



Security Review For Delpho



Collaborative Audit Prepared For:
Lead Security Expert(s):

Delpho
eeyore
TessKimy

Date Audited:

April 8 - April 22, 2026

Introduction

Delpho is a yield-bearing stablecoin protocol built on Hyperliquid. It enables users to hold USDV – a stablecoin that accrues yield natively – by deploying capital into a diversified set of on-chain strategies. Delpho combines the stability of a dollar-pegged asset with the return profile of active DeFi participation, targeting sustainable, risk-managed yield for both retail and institutional stablecoin holders.

Scope

Repository: `Delpho-labs/delpho_contracts`

Audited Commit: `e3c0c8633cb4254ca070244a9a8d1b1fb3f8ee82`

Final Commit: `a53984283a02592694518c68ae343a02b23e9a46`

Files:

- `src/base/DelphoBase.sol`
- `src/base/DelphoConfigurator.sol`
- `src/base/interfaces/IDelphoBase.sol`
- `src/base/interfaces/IDelphoConfigurator.sol`
- `src/configurators/base/BaseConfig.sol`
- `src/configurators/base/DelphoExecutorConfig.sol`
- `src/configurators/base/DelphoHyperCoreConfig.sol`
- `src/configurators/base/DelphoMorphoConfig.sol`
- `src/configurators/base/DelphoOracleConfig.sol`
- `src/configurators/base/DelphoRedemptionConfig.sol`
- `src/configurators/base/DelphoRewardDistributorConfig.sol`
- `src/configurators/base/DelphoTokenConfig.sol`
- `src/configurators/base/DelphoVaultConfig.sol`
- `src/configurators/base/interfaces/IBaseConfig.sol`
- `src/configurators/base/interfaces/IDelphoExecutorConfig.sol`
- `src/configurators/base/interfaces/IDelphoHyperCoreConfig.sol`
- `src/configurators/base/interfaces/IDelphoMorphoConfig.sol`
- `src/configurators/base/interfaces/IDelphoOracleConfig.sol`
- `src/configurators/base/interfaces/IDelphoRedemptionConfig.sol`
- `src/configurators/base/interfaces/IDelphoRewardDistributorConfig.sol`

- src/configurators/base/interfaces/IDelphoTokenConfig.sol
- src/configurators/base/interfaces/IDelphoVaultConfig.sol
- src/configurators/DelphoConfigProvider.sol
- src/configurators/interfaces/IDelphoConfigProvider.sol
- src/core/DelphoOracle.sol
- src/core/DelphoSortedDebts.sol
- src/core/DelphoStabilityPool.sol
- src/core/DelphoStableManager.sol
- src/core/DelphoStable.sol
- src/core/DelphoVault.sol
- src/core/interfaces/IDelphoOracle.sol
- src/core/interfaces/IDelphoVault.sol
- src/core/interfaces/IStableManager.sol
- src/core/types/DelphoCoreTypes.sol
- src/DelphoRewardDistributor.sol
- src/DelphoStaker.sol
- src/executor/base/DelphoBaseExecutor.sol
- src/executor/base/DelphoExecutorCoreExtension.sol
- src/executor/base/DelphoExecutorFeeExtension.sol
- src/executor/base/DelphoExecutorRecoveryExtension.sol
- src/executor/base/DelphoExecutorSwapExtension.sol
- src/executor/base/interfaces/IDelphoExecutorBