencies are validated only relative to the callback-supplied stableAsset; the function then calls PoolManager.take using callbackData.recipient directly, without asserting recipient == address(this) or that the poolKey equals the configured trusted pool. After the callback returns, Swapper.processRewards only checks the return length and measures the local stableAsset balance delta; it does not enforce params.minAmountOut on the actually received tokens. If the PoolManager deviates from canonical behavior and forwards modified callback bytes (e.g., different recipient, poolKey, or minAmountOut), swap proceeds can be diverted or misrouted while still satisfying delta-level checks. Even with an honest PoolManager, if the stable asset is fee-on-transfer or otherwise nonstandard, the strategy can receive fewer tokens than minAmountOut without revert because minAmountOut is not enforced on the final balance delta.

Severity

Impact Explanation: [Medium] Loss primarily affects reward proceeds (yield) during post-processing rather than principal collateral, constituting a material yield loss.

Likelihood Explanation: [Low] Primary exploitation requires a trusted integration (PoolManager) to deviate from canonical behavior; the fee-on-transfer case depends on selecting a nonstandard token.

Exploitation

Exploitation Scenarios:

Scenario 1.

PoolManager modifies callback recipient to steal proceeds: Operator initiates rewards processing to VAULT_ASSET. Swapper.processRewards calls unlock with recipient = address(this). A deviating PoolManager invokes the callback with modified bytes setting recipient = attacker. Swapper.unlockCallback validates currencies relative to the supplied

stableAsset, executes the swap, enforces takeAmount >= minAmountOut, then calls take to the attacker's address. Post-unlock, the strategy measures zero increase and returns without revert, losing the swapped MON proceeds to the attacker.

Preconditions / Assumptions

- (a). The PoolManager at TRUSTED_POOL_MANAGER deviates from canonical behavior and forwards modified callback bytes (recipient altered).
- (b). Operator initiates RewardsAdapter.processRewards with outputAsset == VAULT_ASSET and a non-zero minAmountOut.
- (c). A valid rewardSwapConfig is set.

Scenario 2.

PoolManager modifies poolKey/stableAsset/minAmountOut to misroute or degrade execution: Operator triggers reward swap. The PoolManager calls back with a different poolKey and matching stableAsset (or with reduced minAmountOut). Swapper.unlockCallback only checks currencies against the supplied stableAsset and proceeds. The swap runs on an attacker-chosen market or with a weaker bound, and proceeds are taken to the supplied recipient (address(this) or attacker if also modified). The strategy either receives an unexpected asset (measured output zero) or fewer tokens than intended, resulting in loss of reward value.

Preconditions / Assumptions

- (a). The PoolManager deviates from canonical behavior and forwards modified callback bytes (poolKey/stableAsset and/or minAmountOut altered consistently).
- (b). Operator initiates RewardsAdapter.processRewards with outputAsset == VAULT_ASSET.
- (c). A valid rewardSwapConfig is set.

Scenario 3.

Fee-on-transfer stable asset undermines minAmountOut on actual receipt: Operator sets a non-zero minAmountOut and performs a reward swap with a fee-on-transfer stablecoin. Swapper.unlockCallback enforces minAmountOut against the Uniswap delta and calls take for the pre-fee amount. Due to transfer fees, the strategy's credited amount is lower than minAmountOut. Swapper.processRewards does not re-check minAmountOut on the measured balance delta, so the under-delivery is silently accepted.

Preconditions / Assumptions

- (a). The configured vault asset is a fee-on-transfer or otherwise nonstandard ERC20.
- (b). The PoolManager behaves canonically and forwards bytes unchanged.
- (c). Operator provides a non-zero minAmountOut expecting enforcement on actual receipt.

Proposed fix

Swapper.sol

File: `src/adapters/rewards/Swapper.sol`

Source

```
 ... 230 unchanged lines ...
         }
         outputAmount = IERC20(stableAsset).balanceOf(address(this)) - stableBefore;
+        if (outputAmount < params.minAmountOut) {
+            revert InsufficientRewardOutput(outputAmount, params.minAmountOut);
+        }
     }

 ... 47 unchanged lines ...
         // Pull the swapped vault asset out of the pool manager into the designated strategy recipient.
         IPoolManager(TRUSTED_POOL_MANAGER)
-            .take(IUniswapV4.Currency.wrap(callbackData.stableAsset), callbackData.recipient, takeAmount);
+            .take(IUniswapV4.Currency.wrap(callbackData.stableAsset), address(this), takeAmount);
```

```
        return abi.encode(delta);
... 34 unchanged lines ...
```

RewardsAdapter.sol

File: `src/adapters/rewards/RewardsAdapter.sol`

Source

```
... 186 unchanged lines ...
        emit RewardUnlockStatusChanged(false);
        if (unlockContext == UNLOCK_CONTEXT_REWARD_SWAP) {
+           Swapper.UnlockCallbackData memory cbd = abi.decode(data, (Swapper.UnlockCallbackData));
+           if (
+               !Swapper.poolKeysEqual(cbd.poolKey, $.rewardSwapConfig.poolKey)
+                   || cbd.stableAsset != _vaultAsset()
+           ) {
+               revert Swapper.UnexpectedRewardPoolKey();
+           }
            return Swapper.unlockCallback(data);
        }
... 66 unchanged lines ...
```

15. [Medium] Staleness masking of shMON/MON leg in CompositeOracle and single-timestamp freshness in AusdStrategy causes overminted shares/overpaid withdrawals

Status

Review status: Unresolved
Remediation status: Unremediated
Remediation note: Created by pipeline analysis

Description

AusdStrategy's NAV can be incorrect when consumed for share pricing because CompositeOracle may surface a fresh updatedAt while using a stale shMON/MON leg, and the strategy's OracleHelpers checks only that single updatedAt. With no NAV-side guard and a negative-to-zero NAV floor, deposits can overmint shares and withdrawals can be overpaid.

AusdStrategy's _calculateTotalValue() values shMON via SHMON_ORACLE using OracleHelpers.getPrice, which enforces freshness against a single updatedAt from oracle.latestRoundData(). CompositeOracle composes shMON/USD by multiplying a shMON/MON base leg and a MON/USD quote leg, picking the newest source within each leg without rejecting stale data by heartbeat, and then sets composite updatedAt as the quote's updatedAt whenever the base leg is within its (potentially long) heartbeat. This can mask a stale base-leg price behind a fresh composite updatedAt. The strategy then accepts this as fresh due to the single-timestamp check and uses the stale base-leg price in NAV. The vault's share pricing (after yield accrual) relies on this NAV, so deposits can overmint when NAV is understated and withdrawals can be overpaid when NAV is overstated. The basis guard is only enforced on allocateFunds/deallocateFunds and does not protect NAV reads. The negative-to-zero floor in _calculateTotalValue() can amplify overminting when temporary undervaluation pushes net value below debts.

Severity

Impact Explanation: [High] Overminting and over-redemption are direct, material losses of principal for existing LPs.

Likelihood Explanation: [Low] Exploitation depends on privileged configuration (e.g., misaligned CompositeOracle base-leg heartbeat or selection of a manipulable on-chain source) and/or operator timing of deposits/withdrawal processing during stale/manipulated windows, which are plausible but not fully attacker-controlled.

Exploitation

Exploitation Scenarios:

Scenario 1.

Overminted shares via stale undervaluation at deposit time: CompositeOracle reports a fresh updatedAt (from the quote leg) while the base leg's shMON/MON price is stale but within its long heartbeat; AusdStrategy accepts it as fresh and undervalues shMON collateral; attacker deposits when deposits are open, receiving excess shares; later normalization reveals dilution of existing LPs.

Preconditions / Assumptions

- (a). SHMON_ORACLE is CompositeOracle
- (b). CompositeOracle SHMON_MON_HEARTBEAT is set longer than the strategy's SHMON_ORACLE_HEARTBEAT, allowing base-leg staleness to be masked by the quote leg's updatedAt
- (c). Base shMON/MON leg is stale but within CompositeOracle's heartbeat and true shMON PPS has increased materially since last update
- (d). Deposits are permissionless (or the manager processes deposits during the stale window)
- (e). OracleHelpers checks only the composite updatedAt and accepts it as fresh

Scenario 2.

Overpaid withdrawals via stale overvaluation at epoch processing: CompositeOracle masks a stale high shMON/MON base price behind a fresh updatedAt; NAV is overstated; queued withdrawals process against inflated total assets, overpaying withdrawers and harming remaining LPs.

Preconditions / Assumptions

- (a). SHMON_ORACLE is CompositeOracle
- (b). CompositeOracle SHMON_MON_HEARTBEAT is set longer than the strategy's SHMON_ORACLE_HEARTBEAT, allowing base-leg staleness to be masked by the quote leg's updatedAt
- (c). Base shMON/MON leg is stale but within CompositeOracle's heartbeat and true shMON PPS has decreased materially since last update
- (d). Withdrawal epoch processing occurs during the stale window
- (e). OracleHelpers checks only the composite updatedAt and accepts it as fresh

Scenario 3.

Sandwiching with a manipulable on-chain AggregatorV3Interface leg: Governance wires a CompositeOracle leg to a short-window AMM TWAP or similar; attacker moves the on-chain price/TWAP, CompositeOracle returns a fresh composite updatedAt; OracleHelpers accepts it; NAV is distorted during deposits/withdrawals, enabling overmint/over-redemption at LPs' expense.

Preconditions / Assumptions

- (a). Governance configures a CompositeOracle leg to an AggregatorV3Interface-compatible on-chain source that is manipulable (e.g., short-window AMM TWAP or upgradeable admin oracle)
- (b). Attacker can move the on-chain market or TWAP to distort the oracle leg briefly
- (c). CompositeOracle returns a fresh composite updatedAt using the manipulated leg; OracleHelpers accepts it
- (d). Deposits are permissionless or withdrawal processing aligns with the manipulated window

Proposed fix

CompositeOracle.sol

File: `src/oracle/CompositeOracle.sol`

Source

```
 ... 180 unchanged lines ...
         startedAt = baseData.startedAt < quoteData.startedAt ? baseData.startedAt : quoteData.startedAt;

-        // When the slow shMON/MON leg is within its heartbeat, reflect MON/USD freshness.
-        if (block.timestamp - baseData.updatedAt <= baseLeg.heartbeat) {
-            updatedAt = quoteData.updatedAt;
-        } else {
-            updatedAt = baseData.updatedAt;
-        }
```

```
+        // Reflect the oldest component's freshness so the composite cannot appear fresher than a stale leg.
+        updatedAt = baseData.updatedAt < quoteData.updatedAt ? baseData.updatedAt : quoteData.updatedAt;

         // Reuse the base round lineage as the canonical round identifier for the composite output.
 ... 30 unchanged lines ...
             (RoundData memory candidate, bool valid) = _getChainlinkSourceData(leg.sources[i]);
             if (!valid) continue;
+            if (block.timestamp - candidate.updatedAt > leg.heartbeat) continue;
             if (!found || candidate.updatedAt > data.updatedAt) {
                 data = candidate;
                 found = true;
             }
         }

         if (leg.hasPyth) {
             (RoundData memory candidate, bool valid) = _getPythSourceData(leg.pyth, leg.pythId);
-            if (valid && (!found || candidate.updatedAt > data.updatedAt)) {
+            if (valid && (block.timestamp - candidate.updatedAt <= leg.heartbeat) && (!found || candidate.updat
                 data = candidate;
                 found = true;
 ... 115 unchanged lines ...
```

Related findings

[Medium] Missing per-leg staleness checks and unsafe Pyth getter in CompositeOracle causes stale-price-based mis-sizing and potential liquidation

Description

CompositeOracle uses Pyth.getPriceUnsafe and does not enforce per-leg staleness (heartbeat) checks. It also sets updatedAt to the fresher leg in certain cases, which can mask a stale leg. Downstream consumers (OracleHelpers/LTVCalculations) enforce freshness using only the composite's updatedAt, so stale component data can be accepted as fresh. This can mis-size max-borrow and safe-withdraw calculations, potentially over-leveraging positions and leading to liquidation after oracles refresh. AusdStrategy's basis guard mitigates large divergences but does not fully eliminate risk.

CompositeOracle aggregates shMON/MON (base leg) and MON/USD (quote leg) to produce a composite shMON/USD price. The leg readers do not enforce a max-age bound: _getPythSourceData uses IPyth.getPriceUnsafe (no age check), and _getChainlinkSourceData also omits heartbeat checks. _getLatestLegData selects the newest valid source by timestamp but never rejects stale candidates. In latestRoundData, updatedAt is set to the quote leg's timestamp if the base leg is 'within its heartbeat' (a value not actually enforced for validity), otherwise to the base leg's timestamp. Downstream, OracleHelpers.getPrice (used by LTVCalculations in adapters/strategies) checks only the single updatedAt from the composite against a heartbeat, not the legs independently. As a result, a stale base leg can be masked by a fresh quote leg, causing the composite to pass freshness checks and be used in execution flows (max borrow/safe withdrawal). AusdStrategy's basis guard compares oracle-priced shMON vs implied PPS via WMON, and defaults to a 2% threshold, which mitigates large stale-leg divergences but allows small ones. Underlying protocols (Curvance/Euler) may also enforce health checks, converting some mis-sizings into reverts, but this does not fully remove the risk that mis-sized actions succeed and positions become over-levered or under-collateralized when oracles refresh.

Severity

Impact Explanation: [High] Accepting stale leg data as fresh can cause inflated max-borrow and overstated safe-withdraw calculations, potentially leading to material principal loss via liquidation or forced unwind after oracles refresh. While the basis guard mitigates large divergences, smaller masked staleness can still result in harmful mis-sizing near thresholds.

Likelihood Explanation: [Low] Multiple preconditions must align: CompositeOracle must be used in production; base leg must be stale beyond the consumer's heartbeat while the quote is fresh; the operator must rely on the on-chain 'max-safe' paths; the basis guard must not trigger (deviation within threshold); and the underlying protocol must permit the action. These multiplicative conditions reduce likelihood.

Exploitation

Exploitation Scenarios:

Scenario 1.

Inflated max-borrow: The shMON/MON leg is stale (e.g., 2 hours) while MON/USD (Pyth) is fresh. CompositeOracle reports updatedAt from the fresh quote leg, so OracleHelpers accepts the price. LTVCalculations overvalues shMON collateral and returns an inflated max borrow. The strategy borrows more than intended; when oracles refresh, the position becomes unsafe and can be liquidated.

Preconditions / Assumptions

- (a). CompositeOracle is configured and used as the shMON/USD oracle by the strategy/adapters.
- (b). Base leg = shMON/MON via Chainlink-style sources; quote leg = MON/USD via Pyth (hasPyth = true).
- (c). Per-leg heartbeats are configured but not enforced by CompositeOracle's leg readers.
- (d). Consumer heartbeat (used in OracleHelpers.getPrice) is stricter than the base leg's heartbeat.
- (e). Base leg becomes stale beyond the consumer heartbeat; quote leg remains fresh; base is still 'within' its own heartbeat so CompositeOracle sets updatedAt to quote.updatedAt.
- (f). Operator relies on on-chain 'max-safe' path (amount == 0) so adapters compute max borrow from oracles.
- (g). AusdStrategy basis guard does not trigger (basis deviation <= configured guard, e.g., 2%).
- (h). Underlying protocol permits the borrow under its own health checks.
- (i). Position is close enough to thresholds that the mis-sizing can lead to liquidation on oracle refresh.

Scenario 2.

Overstated safe withdrawal: With the same masked-staleness setup, getMaxSafeWithdrawal overvalues shMON collateral, returning a larger 'safe' withdrawal. The operator withdraws too much collateral; the remaining position is closer to liquidation. When oracles refresh, the position can breach thresholds and be liquidated.

Preconditions / Assumptions

- (a). CompositeOracle is configured and used as the shMON/USD oracle by the strategy/adapters.
- (b). Same leg configuration and heartbeat behavior as Scenario 1.
- (c). Base leg stale beyond consumer heartbeat; quote leg fresh; CompositeOracle updatedAt reflects quote leg.
- (d). Operator relies on on-chain 'safe withdrawal' calculation via adapter.
- (e). AusdStrategy basis guard does not trigger (basis deviation <= guard).
- (f). Underlying protocol permits the withdrawal under its own checks.
- (g). Position is near health thresholds so mis-sizing can push it into liquidation risk upon oracle refresh.

Scenario 3.

External integrator reliance: An external integrator uses CompositeOracle.latestRoundData and applies a single heartbeat check to updatedAt. Because CompositeOracle reports the fresher leg's updatedAt, the integrator accepts a composite price built from a stale leg, leading to mis-accounting or unsafe execution decisions.

Preconditions / Assumptions

- (a). An external integrator uses CompositeOracle.latestRoundData directly and enforces a single heartbeat check against updatedAt.
- (b). Base leg stale beyond the integrator's intended freshness; quote leg fresh; CompositeOracle updatedAt reflects quote leg.
- (c). Integrator trusts the composite as fresh and proceeds with pricing/risk logic without independently validating each leg's staleness.

Proposed fix

CompositeOracle.sol

File: `src/oracle/CompositeOracle.sol`

Source

```
... 181 unchanged lines ...

        // When the slow shMON/MON leg is within its heartbeat, reflect MON/USD freshness.
```

```
-        if (block.timestamp - baseData.updatedAt <= baseLeg.heartbeat) {
-            updatedAt = quoteData.updatedAt;
-        } else {
-            updatedAt = baseData.updatedAt;
-        }
-
+        updatedAt = baseData.updatedAt < quoteData.updatedAt ? baseData.updatedAt : quoteData.updatedAt;
         // Reuse the base round lineage as the canonical round identifier for the composite output.
         answeredInRound = baseData.answeredInRound;
 ... 27 unchanged lines ...

         for (uint256 i = 0; i < leg.sources.length; i++) {
-            (RoundData memory candidate, bool valid) = _getChainlinkSourceData(leg.sources[i]);
+            (RoundData memory candidate, bool valid) = _getChainlinkSourceData(leg.sources[i], leg.heartbeat);
             if (!valid) continue;
             if (!found || candidate.updatedAt > data.updatedAt) {
                 data = candidate;
                 found = true;
             }
         }

         if (leg.hasPyth) {
-            (RoundData memory candidate, bool valid) = _getPythSourceData(leg.pyth, leg.pythId);
+            (RoundData memory candidate, bool valid) = _getPythSourceData(leg.pyth, leg.pythId, leg.heartbeat);
             if (valid && (!found || candidate.updatedAt > data.updatedAt)) {
                 data = candidate;
                 found = true;
             }
         }

         if (!found) revert InvalidPrice();
     }

-    function _getChainlinkSourceData(AggregatorV3Interface oracle)
+    function _getChainlinkSourceData(AggregatorV3Interface oracle, uint256 heartbeat)
         private
         view
         returns (RoundData memory data, bool valid)
     {
         try oracle.latestRoundData() returns (
             uint80 roundId, int256 price, uint256 startedAt, uint256 updatedAt, uint80 answeredInRound
         ) {
             if (price <= 0) return (data, false);
             if (answeredInRound < roundId) return (data, false);
             if (updatedAt == 0 || updatedAt > block.timestamp) return (data, false);
+            if (block.timestamp - updatedAt > heartbeat) return (data, false);

             try oracle.decimals() returns (uint8 priceDecimals) {
                 data = RoundData(roundId, price, startedAt, updatedAt, answeredInRound, priceDecimals);
                 return (data, true);
             } catch {
                 return (data, false);
             }
         } catch {
             return (data, false);
         }
     }

-    function _getPythSourceData(IPyth pyth, bytes32 priceId) private view returns (RoundData memory data, bool
-        try pyth.getPriceUnsafe(priceId) returns (IPyth.Price memory price) {
+    function _getPythSourceData(IPyth pyth, bytes32 priceId, uint256 heartbeat) private view returns (RoundData
+        try pyth.getPriceNoOlderThan(priceId, heartbeat) returns (IPyth.Price memory price) {
```

```
            if (price.price <= 0) return (data, false);
            if (price.publishTime == 0 || price.publishTime > block.timestamp) return (data, false);
... 82 unchanged lines ...
```

16. [Low] Missing self/cycle validation and unbounded upstream calls in CompositeOracle cause allocation DoS via oracle failure

Status

Review status: Unresolved
Remediation status: Unremediated
Remediation note: Created by pipeline analysis

Description

CompositeOracle does not prevent self-referential or cyclic source configurations and calls upstream feeds without gas caps. If a leg is wired to call back into the same CompositeOracle (or a second composite that calls back into it), reads recurse and the leg resolves as invalid; with both leg sources unusable (and no Pyth on that leg), the composite reverts. AusdStrategy relies on this composite for shMON/USD in its basis guard, so a miswired composite blocks allocateFunds and disrupts valuation analytics. This is an admin misconfiguration/liveness risk rather than an external attacker exploit.

The CompositeOracle constructor enforces only non-zero source addresses, heartbeat bounds, and coherent Pyth configuration; it does not forbid setting any leg source to this CompositeOracle instance, nor detect cycles. During latestRoundData(), each leg is resolved by iterating sources and calling external latestRoundData()/decimals() (and optionally Pyth getPriceUnsafe()) with full gas and no recursion guard. If a configured source points back to this CompositeOracle (or to another composite that calls back), calls recurse until a revert; the try/catch then marks that candidate invalid. When all sources for a leg are unusable (e.g., both self-referential/invalid and no Pyth for that leg), _getLatestLegData reverts InvalidPrice, making the composite feed unavailable. AusdStrategy's allocateFunds enforces a basis guard that calls getBasisBps(), which in turn uses OracleHelpers.getPrice on the shMON/USD oracle; if that oracle is this miswired CompositeOracle, the basis check reverts and allocation fails. Deallocation tolerates basis-read failure best-effort, so funds are not frozen, but allocation and some analytics are significantly impaired until reconfiguration. The issue is entirely dependent on admin/deployer configuration and has no post-deployment attacker path.

Severity

Impact Explanation: [Medium] Allocations are blocked by the basis guard and strategy valuation degrades, causing significant but temporary availability loss of a core strategy operation and material yield loss until the oracle is corrected. No principal loss or long-term fund freeze occurs.

Likelihood Explanation: [Low] All scenarios require admin/deployer misconfiguration or insider malice at deployment (self-referential or cyclic wiring, or selecting a malicious oracle). There is no post-deployment external attacker path.

Exploitation

Exploitation Scenarios:

Scenario 1.

Self-referential shMON/MON leg: CompositeOracle is used as AusdStrategy's SHMON_ORACLE, and one or both shMON/MON sources are set to address(this). latestRoundData() recurses when resolving that leg and ultimately reverts as invalid; getBasisBps() fails and AusdStrategy.allocateFunds reverts, blocking allocations and yield accrual.

Preconditions / Assumptions

- (a). AusdStrategy is configured to use this CompositeOracle as SHMON_ORACLE
- (b). CompositeOracle is deployed with SHMON_MON_SOURCE_0 and/or SHMON_MON_SOURCE_1 set to address(this) (e.g., via CREATE2 predictable address)
- (c). MON/USD leg is otherwise valid (any standard Chainlink/Redstone/Pyth)

Scenario 2.

Two-composite cycle: Two CompositeOracle instances (A and B) are deployed; A's shMON/MON leg lists B and B's lists A. AusdStrategy uses A as SHMON_ORACLE. Resolving the leg triggers A↔B recursion, the leg resolves invalid, the composite reverts, and allocateFunds is blocked by the basis guard.

Preconditions / Assumptions

- (a). Two CompositeOracle instances A and B are deployed
- (b). A's shMON/MON leg includes B as a source; B's shMON/MON leg includes A as a source (cycle)
- (c). AusdStrategy uses A as SHMON_ORACLE
- (d). MON/USD legs are otherwise valid

Scenario 3.

Mixed bad leg with no Pyth: CompositeOracle is SHMON_ORACLE; SHMON_MON_SOURCE_0 is self-referential (address(this)), SHMON_MON_SOURCE_1 is stale/invalid, and there is no Pyth for that leg. Both sources are rejected; the leg has no valid source; composite reverts; basis guard fails; allocateFunds reverts.

Preconditions / Assumptions

- (a). AusdStrategy uses this CompositeOracle as SHMON_ORACLE
- (b). SHMON_MON_SOURCE_0 is address(this) (self-referential)
- (c). SHMON_MON_SOURCE_1 is a non-zero but invalid/stale aggregator
- (d). No Pyth is configured for the shMON/MON leg

Proposed fix

CompositeOracle.sol

File: `src/oracle/CompositeOracle.sol`

Source

```
 ... 124 unchanged lines ...
             revert InvalidPythConfig();
         }
+        // Prevent self-referential or cyclic CompositeOracle sources and degenerate duplicates.
+        if (shMonMonSource0_ == shMonMonSource1_ || monUsdSource0_ == monUsdSource1_) revert InvalidSource();
+        bytes32 selfHash; assembly { selfHash := extcodehash(address()) }
+        bytes32 h;
+        assembly { h := extcodehash(shMonMonSource0_) } if (h == selfHash) revert InvalidSource();
+        assembly { h := extcodehash(shMonMonSource1_) } if (h == selfHash) revert InvalidSource();
+        assembly { h := extcodehash(monUsdSource0_) } if (h == selfHash) revert InvalidSource();
+        assembly { h := extcodehash(monUsdSource1_) } if (h == selfHash) revert InvalidSource();

         SHMON_MON_SOURCE_0 = AggregatorV3Interface(shMonMonSource0_);
 ... 220 unchanged lines ...
```

Related findings

[Informational] Missing try/catch around oracle.decimals() in OracleHelpers.getPrice causes inconsistent error handling and potential operational friction under oracle misbehavior

Description

OracleHelpers.getPrice wraps latestRoundData() in try/catch but calls decimals() unguarded. If decimals() reverts (e.g., due to a misconfigured or non-standard oracle), callers see a raw revert rather than the intended OracleCallReverted error. While upstream oracle failures would still cause execution to fail regardless, this omission reduces error consistency and can create operational friction when allocation or dynamic LTV-based sizing is attempted during oracle misbehavior.

In OracleHelpers.getPrice, the external call to oracle.latestRoundData() is properly wrapped in try/catch, but the subsequent call to oracle.decimals() is not. If an oracle returns valid round data yet reverts on decimals(), getPrice reverts with a raw error, not the standardized OracleCallReverted error. OracleHelpers is used by LTVCalculations and AusdStrategy for basis checks and valuation. Practically, allocation or dynamic deallocation/borrow sizing can still fail when an oracle misbehaves, but that failure would occur even if decimals() were wrapped, because the code is intentionally fail-fast on oracle-call issues. Termination flows do not depend on OracleHelpers and remain available.

Deallocation's basis guard is already try/caught to proceed when basis reads fail. Therefore, the primary impact is inconsistent error surfacing and diagnosability, with low-likelihood operational friction if a non-standard or regressing oracle is used.

Severity

Impact Explanation: [Informational] The missing try/catch around decimals() only affects error-surface consistency (raw revert vs custom error). Oracle failures would still cause execution to revert regardless; termination is unaffected; some deallocation paths already bypass basis-guard errors and operators can use explicit amounts as a workaround.

Likelihood Explanation: [Low] Requires a non-standard/misconfigured or regressing oracle to revert on decimals(); canonical Chainlink/CompositeOracle implementations are unlikely to do so; not attacker-controlled.

Exploitation

Exploitation Scenarios:

Scenario 1.

Allocation fails when SHMON_ORACLE successfully returns latestRoundData() but reverts on decimals(): AusdStrategy.allocateFunds runs the basis guard (getBasisBps), which calls OracleHelpers.getPrice for SHMON; decimals() reverts and bubbles up, aborting allocation.

Preconditions / Assumptions

- (a). SHMON_ORACLE is a non-standard or misconfigured Aggregator that reverts on decimals() while latestRoundData() succeeds
- (b). Vault calls AusdStrategy.allocateFunds, triggering the basis guard read via OracleHelpers

Scenario 2.

Dynamic safe-withdraw deallocation fails when either the collateral or debt oracle reverts on decimals(): BaseStrategy._withdrawCollateral requests adapter.getMaxSafeWithdrawal (amount == 0), which calls LTVCalculations → OracleHelpers.getPrice; decimals() reverts, causing the deallocation command to fail.

Preconditions / Assumptions

- (a). Collateral or debt oracle for a Curvance/Euler leg is non-standard or misconfigured and reverts on decimals() while latestRoundData() succeeds
- (b). Deallocation batch uses amount == 0 (adapter-computed max safe withdrawal), invoking LTVCalculations and OracleHelpers

Scenario 3.

Dynamic borrow sizing (amount == 0) fails when the debt oracle reverts on decimals(): BaseStrategy._borrow requests adapter.getMaxBorrowAmount, which calls LTVCalculations → OracleHelpers.getPrice; decimals() reverts and the borrow step fails.

Preconditions / Assumptions

- (a). Debt oracle is non-standard or misconfigured and reverts on decimals() while latestRoundData() succeeds
- (b). Borrow command uses amount == 0 (adapter-computed max borrow), invoking LTVCalculations and OracleHelpers

Proposed fix

OracleHelpers.sol

File: `src/libraries/OracleHelpers.sol`

Source

```
... 124 unchanged lines ...
```

```
             // Casting to `uint256` is safe because negative values were already rejected above.
+            uint8 _decimals;
+            try oracle.decimals() returns (uint8 d) {
+                _decimals = d;
+            } catch {
+                revert OracleCallReverted();
+            }
             // forge-lint: disable-next-line(unsafe-typecast)
-            return (uint256(priceInt), oracle.decimals());
+            return (uint256(priceInt), _decimals);
         } catch {
             // Surface a deterministic protocol error instead of the raw aggregator revert payload.
             revert OracleCallReverted();
         }
     }
 }
```

17. [Low] Missing duplicate-token validation in MerklAdapter claim request causes harvest revert and potential missed rewards

Status

Review status: Unresolved
Remediation status: Unremediated
Remediation note: Created by pipeline analysis

Description

MerklAdapter does not reject duplicate tokens in MerklClaimRequest. It filters entries using a single pre-claim snapshot and forwards all entries whose cumulative amount exceeds that snapshot. With duplicate tokens for the same recipient, both can be sent; after the first updates state, the second may become zero-delta and cause the distributor to revert mid-call. This results in a reverted harvest and, in certain environments with rotating Merkle roots, can lead to missed rewards.

MerklAdapter._validateClaim only checks that the distributor is nonzero, the token list is non-empty, and array lengths match. It does not require claimRequest.tokens to be unique. In claim(), the adapter reads merklDistributor.claimed(recipient, token) once per entry before any external call and appends entries where requested cumulative amount > claimed amount. If a request contains duplicate entries for the same (recipient, token) with the same cumulative amount/proof, both pass this local filter and are forwarded to the distributor in a single claim() call. After the first duplicate updates the distributor's cumulative claimed amount to the requested amount, the second becomes a zero-delta claim. Some distributors revert on zero-delta leaves, causing the entire harvest call to fail. This does not allow over-claiming under cumulative semantics, but it can cause operational DoS of harvesting and, if roots rotate with expiry, potential loss of that window's rewards. Only the operator (OPERATOR_ROLE) can submit claims, so this relies on operator-input error rather than an attacker path.

Severity

Impact Explanation: [Medium] Potential direct, material loss of yield if a claim window is missed due to repeated reverts and Merkle root rotation; otherwise impacts are single-transaction harvest/processing failures without asset loss.

Likelihood Explanation: [Low] Relies on a trusted operator mistake (submitting duplicate tokens) and a specific external distributor behavior (reverting on zero-delta), and for missed-window loss also requires timing with root rotation; these multiplicative preconditions make occurrence unlikely.

Exploitation

Exploitation Scenarios:

Scenario 1.

Duplicate token in MerklClaimRequest causes harvest revert: The operator submits a request with two identical entries for the same token and amount A where A > pre-claim C. Both pass the adapter's pre-claim filter. The distributor processes the first (transfers A−C and sets claimed=A), then reverts on the second due to zero-delta, reverting the entire harvest.

Preconditions / Assumptions

- (a). Distributor uses cumulative semantics and reverts on zero-delta leaves
- (b). There are claimable rewards for the (recipient, token) pair with cumulative amount A > pre-claim C
- (c). Operator (trusted role) mistakenly includes duplicate entries for the same token and amount/proof
- (d). Operator calls claimRewards with that request

Scenario 2.

claimAndProcessRewards bundle reverts entirely: The operator uses claimAndProcessRewards with a duplicate-token request. The same mid-call zero-delta revert occurs at the distributor, causing the entire bundled claim-and-process transaction to revert and deferring both harvesting and post-processing.

Preconditions / Assumptions

- (a). Same as Scenario 1
- (b). Operator calls claimAndProcessRewards with the duplicate-token request

Scenario 3.

Missed claim window due to repeated duplicates until root rotation: The operator repeatedly submits duplicate-token requests that revert on a strict distributor during a limited claim window. If the distributor rotates Merkle roots and unclaimed rewards do not carry over, the strategy permanently misses that window’s rewards.

Preconditions / Assumptions

- (a). Same as Scenario 1
- (b). Operator repeatedly submits duplicate-token requests during a claim window
- (c). Distributor rotates Merkle roots on a schedule and unclaimed rewards do not carry over

Proposed fix

MerklAdapter.sol

File: `src/adapters/merkl/MerklAdapter.sol`

Source

```
 ... 58 unchanged lines ...
     /// @param errNum Numeric discriminator that identifies the invalid field.
     /// `1` means zero distributor, `2` means empty token list, `3` means amount length mismatch,
-    /// `4` means proof length mismatch.
+    /// `4` means proof length mismatch, `5` means duplicate token entry.
     error InvalidParams(uint256 errNum);
     /// @notice Revert when none of the provided rewards can be claimed anymore.
 ... 62 unchanged lines ...
         if (claimRequest.amounts.length != len) revert InvalidParams(3);
         if (claimRequest.proofs.length != len) revert InvalidParams(4);
+
+        // Enforce unique tokens to avoid forwarding multiple entries for the same (recipient, token)
+        // which can cause mid-call zero-delta reverts on strict distributors.
+        for (uint256 i = 0; i < len; ++i) {
+            for (uint256 j = i + 1; j < len; ++j) {
+                if (claimRequest.tokens[i] == claimRequest.tokens[j]) revert InvalidParams(5);
+            }
+        }
     }
 }
```

18. [Low] Missing adapter-compatibility and zero-balance guards in BaseStrategy migration causes stranded positions and misreported NAV/LTV

Status

Review status: Unresolved
Remediation status: Unremediated
Remediation note: Created by pipeline analysis

Description

BaseStrategy rewires primary/secondary adapters during initialization/migration without verifying adapter compatibility or that existing positions are closed. Because adapters are delegatecalled and hard-wire their paired markets and oracles, while AusdStrategy assumes fixed asset roles per slot, a miswired upgrade can strand live positions on old venues and desynchronize NAV/LTV/termination logic and safe-sizing from reality.

BaseStrategy._applyMigratedBaseState and __BaseStrategy_init only enforce non-zero adapter addresses when wiring the four adapters. The adapters (Euler/Curvance) use immutable references to their specific counterpart markets/oracles and ignore the provided asset parameter in balance views. AusdStrategy assumes the primary slots are stablecoin collateral + WMON debt and the secondary slots are shMON collateral + stablecoin debt for NAV/LTV and termination. If governance/admin miswires adapters or rewires while positions are still open, (1) previously opened positions remain live at the underlying protocols on the strategy address (delegatecall) but the strategy's own getters and termination act only on the newly wired adapters, and (2) NAV/LTV can be misreported (e.g., stablecoin debt priced as WMON) and zero-amount safe-sizing can be desynced (typically reverting at protocol level). This creates operational risk (stranded/unmanaged positions) and economic risk (incorrect fee accounting) despite no public attacker path.

Severity

Impact Explanation: [Medium] Realized, immediate impacts include significant but temporary availability/risk-management degradation (termination and management ignoring stranded legs) and direct, material fee/yield mis-accounting; principal loss via liquidation requires additional conditions beyond admin misconfiguration.

Likelihood Explanation: [Low] All scenarios require trusted-role (admin) misuse or mistake during migration/rewiring; underlying protocols also enforce health on borrow/withdraw, and stranded positions remain on-chain observable, reducing chance of escalation.

Exploitation

Exploitation Scenarios:

Scenario 1.

Admin rewires primary adapters from Curvance to Euler without unwinding; the old Curvance collateral/debt remain live under the strategy address, but AusdStrategy views and termination now only see Euler, ignoring the stranded Curvance leg and exposing users to unmanaged interest/liquidation risk until ops rewire back and unwind.

Preconditions / Assumptions

- (a). An open primary Curvance position (stablecoin collateral + WMON debt) exists under the strategy address
- (b). Strategy admin can call reinitializeUpgrade and rewires adapters without prior unwind
- (c). No immediate follow-up rewire back or manual unwind occurs

Scenario 2.

Admin miswires PRIMARY_DEBT_ADAPTER to a stablecoin debt adapter; AusdStrategy still treats primary debt as WMON and converts the returned balance with the WMON oracle/decimals, mispricing total debt and NAV, leading to material fee mis-accounting if the vault uses strategy-reported value for fees.

Preconditions / Assumptions

- (a). Strategy admin rewires PRIMARY_DEBT_ADAPTER to a stablecoin debt market
- (b). Vault/fee logic relies on strategy-reported value (totalAllocatedValue/getCurrentValue) for performance fees

Scenario 3.

Admin mismatches secondary adapters so SECONDARY_DEBT_ADAPTER points to a market with zero debt while SECONDARY_COLLATERAL_ADAPTER remains correct; termination reads zero secondary debt and tries to withdraw all shMON, but the underlying protocol blocks withdrawal because real debt is still open, causing termination to fail until wiring is corrected.

Preconditions / Assumptions

- (a). Secondary shMON collateral and stablecoin debt are open at one venue
- (b). Strategy admin rewires SECONDARY_DEBT_ADAPTER to a different market where the strategy has no debt
- (c). SECONDARY_COLLATERAL_ADAPTER remains pointed at the original shMON collateral venue

Proposed fix

AusdStrategy.sol

File: `src/strategies/AusdStrategy.sol`

Source

```
 ... 1153 unchanged lines ...
     }

+    function _beforeAdaptersUpdated(address, address, address, address) internal view override {
+        _assertNoPendingShMonadUnstakes();
+        if (
+            _getCollateralBalance(address(PRIMARY_COLLATERAL_ADAPTER()), address(STABLECOIN)) != 0 ||
+            _getDebtBalance(address(PRIMARY_DEBT_ADAPTER()), address(WMON)) != 0 ||
+            _getCollateralBalance(address(SECONDARY_COLLATERAL_ADAPTER()), address(SHMON)) != 0 ||
+            _getDebtBalance(address(SECONDARY_DEBT_ADAPTER()), address(STABLECOIN)) != 0
+        ) revert InvalidParams();
+    }
+
     /// @notice Deploy and register a new dedicated WMON unwrapper for this strategy proxy.
     /// @dev Intended for existing upgraded proxies that did not get an unwrapper during initial deployment.
 ... 31 unchanged lines ...
```

BaseStrategy.sol

File: `src/strategies/BaseStrategy.sol`

Source

```
 ... 929 unchanged lines ...
     }

+    /// @dev Child strategies may enforce safety checks (e.g., zero positions) before adapter rewiring.
+    function _beforeAdaptersUpdated(address, address, address, address) internal view virtual { }
+
     /// @dev Shared base-state migration helper used by both calldata and memory upgrade payloads.
     function _applyMigratedBaseState(
         address primaryCollateralAdapter,
         address primaryDebtAdapter,
         address secondaryCollateralAdapter,
         address secondaryDebtAdapter,
         address wmonUnwrapper
     )
         internal
     {
         if (
             primaryCollateralAdapter == address(0) || primaryDebtAdapter == address(0)
                 || secondaryCollateralAdapter == address(0) || secondaryDebtAdapter == address(0)
         ) {
             revert InvalidParams();
         }

+        _beforeAdaptersUpdated(primaryCollateralAdapter, primaryDebtAdapter, secondaryCollateralAdapter, second
+
         BaseStrategyStorageLib.BaseStrategyStorage storage $ = _getBaseStrategyStorage();
```

```
        address previousWMonUnwrapper = $.wmonUnwrapper;
... 443 unchanged lines ...
```

19. [Low] maxWithdraw/onWithdraw mismatch in BaseStrategy under queue-disabled GatedVaultImpl causes withdrawal DoS

Status

Review status: Unresolved
Remediation status: Unremediated
Remediation note: Created by pipeline analysis

Description

Async-only strategies report positive maxWithdraw but always revert on onWithdraw. When GatedVaultImpl's withdrawal queue is disabled, the vault's synchronous withdrawal path uses maxWithdraw and then calls onWithdraw, causing withdrawals to revert and liquidity previews to be misleading.

BaseStrategy.maxWithdraw() returns the strategy's mark-to-market value when canWithdrawNow() and valuation reads succeed, and 0 otherwise. However, BaseStrategy.onWithdraw(uint256) always reverts (e.g., "Async withdrawal - no immediate liquidity"). AusdStrategy inherits this behavior. GatedVaultImpl, when its withdrawal queue is disabled (isQueueActive = false), falls back to the standard synchronous ERC-4626 withdrawal path in ConcreteStandardVaultImpl._executeWithdraw. That path, when idle vault funds are insufficient, iterates strategies in deallocation order, uses IStrategyTemplate.maxWithdraw() to size a pull, and then calls IStrategyTemplate.onWithdraw() without try/catch. Because onWithdraw always reverts for these strategies, any attempt to source liquidity from them aborts the entire withdrawal. Meanwhile, previews and liquidity checks (_simulateWithdraw) sum strategies' maxWithdraw() and do not anticipate onWithdraw reverts, overstating available immediate liquidity in this configuration. The result is a configuration-dependent, admin-induced temporary DoS of withdrawals and misleading previews when the queue is disabled and async-only strategies remain in the deallocation order.

Severity

Impact Explanation: [Medium] Withdrawals can be significantly but temporarily unavailable (DoS) in the queue-disabled configuration. No direct loss of principal; the issue is reversible by admin action.

Likelihood Explanation: [Low] All scenarios require a trusted role to disable the withdrawal queue and maintain an incompatible deallocation order including async-only strategies; multiple admin-controlled preconditions must coincide.

Exploitation

Exploitation Scenarios:

Scenario 1.

Queue disabled with insufficient idle funds: User attempts a withdrawal that requires strategy liquidity. The vault reads a positive maxWithdraw() from an async-only strategy and then calls onWithdraw(), which reverts, aborting the entire withdrawal.

Preconditions / Assumptions

- (a). GatedVaultImpl is deployed with the withdrawal queue disabled (isQueueActive = false)
- (b). AusdStrategy (or any BaseStrategy child with reverting onWithdraw) is active in the vault's deallocation order and holds allocated funds
- (c). Vault idle balance is less than the user's requested withdrawal
- (d). BaseStrategy.maxWithdraw() returns a positive amount (cooldowns passed and valuation reads succeed)
- (e). No other strategies cover the shortfall before reaching the async-only strategy

Scenario 2.

Mixed strategies with partial liquidity: The vault deallocates some funds from an earlier sync-capable strategy, then reaches an async-only strategy that reports positive maxWithdraw(); on calling onWithdraw() the transaction reverts, rolling back the whole withdrawal.

Preconditions / Assumptions

- (a). Queue disabled (isQueueActive = false)
- (b). Deallocation order includes at least one sync-capable strategy followed by an async-only strategy (e.g., AusdStrategy)
- (c). Combined idle funds and earlier strategies' liquidity are insufficient for the requested withdrawal
- (d). Async-only strategy's maxWithdraw() returns a positive amount

Scenario 3.

Misleading previews and failed attempts: With the queue disabled, previews indicate sufficient liquidity based on maxWithdraw(), but the actual withdrawal path reaches onWithdraw() and reverts, causing failed user transactions and gas loss.

Preconditions / Assumptions

- (a). Queue disabled (isQueueActive = false)
- (b). Async-only strategy is present in the deallocation order and reports positive maxWithdraw()
- (c). User or integrator relies on vault previews (e.g., maxWithdraw(owner), previewWithdraw/previewRedeem) before attempting withdrawal

Proposed fix

BaseStrategy.sol

File: `src/strategies/BaseStrategy.sol`

Source

```
 ... 412 unchanged lines ...
     /// @inheritdoc IStrategyTemplate
     function maxWithdraw() external view returns (uint256) {
-        try this.canWithdrawNow() returns (bool canWithdraw) {
-            if (!canWithdraw) return 0;
-        } catch {
-            return 0;
-        }
-
-        try this.getCurrentValue() returns (uint256 currentValue) {
-            return currentValue;
-        } catch {
-            return 0;
-        }
+        return 0;
     }

 ... 208 unchanged lines ...
     /// @inheritdoc IStrategyTemplate
     function onWithdraw(uint256) external view onlyVault returns (uint256) {
-        revert("Async withdrawal - no immediate liquidity");
+        return 0;
     }

 ... 753 unchanged lines ...
```

20. [Low] Token-based reserve cap check in BaseStrategy._withdrawCollateral allows bypass causing excess collateral withdrawal and yield drag

Status

Review status: Unresolved
Remediation status: Unremediated
Remediation note: Created by pipeline analysis

Description

BaseStrategy._withdrawCollateral caps deallocation withdrawals only when the provided token equals the vault asset, but both Euler and Curvance collateral adapters ignore the asset parameter and always act on their configured market. A privileged batch can pass a non-asset token to skip the cap while still withdrawing the vault asset, leading to larger-than-needed withdrawals, stranded idle capital, and yield drag. This is a privileged-operator footgun rather than an external exploit.

In BaseStrategy._withdrawCollateral, the deallocation reserve clamp (intended to "withdraw only what is needed") is enforced only if token == this.asset(). The collateral adapters (EulerCollateralAdapter and CurvanceCollateralAdapter) ignore the asset parameter in getMaxSafeWithdrawal() and withdraw(), always operating on their fixed configured collateral/debt markets. Consequently, a deallocation batch can supply any non-asset token to bypass the reserve cap, yet the adapter will still withdraw the real vault asset collateral. The withdrawal amount is bounded by adapter LTV/oracle safety logic, but it can exceed the intended small deallocation need. Excess withdrawn vault asset remains idle on the strategy, causing avoidable yield drag until returned. On Curvance, any withdrawal also triggers a cooldown; while this cooldown is not caused by the bypass itself, the bypass can exacerbate inefficiency by stranding extra idle assets that cannot be returned in subsequent deallocation calls until the cooldown expires. Only the vault/operator can execute batches (onlyVault) and wrappers are onlySelf, so this is a privileged-operator safety gap rather than a permissionless exploit.

Severity

Impact Explanation: [Medium] Excess collateral withdrawal leads to direct, material loss of yield due to idle capital stranded on the strategy until it is returned; Curvance cooldown can prolong inefficiency (though cooldown itself is not unique to the bypass).

Likelihood Explanation: [Low] Exploitation requires a trusted role (vault/operator) to submit a malformed or malicious batch; onlyVault and onlySelf gating prevent permissionless triggering.

Exploitation

Exploitation Scenarios:

Scenario 1.

Curvance primary collateral: A vault-controlled deallocation batch uses withdrawCollateral with a spoofed non-asset token and amount=0 (safe max), skipping the reserve cap. The adapter withdraws a large amount of stablecoin collateral (bounded by LTV). The batch returns only a small target amount to the vault, leaving excess idle on the strategy. Curvance’s cooldown (triggered by any withdrawal) then prevents immediate further deallocations, so the excess idle cannot be returned via deallocation until cooldown passes, causing yield drag.

Preconditions / Assumptions

- (a). AusdStrategy configured with CurvanceCollateralAdapter as the PRIMARY collateral leg
- (b). Strategy holds stablecoin collateral on Curvance (and may have WMON debt)
- (c). Vault/operator has authority to call deallocateFunds and craft batches (onlyVault)
- (d). Operator submits withdrawCollateral with token != asset(), amount=0, permissive targetLtv, and a small reserveAmount, followed by a partial returnIdleToVault

Scenario 2.

Euler primary collateral: A vault-controlled deallocation batch uses withdrawCollateral with a spoofed non-asset token and amount=0 (safe max), skipping the reserve cap. Euler’s adapter withdraws more stablecoin than needed into the strategy. The batch returns only the small target amount; the remainder sits idle, reducing yield until later returned.

Preconditions / Assumptions

- (a). AusdStrategy configured with EulerCollateralAdapter as the PRIMARY collateral leg
- (b). Strategy holds stablecoin collateral on Euler (and may have WMON debt)
- (c). Vault/operator has authority to call deallocateFunds and craft batches (onlyVault)
- (d). Operator submits withdrawCollateral with token != asset(), amount=0, permissive targetLtv, and a small reserveAmount, followed by a partial returnIdleToVault

Scenario 3.

Mixed legs operational drift: A vault-controlled deallocation batch bypasses the reserve cap on a Curvance collateral withdrawal, unintentionally over-withdrawing and leaving excess idle stablecoin. Subsequent planned deallocations are delayed by Curvance's cooldown while idle capital remains unproductive, degrading expected returns until operators return it.

Preconditions / Assumptions

- (a). AusdStrategy with Curvance and/or Euler legs deployed and looped
- (b). Vault/operator crafts deallocation batches (onlyVault)
- (c). Operator submits withdrawCollateral with token != asset(), amount=0, permissive targetLtv, followed by returning only a portion of idle to the vault
- (d). Curvance cooldown applies after the withdrawal, delaying further deallocations

Proposed fix

AusdStrategy.sol

File: `src/strategies/AusdStrategy.sol`

Source

```
 ... 863 unchanged lines ...
     }

+    function _isVaultAssetWithdrawal(ProtocolPosition position, address) internal view override returns (bool)
     function _deployMigratedWMonUnwrapper() internal override returns (address) {
         return address(new WMonUnwrapper(address(this), address(WMON)));
 ... 321 unchanged lines ...
```

21. [Low] Permissionless Permit2 deposit entrypoint after ungating in GatedVaultImpl causes forced execution of user deposits

Status

Review status: Unresolved
Remediation status: Unremediated
Remediation note: Created by pipeline analysis

Description

After reopening deposits, depositWithPermit2 becomes permissionless, allowing any holder of an unexpired user signature to force-execute a deposit at a time of their choosing. This creates involuntary lock-in and adverse-timing exposure for users, even though shares are minted to the owner and minShares is enforced.

In GatedVaultImpl, the onlyGatedRole modifier enforces role checks only while depositsGated is true. Once governance calls setDepositsGated(false), depositWithPermit2 is callable by anyone. The Permit2 witness binds only the owner, assets, and minShares; it does not bind a relayer or any epoch tied to gating. As a result, previously collected user signatures remain valid until deadline or nonce invalidation and can be executed by any party post-ungating. While funds and shares are not stolen (shares are minted to the owner and minShares is enforced at execution), users lose control over timing and can be involuntarily exposed to the vault's strategies and non-instant withdrawal lifecycle, potentially causing material yield loss and opportunity costs.

Severity

Impact Explanation: [Medium] Forced execution can cause involuntary exposure to strategy behavior and non-instant exits, leading to direct, material loss of yield for victims; this is user-level harm rather than protocol-wide unavailability.

Likelihood Explanation: [Low] Exploitation requires possession or misuse/leak of unexpired user signatures by a trusted party or third party, and often offers limited direct profit, aligning with low-likelihood griefing conditions.

Exploitation

Exploitation Scenarios:

Scenario 1.

Scenario 1: After deposits are ungated, a third party holding a victim's still-valid Permit2 signature calls depositWithPermit2, pulling the victim's assets into the vault and minting shares to the victim. The victim must now follow the asynchronous withdrawal lifecycle to exit, incurring lock-in and exposure they did not intend at that time.

Preconditions / Assumptions

- (a). Governance has set depositsGated(false), making depositWithPermit2 permissionless
- (b). Victim previously signed a Permit2 deposit (assets A, minShares M, unexpired deadline) and did not invalidate the nonce
- (c). A third party (e.g., ex-manager/relayer or leak) possesses the unexpired signature
- (d). The vault uses an asynchronous withdrawal lifecycle (request/processing/claim)

Scenario 2.

Scenario 2: During a gated rollout many users signed long-deadline permits. Managers are later rotated/removed, and then deposits are ungated. An ex-manager or anyone who retained those signatures executes them in batch via depositWithPermit2, forcing many victims into the vault simultaneously without their active consent at that time.

Preconditions / Assumptions

- (a). Signatures with generous deadlines were collected during the gated period
- (b). Managers are rotated/removed; later, governance ungates deposits
- (c). An ex-manager or third party retained the unexpired signatures; victims did not invalidate nonces
- (d). depositWithPermit2 is permissionless after ungating

Scenario 3.

Scenario 3: An attacker monitors strategy operations and, holding a victim's still-valid Permit2 signature, executes depositWithPermit2 immediately before a foreseeable adverse strategy event. The execution satisfies minShares at that moment, but the subsequent event causes losses, leaving the victim with material yield loss and non-instant exit.

Preconditions / Assumptions

- (a). Deposits are ungated; depositWithPermit2 is permissionless
- (b). Victim's Permit2 signature is still valid and in the attacker's possession
- (c). A foreseeable adverse strategy event is imminent
- (d). The live conversion at execution time meets or exceeds minShares

Proposed fix

GatedVaultImpl.sol

File: `src/GatedVaultImpl.sol`

Source

```
 ... 125 unchanged lines ...
     error DepositsAreGated();

+    /// @notice Raised when Permit2 deposit entrypoint is used after public ERC-4626 deposits are reopened.
+    error Permit2PathDisabledWhenUngated();
+
     /// @notice Raised when `depositWithPermit2` would mint fewer shares than the user signed for.
     /// @param shares Shares computed from the live vault state at execution time.
 ... 528 unchanged lines ...
         returns (uint256 shares)
     {
+        if (!_getGatedVaultStorage().depositsGated) revert Permit2PathDisabledWhenUngated();
+
         // Snapshot total assets before deposit for consistent share pricing.
```

```
        uint256 totalAssetsBeforeDeposit = cachedTotalAssets();
... 87 unchanged lines ...
```

22. [Low] Missing hooks==0 enforcement in reward-swap PoolKey validation (Swapper/RewardsAdapter) causes DoS of VAULT_ASSET reward swaps

Status

Review status: Unresolved
Remediation status: Unremediated
Remediation note: Created by pipeline analysis

Description

Reward-swap configuration validates only the MON/stable pair and a price limit, not that poolKey.hooks == address(0). Governance can configure a hook-enabled pool; during Swapper.unlockCallback, PoolManager.swap then executes hook code. A reverting hook causes the reward-to-vault-asset swap to revert (DoS). Reentrancy is constrained by strict role/self-call gates and PoolManager sender checks; no clear fund-loss or privilege-escalation path is evident.

RewardsAdapter.setRewardSwapConfig stores the provided Uniswap v4 PoolKey after Swapper.validateRewardSwapConfig, which only enforces currency0 == native MON, currency1 == stableAsset, and a non-permissive sqrt price limit. Swapper.validateRewardSwapConfig does not enforce hooks == address(0), and _validateRewardPool only checks the currency pair. When processing rewards to the vault asset, Swapper.processRewards calls PoolManager.unlock, which leads to RewardsAdapter.unlockCallback and then Swapper.unlockCallback invokes PoolManager.swap with the configured poolKey. If poolKey.hooks != 0, Uniswap v4 hook code executes during the swap. A reverting hook causes the swap and the entire reward processing to revert, creating a liveness/DoS issue for vault-asset reward conversion until governance reconfigures the pool. Reentrancy attempts from hooks are blocked by unlockActive gating, msg.sender == TRUSTED_POOL_MANAGER checks, and onlySelf/onlyVault/onlyStrategyAdmin/onlyRewardsManager gates on external functions. minAmountOut and BalanceDelta checks constrain swap outcomes. The primary impact is operational (DoS of the VAULT_ASSET reward-swap path), not principal loss.

Severity

Impact Explanation: [Medium] A reverting or unstable hook breaks an important but non-core function (reward conversion to the vault asset), causing significant but temporary availability loss/DoS of that path. Principal is not at risk and alternatives (WMON/MON outputs or reconfiguration) exist.

Likelihood Explanation: [Low] All scenarios require trusted admin misconfiguration (choosing a hook-enabled pool) and, in some cases, specific hook behavior (revert/instability). Attackers cannot set the poolKey. Thus, the issue relies on a trusted role mistake or integration behavior.

Exploitation

Exploitation Scenarios:

Scenario 1.

Governance configures a hook-enabled MON/stable pool where the hook reverts; operator calls processRewards with VAULT_ASSET output; during Swapper.unlockCallback the PoolManager.swap triggers the hook, which reverts, causing the reward swap to fail and blocking vault-asset reward conversion until reconfiguration.

Preconditions / Assumptions

- (a). Strategy admin sets RewardSwapConfig with poolKey.hooks != address(0) targeting a MON -> stableAsset pool
- (b). The pool's hook reverts in beforeSwap/afterSwap
- (c). Operator (onlyRewardsManager) calls processRewards with outputAsset = VAULT_ASSET
- (d). PoolManager follows canonical v4 semantics and is trusted

Scenario 2.

Governance configures a hook-enabled pool whose hook attempts reentrancy during PoolManager.swap; hook calls back into strategy functions, but unlockActive gating, msg.sender checks, and role/self-call restrictions prevent privilege

escalation; at worst, gas grief or revert occurs.

Preconditions / Assumptions

- (a). Strategy admin sets RewardSwapConfig with poolKey.hooks != address(0)
- (b). The pool's hook attempts reentrancy into strategy contracts during swap
- (c). Operator calls processRewards with outputAsset = VAULT_ASSET
- (d). Hook does not possess strategy roles and is not the PoolManager

Scenario 3.

Governance configures a hook-enabled pool with a gas-heavy or flaky hook; operator calls processRewards for VAULT_ASSET; swaps intermittently fail or incur higher gas due to hook behavior, degrading reward-processing reliability without fund loss.

Preconditions / Assumptions

- (a). Strategy admin sets RewardSwapConfig with poolKey.hooks != address(0)
- (b). The pool's hook is gas-heavy or occasionally reverts
- (c). Operator calls processRewards with outputAsset = VAULT_ASSET

Proposed fix

Swapper.sol

File: `src/adapters/rewards/Swapper.sol`

Source

```
 ... 102 unchanged lines ...
     /// @param currency1 Actual `currency1` address resolved from the pool key.
     error InvalidRewardPool(address currency0, address currency1);
+    /// @notice Revert when the supplied reward pool has non-zero hooks; reward swaps must use hookless pools.
+    error NonHooklessRewardPool(address hooks);
     /// @notice Revert when Uniswap reports a non-negative MON settlement delta.
     /// @param amount0 Unexpected `currency0` delta returned by the swap.
 ... 197 unchanged lines ...

     function _validateRewardPool(IUniswapV4.PoolKey memory poolKey, address stableAsset) private pure {
+        if (address(poolKey.hooks) != address(0)) {
+            revert NonHooklessRewardPool(address(poolKey.hooks));
+        }
         address currency0 = IUniswapV4.Currency.unwrap(poolKey.currency0);
         address currency1 = IUniswapV4.Currency.unwrap(poolKey.currency1);
         if (currency0 != address(0) || currency1 != stableAsset) {
             revert InvalidRewardPool(currency0, currency1);
         }
     }

     function _unwrapWMon(address wmon, address wmonUnwrapper, uint256 amount) private returns (uint256 monRecei
         if (amount == 0) return 0;
         if (wmonUnwrapper == address(0)) revert WMonUnwrapperNotConfigured();

         IERC20(wmon).safeTransfer(wmonUnwrapper, amount);
         monReceived = IWMonUnwrapper(wmonUnwrapper).unwrapToStrategy(amount);
     }
 }
```

Related findings

[Informational] Missing NatSpec for wmonUnwrapper and rewardSwapConfig in Swapper.processRewards causes avoidable reverts and delayed reward processing

Description

Swapper.processRewards omits @param docs for wmonUnwrapper and rewardSwapConfig despite using them for VAULT_ASSET reward conversions. This can mislead operators/admins, causing safe reverts and delayed reward conversion without asset loss.

Swapper.processRewards accepts five parameters (wmon, wmonUnwrapper, stableAsset, params, rewardSwapConfig) but its NatSpec documents only three, omitting wmonUnwrapper and rewardSwapConfig. These two parameters are required for the VAULT_ASSET path: wmonUnwrapper is used to unwrap WMON into MON, and rewardSwapConfig provides the trusted Uniswap v4 pool key and price bound. The function is internal and called by RewardsAdapter, which injects these values from strategy storage, and the code includes strict runtime checks that safely revert on misconfiguration (e.g., RewardSwapConfigNotSet, UnexpectedRewardPoolKey, WMonUnwrapperNotConfigured). The practical impact is operational: avoidable reverts, wasted gas, and delayed conversion of rewards that may remain as WMON longer than intended. No funds are lost and the trusted swap flow is guarded by unlock gating and PoolManager sender validation.

Severity

Impact Explanation: [Low] No principal loss or frozen funds; strict checks revert safely on misconfiguration. Impact is limited to operational friction: reverts, gas waste, and delayed reward conversion.

Likelihood Explanation: [Low] All scenarios require trusted-role (admin/operator) mistakes or misconfiguration; these are plausible but not common and are multiplicative preconditions.

Exploitation

Exploitation Scenarios:

Scenario 1.

Operator attempts VAULT_ASSET reward conversion without rewardSwapConfig set; Swapper.processRewards reverts RewardSwapConfigNotSet; rewards remain as WMON; gas is wasted and conversion is delayed.

Preconditions / Assumptions

- (a). Strategy-admin has not set rewardSwapConfig (sqrtPriceLimitX96 == 0).
- (b). Rewards manager calls RewardsAdapter.processRewards with outputAsset == VAULT_ASSET.
- (c). Operators rely on incomplete parameter documentation for Swapper.processRewards.

Scenario 2.

Operator supplies params.poolKey that does not match the admin-configured trusted poolKey; Swapper.processRewards reverts UnexpectedRewardPoolKey; rewards remain as WMON and conversion is delayed.

Preconditions / Assumptions

- (a). Strategy-admin has set a valid rewardSwapConfig with a trusted poolKey.
- (b). Rewards manager calls RewardsAdapter.processRewards with outputAsset == VAULT_ASSET and a mismatched params.poolKey.
- (c). Operators rely on incomplete parameter documentation regarding poolKey matching.

Scenario 3.

Admin misconfigures the wmonUnwrapper address; VAULT_ASSET conversion triggers _unwrapWMon and the unwrap helper reverts; the transaction fails and rewards remain as WMON, delaying conversion.

Preconditions / Assumptions

- (a). Strategy-admin misconfigures or sets an incorrect wmonUnwrapper address.
- (b). Rewards manager calls RewardsAdapter.processRewards with outputAsset == VAULT_ASSET, causing _unwrapWMon to execute.
- (c). Operators rely on incomplete parameter documentation about the unwrapper requirement.

Proposed fix

Swapper.sol

File: `src/adapters/rewards/Swapper.sol`

Source

```
 ... 162 unchanged lines ...
     ///      3. either keep it as WMON, unwrap it to MON, or swap the excess MON into the vault asset.
     /// @param wmon Wrapped MON token used by the strategy.
+    /// @param wmonUnwrapper Helper used to unwrap WMON into native MON for VAULT_ASSET swaps; must be configur
     /// @param stableAsset Vault asset received from a reward swap.
     /// @param params Reward post-processing parameters.
+    /// @param rewardSwapConfig Admin-configured trusted pool key and price bound for MON -> vault-asset swaps;
     /// @return outputAmount Amount of the selected output asset produced.
+    /// @dev For VAULT_ASSET output, params.poolKey must equal rewardSwapConfig.poolKey; otherwise reverts.
     function processRewards(
         address wmon,
 ... 151 unchanged lines ...
```

[Low] Hardcoded native-currency sentinel and fixed pool ordering in Uniswap v4 reward swap integration causes VAULT_ASSET reward-swap unavailability

Description

The reward-swap logic assumes native MON is address(0), enforces a fixed pool ordering (currency0 = native, currency1 = vault asset), always swaps zeroForOne = true, and settles via native-value. If Monad's PoolManager uses a different native sentinel or pool layout, configuration and execution of VAULT_ASSET reward swaps fail closed (revert). A similar assumption in the v4 shMON/WMON swap path can also cause unavailability. Funds are safe due to strict validation and trusted-callback gating.

Swapper validates the Uniswap v4 pool key by requiring currency0 == address(0) (native MON) and currency1 == the vault asset, then always executes swaps with zeroForOne = true and settles via settle{value: ...}(). setRewardSwapConfig enforces this at configuration time, and unlockCallback re-checks it before swapping. If the environment uses a different native-currency sentinel or the pool ordering differs, setRewardSwapConfig reverts, leaving RewardSwapConfig unset; subsequent processRewards with VAULT_ASSET output reverts RewardSwapConfigNotSet. ShMonSwapAdapter similarly assumes native = address(0) and rejects ERC20-ERC20 v4 pools for shMON/WMON. These checks prevent wrong-side swaps and asset loss, but cause a DoS of these swap features when the environment diverges from the assumptions.

Severity

Impact Explanation: [Medium] Breaks an important non-core feature (integrated VAULT_ASSET reward swaps and v4 shMON/WMON swaps), requiring alternative operational paths.

Likelihood Explanation: [Low] Depends on environmental divergence (native sentinel or pool ordering) outside attacker control; canonical Uniswap v4 semantics would avoid the issue.

Exploitation

Exploitation Scenarios:

Scenario 1.

VAULT_ASSET reward swap unavailable when the PoolManager represents native MON with a nonzero sentinel or supports only ERC20-based pools (e.g., WMON): setRewardSwapConfig reverts due to pool validation; later processRewards(VAULT_ASSET) reverts RewardSwapConfigNotSet.

Preconditions / Assumptions

- (a). Monad Uniswap v4 PoolManager uses a native MON sentinel other than address(0) or does not support native currency in the pool (ERC20-only).
- (b). A MON/stable pool exists under these conventions.
- (c). Admin attempts to configure the reward swap pool and operators later request processRewards with VAULT_ASSET output.

Scenario 2.

VAULT_ASSET reward swap unavailable when the valid MON/stable pool exists with reversed ordering (currency0 = stable, currency1 = native): pool validation rejects the key, configuration fails, and later VAULT_ASSET swaps revert due to unset config.

Preconditions / Assumptions

- (a). A valid MON/stable pool exists but is ordered with currency0 = stable and currency1 = native MON.
- (b). Admin attempts to configure the reward swap pool and operators later request processRewards with VAULT_ASSET output.

Scenario 3.

Unavailability of the dedicated Uniswap v4 shMON/WMON swap when the v4 market is ERC20-ERC20 or uses a different native sentinel: ShMonSwapAdapter pool validation reverts; operators must use the v3 path or existing stake/unstake flows instead.

Preconditions / Assumptions

- (a). The Uniswap v4 shMON/WMON market is ERC20-ERC20 or uses a native sentinel different from address(0).
- (b). Operators attempt to execute the v4 shMON/WMON swap command.

Proposed fix

Swapper.sol

File: `src/adapters/rewards/Swapper.sol`

Source

```
 ... 247 unchanged lines ...
         _validateRewardPool(callbackData.poolKey, callbackData.stableAsset);

+        // TODO: Generalize: derive zeroForOne from configured MON currency vs poolKey ordering, and support ER
+
         IUniswapV4.BalanceDelta delta = IPoolManager(TRUSTED_POOL_MANAGER)
             .swap(
 ... 51 unchanged lines ...
     }

+    // IMPORTANT: This assumes native MON == address(0) and fixed (currency0=native, currency1=stable) ordering
+    // TODO: Accept either ordering and parameterize MON currency (native or ERC20) to avoid environment coupli
     function _validateRewardPool(IUniswapV4.PoolKey memory poolKey, address stableAsset) private pure {
         address currency0 = IUniswapV4.Currency.unwrap(poolKey.currency0);
         address currency1 = IUniswapV4.Currency.unwrap(poolKey.currency1);
         if (currency0 != address(0) || currency1 != stableAsset) {
             revert InvalidRewardPool(currency0, currency1);
         }
     }

     function _unwrapWMon(address wmon, address wmonUnwrapper, uint256 amount) private returns (uint256 monRecei
         if (amount == 0) return 0;
         if (wmonUnwrapper == address(0)) revert WMonUnwrapperNotConfigured();

         IERC20(wmon).safeTransfer(wmonUnwrapper, amount);
         monReceived = IWMonUnwrapper(wmonUnwrapper).unwrapToStrategy(amount);
     }
 }
```

ShMonSwapAdapter.sol

File: `src/common/ShMonSwapAdapter.sol`

Source

```
 ... 21 unchanged lines ...
     /// @dev Monad's trusted Uniswap v4 PoolManager singleton used elsewhere in this strategy stack.
     address private constant TRUSTED_POOL_MANAGER = 0x188d586Ddcf52439676Ca21A244753fA19F9Ea8e;
+    // NOTE: If deployment uses a different native sentinel or ERC20-ERC20 pool, parameterize monCurrency and a
     /// @dev Uniswap v4 represents native MON as the zero-address currency.
     address private constant NATIVE_MON = address(0);
 ... 261 unchanged lines ...
     }

+    // TODO: Do not assume WMON maps to address(0); parameterize native vs ERC20 and support ERC20-ERC20 v4 poo
     function _poolCurrencyForToken(address token, address wmon) private pure returns (address) {
         return token == wmon ? NATIVE_MON : token;
 ... 29 unchanged lines ...
```

[Low] Missing upper bound validation on Uniswap v4 sqrtPriceLimitX96 in reward swap configuration causes persistent DoS of VAULT_ASSET reward swaps

Description

Swapper.validateRewardSwapConfig enforces only a lower bound on sqrtPriceLimitX96 and allows out-of-domain values to be stored. Later, processRewards forwards this invalid limit into a Uniswap v4 swap, which reverts unconditionally under canonical semantics, persistently breaking VAULT_ASSET reward swaps until reconfigured.

RewardsAdapter.setRewardSwapConfig stores Swapper.RewardSwapConfig after Swapper.validateRewardSwapConfig checks only that sqrtPriceLimitX96 > MIN_SQRT_PRICE_PLUS_ONE and that the pool is (MON, vault asset). There is no upper-bound check (e.g., < MAX_SQRT_RATIO-1). This permits saving provably invalid values like type(uint160).max. When processRewards runs with outputAsset=VAULT_ASSET, Swapper.unlockCallback calls IPoolManager.swap with zeroForOne=true and the configured sqrtPriceLimitX96. Under canonical Uniswap v4 semantics, an out-of-domain sqrt price limit causes unconditional revert, so every such reward swap fails until governance fixes the configuration. No funds are lost, and WMON/MON outputs still function, but the VAULT_ASSET reward conversion path is persistently DoS'd by misconfiguration.

Severity

Impact Explanation: [Medium] Breaks important non-core functionality (VAULT_ASSET reward swaps) and may cause direct, material loss of yield due to inability to compound until reconfiguration.

Likelihood Explanation: [Low] Relies on a trusted admin misconfiguring sqrtPriceLimitX96 (trusted role mistake).

Exploitation

Exploitation Scenarios:

Scenario 1.

Admin sets sqrtPriceLimitX96 to type(uint160).max via setRewardSwapConfig; later, a rewards manager calls processRewards with outputAsset=VAULT_ASSET, triggering Swapper.unlockCallback -> IPoolManager.swap with the out-of-domain limit, causing unconditional revert and persistent DoS of VAULT_ASSET reward swaps until reconfiguration.

Preconditions / Assumptions

- (a). Canonical Uniswap v4 PoolManager semantics apply
- (b). Strategy admin has permission and sets reward swap config
- (c). PoolKey is configured with currency0 == address(0) and currency1 == the vault asset
- (d). Rewards manager later calls processRewards with outputAsset=VAULT_ASSET and matching poolKey

Scenario 2.

Admin sets sqrtPriceLimitX96 to a value above the canonical Uniswap v4 maximum (MAX_SQRT_RATIO-1) but below type(uint160).max; setRewardSwapConfig succeeds, and subsequent processRewards calls to VAULT_ASSET revert unconditionally in swap, causing persistent DoS until reconfiguration.

Preconditions / Assumptions

- (a). Canonical Uniswap v4 PoolManager semantics apply
- (b). Strategy admin has permission and sets reward swap config
- (c). PoolKey is configured with currency0 == address(0) and currency1 == the vault asset
- (d). Rewards manager later calls processRewards with outputAsset=VAULT_ASSET and matching poolKey

Proposed fix

Swapper.sol

File: `src/adapters/rewards/Swapper.sol`

Source

```
 ... 41 unchanged lines ...
     /// @dev Lowest valid sqrt price plus one, used to enforce a one-sided MON -> stable swap.
     uint160 private constant MIN_SQRT_PRICE_PLUS_ONE = 4_295_128_740;
+    uint160 private constant MAX_SQRT_PRICE_MINUS_ONE = 1_461_446_703_485_210_103_287_273_052_203_988_822_378_7

     /// @dev The only pool manager allowed to execute reward swaps for this strategy stack.
 ... 99 unchanged lines ...
     function validateRewardSwapConfig(RewardSwapConfig memory config, address stableAsset) internal pure {
         _validateRewardPool(config.poolKey, stableAsset);
-        if (config.sqrtPriceLimitX96 <= MIN_SQRT_PRICE_PLUS_ONE) {
+        if (config.sqrtPriceLimitX96 <= MIN_SQRT_PRICE_PLUS_ONE || config.sqrtPriceLimitX96 > MAX_SQRT_PRICE_MI
             revert InvalidRewardSqrtPriceLimit(config.sqrtPriceLimitX96);
         }
 ... 171 unchanged lines ...
```

23. [Low] Token-agnostic repay with stale unlimited approval in CurvanceDebtAdapter/BaseStrategy causes reserve-clamp bypass and unexpected underlying consumption

Status

Review status: Unresolved
Remediation status: Unremediated
Remediation note: Created by pipeline analysis

Description

CurvanceDebtAdapter.repay approves the caller-supplied token and sets unlimited allowance on amount==0, while BaseStrategy._repay sizes and reserve-clamps using the same caller token. If a prior full-repay left a persistent unlimited allowance on the true Curvance underlying, a later wrong-token repay can still pull the real underlying via the Curvance market, bypassing the reserve clamp and unexpectedly consuming idle balances (including the vault asset when Curvance is used for stablecoin debt).

BaseStrategy._repay measures amountAvailable as the balance of the passed token and applies a reserve clamp only when that token equals the vault asset; it then staticcalls the adapter to resolve the repay instruction and finally delegatecalls adapter.repay(token, amount/sentinel). CurvanceDebtAdapter.resolveRepayInstruction ignores the token and compares amountAvailable vs currentDebt without unit conversion, potentially selecting a full-repay sentinel or a partial amount based on wrong-token balances. CurvanceDebtAdapter.repay uses the passed token only to set ERC20 allowance (type(uint256).max when amount==0) and then calls DEBT_TOKEN.repay(amount), which pulls the true underlying from the strategy via transferFrom. Because the adapter is called via delegatecall, approvals persist on the strategy. If a previous correct full-repay left unlimited approval on the true underlying, a subsequent wrong-token repay can still succeed and consume the true underlying (e.g., stablecoin), even though BaseStrategy sized and reserved based on a different token. This can bypass the reserve clamp during deallocation and reduce idle liquidity expected for withdrawals. Only the vault/operator can trigger this (not user-exploitable), and it additionally requires the stale unlimited allowance and sufficient underlying balance at the time of the wrong-token repay.

Severity

Impact Explanation: [Medium] Unexpected consumption of assets designated for reservation during deallocation can reduce or delay withdrawals and alter unwind behavior; this is a significant but temporary availability/liquidity impact rather than a direct principal loss.

Likelihood Explanation: [Low] Requires a trusted operator/governance to pass a wrong token and a specific prior state (stale unlimited allowance on the true underlying) plus sufficient underlying balance; only the vault/strategy can trigger these paths.

Exploitation

Exploitation Scenarios:

Scenario 1.

Reserve-clamp bypass on secondary Curvance stablecoin debt: a prior correct full-repay left unlimited stablecoin allowance; during deallocation the operator submits a repay with token=WMON and amount=0; BaseStrategy's reserve clamp does not apply (wrong token) and resolveRepayInstruction selects full-repay; Curvance DEBT_TOKEN uses the stale unlimited stablecoin approval to pull the full stablecoin debt, consuming idle stablecoin intended to be reserved for withdrawals.

Preconditions / Assumptions

- (a). Secondary debt leg uses Curvance with the vault asset (stablecoin) as underlying
- (b). A previous correct full-repay (amount==0) left unlimited stablecoin allowance to the specific Curvance DEBT_TOKEN
- (c). Strategy holds enough idle stablecoin to cover the visible debt
- (d). Operator submits a wrong-token repay (e.g., token=WMON) with amount=0
- (e). Only the vault/strategy can invoke repay (trusted operator)

Scenario 2.

Partial wrong-token repay drains stablecoin: with a stale unlimited stablecoin allowance present, the operator submits a repay on the secondary Curvance market with token=WMON; resolveRepayInstruction returns a partial toRepay equal to the wrong-token amountAvailable; Curvance DEBT_TOKEN pulls that amount of stablecoin via the stale unlimited approval, reducing idle stablecoin unexpectedly.

Preconditions / Assumptions

- (a). Secondary debt leg uses Curvance with stablecoin as underlying
- (b). A previous correct full-repay left unlimited stablecoin allowance to the Curvance DEBT_TOKEN
- (c). Curvance MIN_LOAN_SIZE allows a partial repay (remainingDebtUsd >= minLoanUsd)
- (d). Strategy holds at least the partial toRepay amount of stablecoin
- (e). Operator submits a wrong-token repay (e.g., token=WMON)

Scenario 3.

Wrong-token repay on primary Curvance WMON debt consumes WMON: a prior sentinel repay left unlimited WMON allowance; the operator mistakenly submits repay with token=stablecoin on the primary Curvance market; Curvance DEBT_TOKEN uses the stale unlimited WMON approval to pull WMON to reduce/close the debt, altering intended volatile-leg liquidity mid-batch.

Preconditions / Assumptions

- (a). Primary debt leg uses Curvance with WMON as underlying
- (b). A previous correct full-repay left unlimited WMON allowance to the Curvance DEBT_TOKEN
- (c). Strategy holds idle WMON
- (d). Operator submits a wrong-token repay (e.g., token=stablecoin) for the primary Curvance market

Proposed fix

CurvanceDebtAdapter.sol

File: `src/adapters/curvance/CurvanceDebtAdapter.sol`

Source

```
 ... 103 unchanged lines ...
     /// @inheritdoc IDebtAdapter
     function repay(address asset, uint256 amount) external {
+        require(asset == DEBT_TOKEN.asset(), "CurvanceDebtAdapter: wrong asset");
+
         // Use forceApprove instead of approve to handle tokens that require an existing
         // allowance to be cleared before a new one can be set.
         //
         // Curvance interprets amount == 0 as "repay full debt". In that case the debt token
         // decides the exact transfer amount after interest accrual, so approve max.
         uint256 approvalAmount = amount == 0 ? type(uint256).max : amount;

         IERC20(asset).forceApprove(address(DEBT_TOKEN), approvalAmount);
         DEBT_TOKEN.repay(amount);
+        IERC20(asset).forceApprove(address(DEBT_TOKEN), 0);

         // When amount == 0 the protocol repays the full accrued balance even though the emitted
 ... 193 unchanged lines ...
```

Related findings

[Low] Reserve clamp overridden by closeAllDebt in BaseStrategy._repay causes under-delivery of reserved assets to the vault during deallocation

Description

In BaseStrategy._repay, when token == asset() and closeAllDebt is true, the function overwrites the reserve-aware repay amount with the full available balance, allowing consumption of idle vault-asset that was intended to be reserved for withdrawals. Because the repay wrapper accepts both reserveAmount and closeAllDebt simultaneously, batches can unintentionally consume reserved funds, leading to under-delivery to the vault or forcing larger-than-intended unwinds.

The repay command accepts (position, token, amount, reserveAmount[, closeAllDebt]). In _repay, if token == asset() and reserveAmount > 0, the function first clamps the repay amount so only the excess above the reserve may be used. However, immediately afterward, if closeAllDebt is true, it sets amount = availableBalance, overriding the prior reserve clamp. The adapters (Curvance/Euler) rely on this amount as amountAvailable to authorize close-all, requiring only that amountAvailable >= currentDebt. As a result, when there is any excess above reserve, a close-all repay can consume reserved idle vault-asset. A built-in guard prevents this when idle <= reserve (amount becomes 0 and no repay occurs), and subsequent withdrawCollateral steps can restore idle to the reserve target, but if such a step is omitted or constrained, the vault may receive less than intended for the epoch. Only the vault can call deallocateFunds, so this is an operator/batch-construction footgun rather than an external exploit.

Severity

Impact Explanation: [Medium] Temporary but potentially significant under-delivery to the vault during a deallocation epoch can delay user withdrawals or require additional unwinds; however, it does not cause direct principal loss.

Likelihood Explanation: [Low] Exploitation requires trusted operator/batch misuse (combining closeAllDebt with a nonzero reserveAmount on a vault-asset repay and omitting compensating steps). External attackers cannot trigger deallocation.

Exploitation

Exploitation Scenarios:

Scenario 1.

Under-delivery to the vault: idle S = 1,000; reserveAmount = 800; secondary stablecoin debt = 500. Batch uses repay(token=S, amount=0, reserve=800, closeAllDebt=true) then returnIdleToVault(target=800) without a compensating withdraw. _repay overwrites the reserve clamp and adapter repays 500 from idle, leaving 500. Only 500 can be returned, under-delivering versus the intended 800.

Preconditions / Assumptions

- (a). Only the vault can call deallocateFunds (trusted operator).
- (b). Repay targets the vault asset (token == asset()).
- (c). Strategy idle vault-asset balance is greater than reserveAmount (some excess exists).
- (d). Secondary stablecoin debt is open.
- (e). Batch sets amount=0, reserveAmount>0, closeAllDebt=true on repay.
- (f). Batch omits a compensating withdrawCollateral step before returning idle to the vault.

Scenario 2.

Forced larger unwind to rebuild idle: same as above, but the batch includes a withdrawCollateral step after the repay. Because the reserve clamp in withdrawCollateral caps to stillNeeded, it withdraws 300 to restore idle to 800. The reserve target is met, but this causes a larger-than-intended unwind and potential added risk/cost.

Preconditions / Assumptions

- (a). Only the vault can call deallocateFunds (trusted operator).
- (b). Repay targets the vault asset (token == asset()).
- (c). Strategy idle vault-asset balance is greater than reserveAmount (some excess exists).
- (d). Secondary stablecoin debt is open.
- (e). Batch sets amount=0, reserveAmount>0, closeAllDebt=true on repay.
- (f). Batch includes a compensating withdrawCollateral step after the repay to restore idle to the reserve target.

Scenario 3.

Large shortfall for a higher reserve: idle S = 2,400; reserveAmount = 2,000; secondary S debt = 1,800. Batch uses repay(token=S, amount=0, reserve=2,000, closeAllDebt=true) then returnIdleToVault(target=2,000) without a compensating withdraw. Repay consumes 1,800 from idle, leaving 600; only 600 is returned, a severe shortfall for that epoch.

Preconditions / Assumptions

- (a). Only the vault can call deallocateFunds (trusted operator).
- (b). Repay targets the vault asset (token == asset()).
- (c). Strategy idle vault-asset balance is greater than reserveAmount (some excess exists).
- (d). Secondary stablecoin debt is open and sizable.
- (e). Batch sets amount=0, reserveAmount>0, closeAllDebt=true on repay.
- (f). Batch omits a compensating withdrawCollateral step; returnIdleToVault is called expecting the full reserve.

Proposed fix

BaseStrategy.sol

File: `src/strategies/BaseStrategy.sol`

Source

```
 ... 790 unchanged lines ...
                 amount = availableBalance;
             }
-            if (closeAllDebt) {
+            // Preserve reserve clamp for the vault asset during deallocation: do not override
+            // the clamped amount when a nonzero reserve applies to the core asset.
+            if (closeAllDebt && !(reserveAmount > 0 && token == this.asset())) {
                 amount = availableBalance;
             }
 ... 598 unchanged lines ...
```

24. [Low] Missing upgrade-time initialization of basisGuardBps in AusdStrategy upgrades causes allocation/deallocation DoS

Status

Review status: Unresolved
Remediation status: Unremediated
Remediation note: Created by pipeline analysis

Description

AusdStrategy seeds basisGuardBps (default 200 bps) only in initialize(). UUPS upgrades do not rerun initialize(), and there is no child-level migration to set the AUSD-specific storage. An upgraded proxy can retain basisGuardBps == 0. Since allocateFunds() enforces a basis guard and deallocateFunds() typically does as well, any positive basis will revert, causing an operational DoS until governance sets a nonzero guard via setBasisGuardBps().

AusdStrategy relies on an AUSD-specific storage field, basisGuardBps, to cap acceptable oracle basis during key operations. This field is set to 200 bps only in initialize(). During UUPS upgrades, initialize() is not called, and AusdStrategy does not override the base migration hook to seed the AUSD namespace. If an existing proxy is upgraded into this implementation without a post-upgrade call to setBasisGuardBps(), basisGuardBps remains 0. allocateFunds() always calls _checkBasisGuard(), which reverts when getBasisBps() > basisGuardBps; with guard == 0, any positive basis causes revert. deallocateFunds() performs a best-effort check (bypassed only if getBasisBps() reverts due to oracle failure); with healthy oracles, it will also revert. As a result, normal allocation and unwind flows are blocked until governance explicitly sets a nonzero guard using setBasisGuardBps(). This is an operational liveness issue (no direct principal loss) that depends on an admin omission after upgrades.

Severity

Impact Explanation: [Medium] This causes a significant but temporary availability loss of core strategy functions (allocation and deallocation), which can stall withdrawals and strategy operations until governance performs a straightforward corrective action.

Likelihood Explanation: [Low] The condition relies on a trusted admin omission (failing to set basisGuardBps after an upgrade). No external attacker action is required or sufficient to induce the state.

Exploitation

Exploitation Scenarios:

Scenario 1.

Withdrawals stall post-upgrade: After upgrading the strategy proxy and failing to set basisGuardBps, the vault attempts to deallocate to meet user withdrawals. deallocateFunds() computes a positive basis under healthy oracles and, with guard == 0, reverts (BasisOutOfRange), blocking deallocation until governance sets a nonzero guard.

Preconditions / Assumptions

- (a). A strategy proxy is upgraded to this AusdStrategy implementation without calling setBasisGuardBps() afterward
- (b). AusdStrategy has no child-level migration seeding basisGuardBps during upgrade
- (c). Oracles are healthy so getBasisBps() succeeds and returns a positive basis
- (d). Users have pending withdrawals requiring deallocation

Scenario 2.

Allocation/rebalance blocked post-upgrade: After upgrading without resetting basisGuardBps, the vault attempts to allocate or rebalance via allocateFunds(). _checkBasisGuard() sees a positive basis and guard == 0, causing a revert (BasisOutOfRange), preventing allocations until governance fixes the guard.

Preconditions / Assumptions

- (a). A strategy proxy is upgraded to this AusdStrategy implementation without calling setBasisGuardBps() afterward
- (b). AusdStrategy has no child-level migration seeding basisGuardBps during upgrade
- (c). Oracles are healthy so getBasisBps() returns a positive basis

Scenario 3.

Emergency unwind contingent on oracle failure: With guard == 0 post-upgrade, deallocateFunds() normally reverts under healthy oracles. Governance waits for a transient oracle failure (so getBasisBps() reverts) to bypass the guard and

proceed, creating an operational hazard where emergency unwinds depend on oracle outages rather than a reliable configuration.

Preconditions / Assumptions

- (a). A strategy proxy is upgraded to this AusdStrategy implementation without calling setBasisGuardBps() afterward
- (b). AusdStrategy has no child-level migration seeding basisGuardBps during upgrade
- (c). Oracles are healthy by default (so deallocation would revert), and later become stale/unavailable, causing getBasisBps() to revert and bypass the guard

Proposed fix

AusdStrategy.sol

File: `src/strategies/AusdStrategy.sol`

Source

```
 ... 407 unchanged lines ...
         if (basisResolved) {
             uint256 guardBps = AusdStrategyStorageLib.fetch().basisGuardBps;
+            if (guardBps == 0) guardBps = 200; // fallback to safe default on uninitialized/mistakenly zero gua
             if (basisBps > guardBps) revert BasisOutOfRange(basisBps, guardBps);
         }
 ... 57 unchanged lines ...
             if (preview == 0) revert ZeroExpectedShMonUnstake(toConvert);
             uint256 guardBps = AusdStrategyStorageLib.fetch().basisGuardBps;
-            uint256 floorMinOut = guardBps == 0 ? 0 : (preview * (LTVCalculations.BPS - guardBps)) / LTVCalcula
+            if (guardBps == 0) guardBps = 200; // fallback to safe default on uninitialized/mistakenly zero gua
+            uint256 floorMinOut = (preview * (LTVCalculations.BPS - guardBps)) / LTVCalculations.BPS;
             uint256 enforcedMinOut = minConversionOut > floorMinOut ? minConversionOut : floorMinOut;
             SwapAdapter.unstakeShMonToMon(SHMON, toConvert, enforcedMinOut);
 ... 433 unchanged lines ...
         uint256 basisBps = getBasisBps();
         uint256 guardBps = AusdStrategyStorageLib.fetch().basisGuardBps;
+        if (guardBps == 0) guardBps = 200; // fallback to safe default on uninitialized/mistakenly zero guard
         if (basisBps > guardBps) {
             revert BasisOutOfRange(basisBps, guardBps);
 ... 278 unchanged lines ...
```

25. [Low] Unbounded 10**decimals exponentiation in OracleHelpers and related paths causes revert/DoS under misconfigured decimals

Status

Review status: Unresolved
Remediation status: Unremediated
Remediation note: Created by pipeline analysis

Description

Several pricing/normalization helpers raise 10 to unbounded token/oracle decimals. For decimals >= 78 this overflows uint256 and reverts, blocking operations that rely on these helpers when governance misconfigures decimals.

OracleHelpers.convertToUsd/convertFromUsd compute 10**tokenDecimals and 10**priceDecimals without bounding inputs. AusdStrategy.getBasisBps computes 10**shDec similarly, and CompositeOracle._scaleCompositePrice uses _pow10 on decimal gaps. Since 10**x overflows uint256 for x >= 78 in Solidity 0.8+, any oversized decimals (e.g., from adapter constructor params, token .decimals(), or oracle.decimals()) cause deterministic reverts. While typical ERC-20/Chainlink decimals are <= 18, a governance misconfiguration or a custom oracle output precision > 77 will DoS key flows: allocation basis checks, partial Curvance repayments, and 'max safe' deallocation sizing. Some paths degrade gracefully (e.g., deallocation basis bypass via try/catch, ability to pass explicit amounts, Curvance full-repay sentinel), but standard operations can fail until configuration is corrected.

Severity

Impact Explanation: [Medium] Significant but temporary availability loss of important strategy functions (allocation basis guard, partial Curvance repayments, and adapter 'max safe' deallocation sizing). Not a complete protocol failure and workarounds exist (explicit amounts, full-repay sentinel, termination path).

Likelihood Explanation: [Low] Requires trusted role (governance/admin) misconfiguration of token/adapter/oracle decimals; typical ERC-20 and Chainlink feeds use <= 18 decimals.

Exploitation

Exploitation Scenarios:

Scenario 1.

Curvance partial repay reverts: DEBT_DECIMALS > 77 causes OracleHelpers.convertToUsd(remainingDebt, ...) to overflow in _calculateSafeRepayment, preventing partial repayments and increasing liquidation risk until fixed or a full-repay sentinel is used.

Preconditions / Assumptions

- (a). Admin misconfigures CurvanceDebtAdapter with DEBT_DECIMALS > 77 or uses a debt oracle with decimals > 77
- (b). Strategy has a nonzero Curvance WMON debt
- (c). Repay is called with closeAllDebt=false and available balance < currentDebt

Scenario 2.

Allocation blocked by basis guard: AusdStrategy.allocateFunds calls getBasisBps, which overflows on 10**shDec or in OracleHelpers when oracle decimals > 77, causing allocation to revert.

Preconditions / Assumptions

- (a). SHMON.decimals() > 77 or SHMON_ORACLE.decimals() > 77 or WMON_ORACLE.decimals() > 77 (e.g., CompositeOracle DECIMALS set too high)
- (b). Vault invokes AusdStrategy.allocateFunds (basis guard runs before commands)

Scenario 3.

Deallocation 'max safe withdrawal' fails: withdrawCollateral with amount==0 triggers adapter.getMaxSafeWithdrawal -> LTVCalculations -> OracleHelpers; oversized adapter/ oracle decimals overflow and revert, aborting deallocation batches unless explicit amounts are provided.

Preconditions / Assumptions

- (a). Euler/Curvance adapter configured with collateralDecimals_ or debtDecimals_ > 77, or relevant oracle.decimals() > 77
- (b). Deallocation uses withdrawCollateral with amount == 0 to request 'max safe' sizing at targetLtv

Proposed fix

CompositeOracle.sol

File: `src/oracle/CompositeOracle.sol`

Source

```
 ... 343 unchanged lines ...
     /// @return Ten raised to `value`
     function _pow10(uint256 value) private pure returns (uint256) {
+        require(value <= MAX_PYTH_EXPONENT);
         return DECIMAL_BASE ** value;
     }
 }
```

AusdStrategy.sol

File: `src/strategies/AusdStrategy.sol`

Source

```
 ... 838 unchanged lines ...
     function getBasisBps() public view returns (uint256 basisBps) {
         uint8 shDec = SHMON.decimals();
+        require(shDec <= 77);
         uint256 oneShare = 10 ** shDec;
         uint256 pps = SHMON.convertToAssets(oneShare);
 ... 346 unchanged lines ...
```

OracleHelpers.sol

File: `src/libraries/OracleHelpers.sol`

Source

```
 ... 56 unchanged lines ...
         (uint256 price, uint8 priceDecimals) = getPrice(oracle, heartbeat);

+        if (tokenDecimals > 77 || priceDecimals > 77) revert InvalidOraclePrice();
         // Normalize token units and oracle decimals into the protocol's 18-decimal USD unit.
         return (amount * price * USD_DECIMALS) / (DECIMAL_BASE ** tokenDecimals) / (DECIMAL_BASE ** priceDecima
 ... 25 unchanged lines ...
         (uint256 price, uint8 priceDecimals) = getPrice(oracle, heartbeat);

+        if (tokenDecimals > 77 || priceDecimals > 77) revert InvalidOraclePrice();
         // Undo the USD normalization and return the amount in token-native decimals.
         return (usdAmount * (DECIMAL_BASE ** tokenDecimals) * (DECIMAL_BASE ** priceDecimals)) / (price * USD_D
 ... 45 unchanged lines ...
```

Related findings

[Low] Unbounded DECIMALS and unsafe 10**d/int256 casting in CompositeOracle causes oracle revert and allocation DoS

Description

CompositeOracle accepts any DECIMALS and uses 10**d scaling and int256 casts without safe bounds. Combined with large decimal differences or extreme Pyth exponents, latestRoundData can revert (or yield nonsensical prices), which bubbles into OracleHelpers and blocks AusdStrategy allocations via the basis guard.

CompositeOracle exposes an unbounded DECIMALS (uint8) and rescales the product of the two legs using 10**d via _pow10. The EVM EXP returns results modulo 2^256, which are then cast to int256; for large d, the cast can revert (e.g., when the top bit is set), and for d ≥ 256 downscaling can hit division by zero. Separately, _normalizePythPrice allows positive exponents up to 77 and negative exponent magnitudes up to 255. In the positive-expo path, absPrice * 10**expo is only checked against uint256.max, not int256.max, so casting to int256 can revert. In the negative-expo path, the derived quoteDecimals can be extremely large, forcing d into unsafe regions. Any such revert in CompositeOracle.latestRoundData bubbles up to OracleHelpers.getPrice (as OracleCallReverted), and AusdStrategy's allocateFunds enforces a basis guard that depends on a shMON oracle price, causing allocation DoS. Deallocation paths may still proceed since they tolerate basis read failures.

Severity

Impact Explanation: [Medium] Allocation flows in AusdStrategy are blocked due to basis-guard oracle reads reverting, causing significant availability loss and yield impact. Deallocation can still proceed (with basis-guard bypass), and there is no direct principal loss.

Likelihood Explanation: [Low] Triggers require either a trusted admin misconfiguration of DECIMALS or unusual Pyth exponent values for USD pairs. These are rare/exceptional conditions rather than attacker-controlled paths.

Exploitation

Exploitation Scenarios:

Scenario 1.

MON/USD leg uses Pyth; Pyth returns expo = −255 with a valid price. _normalizePythPrice sets priceDecimals = 255. With a typical shMON/MON baseDecimals ≈ 8 and DECIMALS = 8, currentDecimals = 263 and downscaling uses d = 255. _pow10(255) produces a value ≥ 2^255 modulo 2^256, so casting to int256 reverts. CompositeOracle.latestRoundData reverts, OracleHelpers.getPrice reverts, and AusdStrategy.allocateFunds fails due to the basis guard.

Preconditions / Assumptions

- (a). CompositeOracle MON/USD leg configured with hasPyth = true
- (b). Pyth returns a valid price with expo = −255 (publishTime ≤ block.timestamp, rawPrice > 0)
- (c). shMON/MON leg returns valid data with baseDecimals ≈ 8
- (d). CompositeOracle DECIMALS = 8
- (e). Pyth source is invoked (it always is when hasPyth = true) and selected or causes revert during evaluation

Scenario 2.

Deployer configures DECIMALS = 93. With typical baseDecimals + quoteDecimals ≈ 16, upscaling uses d = 77. 10**77 fits in uint256 but exceeds int256.max, so casting to int256 during scaling reverts deterministically. CompositeOracle.latestRoundData always reverts, OracleHelpers.getPrice reverts, and AusdStrategy.allocateFunds is blocked by the basis guard until a corrected oracle is deployed and wired.

Preconditions / Assumptions

- (a). CompositeOracle deployed with DECIMALS = 93
- (b). Both legs provide typical decimals leading to baseDecimals + quoteDecimals ≈ 16
- (c). AusdStrategy configured to use this CompositeOracle as its shMON oracle (directly or through wiring)

Scenario 3.

MON/USD leg uses Pyth; Pyth returns expo in [0..77], e.g., expo = 77 and rawPrice = 1. In _normalizePythPrice, absPrice * 10**expo passes the uint256 overflow check but exceeds int256.max; casting to int256 reverts. Since _getPythSourceData is invoked unconditionally when hasPyth is true, latestRoundData reverts even if Chainlink sources are valid, causing OracleHelpers.getPrice to revert and AusdStrategy.allocateFunds to fail.

Preconditions / Assumptions

- (a). CompositeOracle MON/USD leg configured with hasPyth = true
- (b). Pyth returns expo in [0..77] with rawPrice > 0 (e.g., expo = 77, rawPrice = 1)
- (c). _normalizePythPrice's positive-exponent path computes absPrice * 10**expo > int256.max
- (d). _getPythSourceData is called regardless of Chainlink validity and normalization reverts

Proposed fix

CompositeOracle.sol

File: `src/oracle/CompositeOracle.sol`

Source

```
 ... 124 unchanged lines ...
             revert InvalidPythConfig();
         }
+        require(decimals_ <= 18, "CompositeOracle: decimals too large");

         SHMON_MON_SOURCE_0 = AggregatorV3Interface(shMonMonSource0_);
 ... 164 unchanged lines ...
             uint256 absPrice = uint256(uint64(rawPrice));
             if (absPrice > type(uint256).max / scale) return (0, 0, false);
+            if (absPrice > uint256(type(int256).max) / scale) return (0, 0, false);

             price = int256(absPrice * scale);
```

```
            priceDecimals = 0;
            return (price, priceDecimals, true);
        }

        uint256 magnitude = uint256(uint32(-expo));
        if (magnitude > type(uint8).max) return (0, 0, false);
+       if (magnitude > 38) return (0, 0, false);

        price = int256(rawPrice);
 ... 44 unchanged lines ...
```

[Low] No short-circuit and unbounded decimal scaling in CompositeOracle external oracle calls causes DoS of price reads and strategy operations

Description

CompositeOracle queries all configured sources without short-circuiting and forwards essentially all gas to each, enabling gas-grief DoS if any source misbehaves. Additionally, when Pyth is enabled, unbounded negative exponents can inflate effective decimals so 10^k overflows during scaling, reverting even with otherwise valid data. These failures propagate to AusdStrategy via OracleHelpers, blocking allocation and valuation-dependent flows.

CompositeOracle's _getLatestLegData() always queries both configured Chainlink-style sources (and Pyth if present) and selects the freshest valid candidate by updatedAt. It does not short-circuit after finding a valid source. External calls (latestRoundData() and decimals()) forward essentially all remaining gas; a misbehaving/malicious source can consume gas and revert, leaving the caller with only ~1/64 gas (EIP-150). As the function continues to call remaining sources and perform final math, it may run out of gas and revert, even if a valid answer was obtained earlier. Separately, when Pyth is enabled, _normalizePythPrice permits large negative exponents (magnitude up to 255), producing large effective priceDecimals. _scaleCompositePrice then computes 10^(currentDecimals - DECIMALS); for large exponents this overflows uint256 and reverts, despite the data passing earlier checks. Downstream, AusdStrategy relies on OracleHelpers.convertToUsd(), which calls latestRoundData() on SHMON_ORACLE. On revert, OracleHelpers throws OracleCallReverted. allocateFunds() enforces a basis guard (_checkBasisGuard) that depends on these prices; failures block allocation and other valuation-dependent paths. deallocateFunds() bypasses the guard on failure, preserving some liveness, but important operations remain impacted.

Severity

Impact Explanation: [Medium] Important but non-core operations (e.g., allocation and valuation-dependent paths) can be significantly and temporarily DoS'ed. No direct principal loss, no permanent freeze, and deallocation has a bypass for oracle failures.

Likelihood Explanation: [Low] Requires integration misconfiguration, trusted role misuse/compromise, or upstream oracle malfunction/malice (Chainlink-style source or Pyth). These are trusted/integration failures rather than attacker-only actions.

Exploitation

Exploitation Scenarios:

Scenario 1.

Single-leg gas-grief DoS: One SHMON/MON source returns valid data quickly, but the second source gas-bombs (in latestRoundData or decimals). CompositeOracle still calls the second source, which consumes gas and reverts, leaving too little gas to complete the rest of latestRoundData. The call reverts; OracleHelpers rethrows; AusdStrategy allocation guarded by the basis check fails.

Preconditions / Assumptions

- (a). SHMON_ORACLE points to CompositeOracle
- (b). At least one SHMON/MON source is misconfigured or malicious and gas-bombs in latestRoundData() or decimals()
- (c). Caller relies on CompositeOracle.latestRoundData() (e.g., via AusdStrategy basis guard/valuations)

Scenario 2.

Dual-leg compounded gas-grief: Each leg has one honest and one gas-bomb source (or Pyth misbehaves on the MON/USD leg). CompositeOracle queries all sources; two gas-bomb calls in series reduce remaining gas to ~1/4096, making completion infeasible. The composite read reverts, blocking allocation and other valuation-dependent operations.

Preconditions / Assumptions

- (a). Both legs are configured with at least one malicious/misconfigured gas-bomb source (or Pyth misbehaves on the MON/USD leg)
- (b). CompositeOracle continues to query all sources to pick the freshest answer
- (c). Caller relies on CompositeOracle.latestRoundData() through OracleHelpers

Scenario 3.

Pyth negative-exponent overflow: MON/USD leg uses Pyth with a large negative exponent (e.g., -120). _normalizePythPrice returns priceDecimals = 120. If this is the freshest source, _scaleCompositePrice computes 10^(currentDecimals-DECIMALS) ... overflows uint256, and reverts. CompositeOracle fails consistently until the feed is corrected, blocking allocation and valuations.

Preconditions / Assumptions

- (a). MON/USD leg has Pyth enabled and configured
- (b). Pyth returns a large negative exponent (magnitude potentially > 77) with a current publishTime so it is selected as freshest
- (c). Caller relies on CompositeOracle.latestRoundData() through OracleHelpers

Proposed fix

CompositeOracle.sol

File: `src/oracle/CompositeOracle.sol`

Source

```
 ... 215 unchanged lines ...
     /// @return data Latest valid round payload copied into memory
     function _getLatestLegData(PriceLeg storage leg) private view returns (RoundData memory data) {
+        // NOTE: For liveness hardening against gas-grief, consider:
+        // - short-circuiting after the first valid source instead of probing all sources for freshness,
+        // - issuing gas-stipended staticcalls in _getChainlinkSourceData/_getPythSourceData and treating failu
+        // - querying Pyth only if both Chainlink-style sources are invalid (or stale if heartbeat is enforced)
         bool found;

 ... 81 unchanged lines ...
         uint256 magnitude = uint256(uint32(-expo));
         if (magnitude > type(uint8).max) return (0, 0, false);
+        if (magnitude > MAX_PYTH_EXPONENT) return (0, 0, false);

         price = int256(rawPrice);
 ... 28 unchanged lines ...
         if (currentDecimals > DECIMALS) {
             // Scale down when the raw result uses more decimals than the configured output.
-            compositePrice = compositePrice / int256(_pow10(currentDecimals - DECIMALS));
+            uint256 d = currentDecimals - DECIMALS;
+            if (d > MAX_PYTH_EXPONENT) revert InvalidPrice();
+            compositePrice = compositePrice / int256(_pow10(d));
         } else if (currentDecimals < DECIMALS) {
             // Scale up when the configured output needs more decimals than the raw result.
-            compositePrice = compositePrice * int256(_pow10(DECIMALS - currentDecimals));
+            uint256 d = DECIMALS - currentDecimals;
+            if (d > MAX_PYTH_EXPONENT) revert InvalidPrice();
+            compositePrice = compositePrice * int256(_pow10(d));
         }
```

```
    }

    /// @notice Compute `10 ** value` for decimal scaling.
    /// @param value Exponent used for base-10 scaling
    /// @return Ten raised to `value`
    function _pow10(uint256 value) private pure returns (uint256) {
        return DECIMAL_BASE ** value;
    }
}
```

26. [Low] Incorrect zero-target LTV handling in LTVCalculations.calculateSafeWithdrawal causes batch reverts during auto-sized withdrawals with debt

Status

Review status: Unresolved
Remediation status: Unremediated
Remediation note: Created by pipeline analysis

Description

calculateSafeWithdrawal returns the full collateral amount when targetLtv == 0 even if debt > 0. Adapters propagate this value and BaseStrategy uses it when amount == 0 (auto-size), leading to full-collateral withdrawal attempts while debt is open. Underlying venues revert on health checks, stalling deallocation/unwind batches. No funds are lost, but availability/liveness is impacted.

LTVCalculations.calculateSafeWithdrawal reduces the requested target LTV by a buffer and returns the full collateral balance when the resulting conservativeLtv == 0. This includes the case where targetLtv == 0 and debt > 0, which is mathematically incorrect because no collateral should be safely withdrawable until debt is repaid. Both CurvanceCollateralAdapter.getMaxSafeWithdrawal and EulerCollateralAdapter.getMaxSafeWithdrawal forward this library result. BaseStrategy._withdrawCollateral uses amount == 0 to auto-size via getMaxSafeWithdrawal (with safetyBuffer = 0), so targetLtv == 0 drives a full-collateral request despite outstanding debt. For primary-asset deallocation flows, BaseStrategy's reserveAmount clamp can mitigate size, but for other legs it does not apply. Underlying venues (Curvance/Euler) enforce health and will revert, causing batches to fail and delaying deallocation/unwind. This is a correctness bug with operational liveness impact, not a path to principal loss or attacker exploitation.

Severity

Impact Explanation: [Medium] Batches revert and deallocation/unwind can be stalled, temporarily blocking user exits or operational flows; no principal loss occurs.

Likelihood Explanation: [Low] Requires a trusted operator/admin to misconfigure batches (targetLtv == 0 with amount == 0 before repaying debt); not attacker-driven.

Exploitation

Exploitation Scenarios:

Scenario 1.

Secondary-leg (Curvance) deallocation: operator submits withdrawCollateral with position=SECONDARY, token=shMON, targetLtv=0, amount=0 (no reserve). The adapter reports full collateral as safe; strategy attempts to withdraw all while secondary debt > 0; Curvance health checks revert; batch fails and deallocation stalls.

Preconditions / Assumptions

- (a). Secondary collateral is posted and secondary debt > 0
- (b). Curvance cooldown has passed
- (c). Operator constructs a deallocation batch with withdrawCollateral(position=SECONDARY, token=shMON, targetLtv=0, amount=0, reserveAmount=0)
- (d). Venue enforces health checks (reverts on unsafe withdraw)

Scenario 2.

Primary-leg withdraw without reserve clamp: operator submits withdrawCollateral with position=PRIMARY, token=stablecoin (asset), targetLtv=0, amount=0, reserveAmount=0. The adapter reports full collateral as safe; strategy attempts to withdraw all while primary WMON debt > 0; venue reverts; batch fails and maintenance/allocation flow stalls.

Preconditions / Assumptions

- (a). Primary stablecoin collateral is posted and primary WMON debt > 0
- (b). Operator constructs a batch with withdrawCollateral(position=PRIMARY, token=stablecoin, targetLtv=0, amount=0, reserveAmount=0)
- (c). No reserve clamp is used to cap the withdrawal
- (d). Venue enforces health checks (reverts on unsafe withdraw)

Scenario 3.

Euler-leg variant: operator submits withdrawCollateral on an Euler-managed leg with targetLtv=0, amount=0 while paired debt > 0. Adapter reports full collateral; EVault enforces health and reverts; batch fails, delaying execution.

Preconditions / Assumptions

- (a). Euler-managed leg has posted collateral and paired debt > 0
- (b). Operator submits withdrawCollateral with targetLtv=0 and amount=0 for that leg
- (c). Venue enforces health checks (reverts on unsafe withdraw)

Proposed fix

LTVCalculations.sol

File: `src/libraries/LTVCalculations.sol`

Source

```
 ... 106 unchanged lines ...

         // Tighten the requested target with the operator-supplied safety margin.
-        uint256 conservativeLtv = targetLtv > bufferBps ? targetLtv - bufferBps : targetLtv;
+        uint256 conservativeLtv = targetLtv > bufferBps ? targetLtv - bufferBps : 0;

-        // H-04: if conservativeLtv is zero (targetLtv == 0 with debt > 0) the division below
-        // would revert. Return full collateral so the caller can handle an unconstrained position.
-        if (conservativeLtv == 0) return collateral;
+        // If conservativeLtv is zero after buffering and debt > 0, no collateral is safely withdrawable.
+        if (conservativeLtv == 0) return 0;

         // Solve `debtUsd / minCollateralUsd = conservativeLtv / BPS` for minCollateralUsd.
 ... 115 unchanged lines ...
```

Related findings

[Low] Oracle-gated adapter checks in BaseStrategy repay/withdraw under oracle staleness cause operational DoS of partial debt management and deallocation

Description

BaseStrategy’s repay and auto-sized collateral-withdraw flows use adapter view logic via static calls. For Curvance, these paths rely on Chainlink oracles and revert on staleness, preventing partial repay and safe-withdraw sizing. Close-all repay bypasses oracles but may be infeasible without full coverage and, for stablecoin debt, overrides reserveAmount, draining idle reserves.

In BaseStrategy._repay, the strategy calls IDebtAdapter.getDebtBalance and IDebtAdapter.resolveRepayInstruction via staticcall before executing repay. CurvanceDebtAdapter.resolveRepayInstruction uses OracleHelpers (Chainlink) to compute safe partial repayments and reverts on stale oracle data. Similarly, withdrawCollateral with amount==0 triggers ICollateralAdapter.getMaxSafeWithdrawal, which uses LTVCalculations and OracleHelpers and can revert on stale oracles. Under oracle staleness, these adapter checks block partial repayments and auto-sized safe withdrawals. While

closeAllDebt=true bypasses oracles, it requires full coverage and, for stablecoin debt, unconditionally sets amount=availableBalance, overriding reserveAmount, consuming idle assets intended for withdrawals. Operators can mitigate by using explicit withdrawal amounts (avoiding oracle-based sizing), converting assets to increase coverage and then using close-all, or executing the termination routine (which repays via direct adapter calls). Nonetheless, stale oracles create a practical liveness/DoS risk for routine deallocation and debt management.

Severity

Impact Explanation: [Medium] The issue causes temporary DoS of routine deallocation and debt management flows under oracle staleness, impairing withdrawals and risk reduction during stress. It does not directly cause principal loss and has practical workarounds (explicit amounts, conversion plus close-all, or termination).

Likelihood Explanation: [Low] Multiple multiplicative preconditions must align: Curvance debt involvement, sustained oracle staleness during execution windows, and specific operator choices (e.g., partial repay, auto-sized withdraw). Operators have alternative flows that avoid the failing checks, reducing overall occurrence probability.

Exploitation

Exploitation Scenarios:

Scenario 1.

Curvance partial repay DoS: With a Curvance debt leg and a stale debt oracle, BaseStrategy._repay (closeAllDebt=false) calls CurvanceDebtAdapter.resolveRepayInstruction, which reverts on staleness, preventing partial debt reduction and increasing liquidation risk during stress.

Preconditions / Assumptions

- (a). Strategy is configured with CurvanceDebtAdapter on the affected debt leg
- (b). The relevant Chainlink debt oracle is stale beyond its heartbeat
- (c). Operator submits a partial repay (closeAllDebt=false) batch
- (d). Oracle staleness persists during the transaction window
- (e). Operator does not or cannot first increase coverage to enable a close-all repay

Scenario 2.

Close-all drains reserves: Oracle staleness blocks partial repay on a stablecoin debt leg, leading the operator to use closeAllDebt=true. BaseStrategy then sets amount=availableBalance, overriding reserveAmount and consuming idle stablecoin, leaving little or no liquidity to return to the vault in that epoch.

Preconditions / Assumptions

- (a). A stablecoin debt leg is present (token equals the vault asset)
- (b). Partial repay is blocked by oracle staleness, leading the operator to use closeAllDebt=true
- (c). Batch is designed to preserve idle reserves via reserveAmount for user withdrawals
- (d). Available balance is sufficient to perform a close-all repay
- (e). Deallocation includes returning idle assets to the vault in the same epoch

Scenario 3.

Combined deallocation DoS: During deallocation, withdrawCollateral(amount==0) uses adapter getMaxSafeWithdrawal (oracle-based) and partial repay on Curvance uses resolveRepayInstruction (oracle-based). With stale oracles, both paths revert, blocking routine liquidity freeing and de-risking until operators switch to explicit amounts or alternative flows.

Preconditions / Assumptions

- (a). Operator uses withdrawCollateral with amount==0 to auto-size a safe withdrawal via the adapter
- (b). Relevant oracles are stale beyond heartbeat
- (c). Operator also attempts a Curvance partial repay in the same or proximate flow
- (d). Oracle staleness persists through these operations
- (e). Operators have not switched to explicit fixed withdrawal amounts or termination yet

Proposed fix

BaseStrategy.sol

File: `src/strategies/BaseStrategy.sol`

Source

```
 ... 798 unchanged lines ...
             // Get current debt using staticcall
             bytes memory data = abi.encodeWithSelector(IDebtAdapter.getDebtBalance.selector, token, address(thi
-            bytes memory result = address(adapter).functionStaticCall(data);
+            (bool okDebt, bytes memory result) = address(adapter).staticcall(data);
+            if (!okDebt) { return; }
             uint256 currentDebt = abi.decode(result, (uint256));

             bytes memory instructionData = abi.encodeWithSelector(
                 IDebtAdapter.resolveRepayInstruction.selector, amount, currentDebt, closeAllDebt
             );
-            bytes memory instructionResult = address(adapter).functionStaticCall(instructionData);
+            (bool okInstr, bytes memory instructionResult) = address(adapter).staticcall(instructionData);
+            if (!okInstr) { return; }
             (uint256 toRepay, bool useFullRepaySentinel) = abi.decode(instructionResult, (uint256, bool));
             if (toRepay > 0 || useFullRepaySentinel) {
 ... 584 unchanged lines ...
```

[Low] Pair-local auto-size quotes ignoring cross-market health in BaseStrategy amount==0 borrow/withdraw (Curvance context) cause allocation/deallocation DoS

Description

When amount==0, BaseStrategy auto-sizes borrow/withdraw via Curvance adapters using only a single market pair, ignoring other live Curvance positions held by the same strategy address. The protocol enforces portfolio-level health at execution, so the pair-local quote can be too high and the actual call reverts, causing batch DoS. Euler is unlikely affected due to its per-vault controller model.

BaseStrategy._withdrawCollateral and _borrow interpret amount==0 as a request to auto-size using adapter quote functions. CurvanceCollateralAdapter.getMaxSafeWithdrawal and CurvanceDebtAdapter.getMaxBorrowAmount compute headroom from only their paired collateral/debt markets. If the strategy address also has other Curvance markets open (e.g., both legs on Curvance or leftover markets after an adapter migration), the true portfolio-level health can be stricter than the pair-local quote. Curvance then reverts the borrow/withdraw at execution, causing allocation or deallocation batches to fail. This is a liveness/operational risk (batches revert and must be retried with explicit sizing or after cleaning up positions), not a direct loss-of-funds bug. For Euler, the controller/vault-pair model aligns quotes with execution checks, so this cross-market mismatch is unlikely.

Severity

Impact Explanation: [Medium] Causes significant but temporary availability loss of core strategy operations (allocation/deallocation) by reverting batches and delaying redemptions or deployment until operators adjust.

Likelihood Explanation: [Low] Relies on trusted operator/admin actions or configuration (e.g., migrating without full unwind, configuring both legs on Curvance, or persisting residual positions while using amount==0).

Exploitation

Exploitation Scenarios:

Scenario 1.

Post-migration leftover Curvance positions: After switching adapters to a new Curvance market without fully unwinding the old one, an operator runs a deallocation with withdrawCollateral(amount=0). The adapter quotes a pair-local max, but Curvance enforces portfolio-level health including the leftover market, so the withdraw reverts and the batch fails.

Preconditions / Assumptions

- (a). Strategy previously used Curvance Market A and migrated adapters to Curvance Market B without a full unwind
- (b). The strategy address still holds live position(s) in Market A
- (c). Operator uses zero-amount withdrawCollateral in a deallocation batch

Scenario 2.

Both legs configured on Curvance: The strategy uses two Curvance markets for primary and secondary legs. An allocation uses borrow(amount=0) on the primary leg. The adapter's pair-local quote ignores secondary exposure; Curvance rejects the borrow due to portfolio-level health, reverting the batch.

Preconditions / Assumptions

- (a). Both primary and secondary legs are configured on Curvance (two markets on the same address)
- (b). Operator uses zero-amount borrow on the primary Curvance leg
- (c). Curvance enforces portfolio-level health across both markets for the address

Scenario 3.

Residual Curvance dust blocks auto-size: A small residual Curvance position remains from earlier operations. Routine rebalance/deallocation uses amount==0 auto-sizing on the active Curvance market. The adapter continues to overshoot due to the residual exposure; Curvance reverts each time, repeatedly DoSing batches until cleanup or explicit amounts are used.

Preconditions / Assumptions

- (a). A residual Curvance position remains in a legacy market (collateral and/or debt)
- (b). Operator continues using zero-amount auto-size for borrow/withdraw on the active Curvance market
- (c). Curvance portfolio-level health includes the residual position, tightening real headroom

Proposed fix

BaseStrategy.sol

File: `src/strategies/BaseStrategy.sol`

Source

```
 ... 681 unchanged lines ...
         // withdraw flows.

+        // Disable zero-amount auto-size to avoid pair-local misquotes causing protocol-level reverts.
+        if (amount == 0) revert InvalidParams();
+
     {
         if (amount == 0) {
 ... 43 unchanged lines ...
         internal
     {
+        // Disable zero-amount auto-size to avoid pair-local misquotes causing protocol-level reverts.
+        if (amount == 0) revert InvalidParams();
+
         IDebtAdapter adapter = _getDebtAdapter(position);
         if (amount == 0) {
 ... 661 unchanged lines ...
```

27. [Informational] Missing post-borrow receipt validation in CurvanceDebtAdapter.borrow causes silent principal loss with weird ERC20 debt tokens

Status

Review status: Unresolved
Remediation status: Unremediated
Remediation note: Created by pipeline analysis

Description

CurvanceDebtAdapter.borrow does not verify how many tokens were actually received after a borrow. If the Curvance debt asset is a non-standard ERC20 (fee-on-transfer/short-delivery), the strategy's debt increases by the full amount while receiving fewer tokens, silently realizing a loss.

The Curvance debt adapter calls DEBT_TOKEN.borrow(amount, address(this)) and emits success without checking the strategy's actual token balance delta. Unlike the Euler adapter, which validates the returned amount, the Curvance adapter assumes exact-transfer semantics. If a Curvance-listed debt token is fee-on-transfer/deflationary or otherwise short-delivery, the adapter records full debt creation while transferring fewer tokens to the strategy. Because subsequent strategy steps typically clamp to actual balances, execution may proceed without reverting while the loss is silently realized. This is a valid integration hardening gap; however, it primarily impacts scenarios involving atypical ERC20 tokens, which are uncommon in lending markets and not the intended assets here.

Severity

Impact Explanation: [High] If a non-standard debt token is used, the borrow can create full debt while delivering fewer (or no) tokens to the strategy, causing direct and material principal loss.

Likelihood Explanation: [Low] All scenarios require rare/exceptional preconditions: a Curvance-listed debt token with non-standard transfer behavior and/or trusted-role misuse (malicious token admin or configuration). Such listings are uncommon in lending markets.

Exploitation

Exploitation Scenarios:

Scenario 1.

Fee-on-transfer debt token: The strategy borrows amount A of a Curvance-listed debt token T that charges a 2% transfer tax. Curvance records debt A and transfers T, but only 0.98·A arrives due to the tax. The adapter does not detect the shortfall, and subsequent steps proceed on the reduced balance, realizing a 2% loss per borrow.

Preconditions / Assumptions

- (a). The Curvance market used by the strategy lists a fee-on-transfer/deflationary debt token T with a 2% transfer tax.
- (b). The strategy borrows from that market via CurvanceDebtAdapter.borrow without receipt validation.
- (c). Subsequent strategy steps clamp to actual balances, allowing the batch to proceed without reverting.

Scenario 2.

Malicious token admin toggles tax: A Curvance-listed debt token T has admin-togglable transfer taxes. An attacker front-runs the borrow to raise tax to 50%, Curvance records debt A but only 0.5·A reaches the strategy, then the attacker resets the tax. The adapter emits success without detecting the shortfall, causing a large, immediate loss.

Preconditions / Assumptions

- (a). The Curvance market lists a debt token T with admin-togglable transfer taxes.
- (b). A malicious or compromised token admin can raise the tax for the borrow window and reset it afterward.
- (c). The strategy borrows via CurvanceDebtAdapter.borrow, which lacks a receipt check.

Scenario 3.

Short-delivery to zero: A Curvance-listed debt token T exhibits nonstandard behavior delivering near-zero tokens on transfer while returning success. Curvance records debt A; the strategy receives ~0. The adapter emits success and the batch often continues (clamping to balance), leaving the strategy with full debt and no corresponding assets.

Preconditions / Assumptions

- (a). The Curvance market lists a debt token T with nonstandard short-delivery behavior (e.g., transfers succeed but deliver near-zero tokens to the strategy).
- (b). The strategy borrows via CurvanceDebtAdapter.borrow, which does not validate pre/post balance deltas.
- (c). Subsequent steps often clamp to actual balances, so the batch may continue without reverting.

Proposed fix

CurvanceDebtAdapter.sol

File: `src/adapters/curvance/CurvanceDebtAdapter.sol`

Source

```
 ... 95 unchanged lines ...
     /// @inheritdoc IDebtAdapter
     function borrow(address asset, uint256 amount) external {
-        // Curvance reverts internally on invalid borrow attempts, so reaching this point means
-        // the protocol accepted the requested amount for the adapter-owned position.
-        DEBT_TOKEN.borrow(amount, (address(this)));
-        emit Borrowed(asset, amount, (address(this)));
+        // Measure actual receipt and revert on short-delivery to prevent silent losses.
+        uint256 preBalance = IERC20(asset).balanceOf(address(this));
+        DEBT_TOKEN.borrow(amount, address(this));
+        uint256 received = IERC20(asset).balanceOf(address(this)) - preBalance;
+        if (received < amount) {
+            revert DebtBorrowFailed(amount, received);
+        }
+
+        emit Borrowed(asset, amount, address(this));
     }

 ... 206 unchanged lines ...
```

28. [Informational] Trusting cWMON redeem() return value in Swapper.processRewards causes reserve undershoot or reward-processing revert

Status

Review status: Unresolved
Remediation status: Unremediated
Remediation note: Created by pipeline analysis

Description

Swapper.processRewards trusts the cWMON redeem() return value to size post-redeem WMON instead of measuring the actual balance delta. If redeem() ever returns a value that differs from the WMON actually transferred, the function can mis-size the unwrap amount, leading to reserve undershoot or a revert; WMON-output mode can overreport the returned amount.

In Swapper.processRewards, the function computes wmonAfterRedeem = wmonBefore + redeemedWmon, where redeemedWmon is the raw return from ICurvanceWrappedMON.redeem. Unlike other legs that use balance-delta accounting, this path trusts the external return value. If Curvance's redeem() ever returns an amount that does not equal the WMON actually transferred to the strategy, then: (1) In VAULT_ASSET mode, monAmount is computed from an inflated WMON balance and can unwrap more than the true excess, consuming some of the pre-existing reserve below targetRemainingWmon, or revert if attempting to transfer more WMON than held; (2) In WMON-output mode, the function overreports outputAmount, though on-chain balances remain correct. Later legs (MON unwrap delta and stable-asset swap delta) are already measured by actual deltas, limiting impact to reserve undershoot or an operational revert.

Severity

Impact Explanation: [Low] No principal loss or core invariant break; effects are reserve undershoot (soft policy) or a transient revert, and a view-only overreport in WMON-output mode.

Likelihood Explanation: [Low] Requires an integration mismatch (redeem() return not matching actual transferred WMON), which is unlikely under standard ERC-4626-like semantics and WMON behavior.

Exploitation

Exploitation Scenarios:

Scenario 1.

VAULT_ASSET: cWMON.redeem returns R but actually transfers D < R; monAmount is sized from B + R (B=pre-existing WMON). Transfer to unwrapper succeeds using some of B, leaving post-processing WMON below targetRemainingWmon.

Preconditions / Assumptions

- (a). Operator calls processRewards with outputAsset=VAULT_ASSET and a nonzero targetRemainingWmon
- (b). Strategy has pre-existing WMON reserve B
- (c). cWMON.redeem returns R while actually transferring D < R to the strategy
- (d). B + D >= monAmount computed from (B + R - targetRemainingWmon)

Scenario 2.

VAULT_ASSET: cWMON.redeem returns R but actually transfers D < R; the computed monAmount exceeds the actual WMON balance (B + D), causing the transfer to the unwrapper to revert and the reward-processing call to fail.

Preconditions / Assumptions

- (a). Operator calls processRewards with outputAsset=VAULT_ASSET
- (b). cWMON.redeem returns R while actually transferring D < R to the strategy
- (c). Computed monAmount = (B + R - targetRemainingWmon) exceeds actual WMON balance B + D

Scenario 3.

WMON-output: cWMON.redeem returns R but actually transfers D < R; the function returns R as outputAmount while only D was received on-chain, causing a view-level overreporting without affecting balances.

Preconditions / Assumptions

- (a). Operator calls processRewards with outputAsset=WMON
- (b). cWMON.redeem returns R while actually transferring D < R to the strategy

Proposed fix

Swapper.sol

File: `src/adapters/rewards/Swapper.sol`

Source

```
 ... 178 unchanged lines ...
         // harvested reward inventory from any operator-reserved WMON already held by the strategy.
         uint256 wmonBefore = IWrappedToken(wmon).balanceOf(address(this));
-        uint256 redeemedWmon = _redeemCwMon(params.cwMon);
-        uint256 wmonAfterRedeem = wmonBefore + redeemedWmon;
+        _redeemCwMon(params.cwMon);
+        uint256 wmonAfterRedeem = IWrappedToken(wmon).balanceOf(address(this));

         // Keep everything as WMON when the caller wants a wrapped balance for later use.
 ... 136 unchanged lines ...
```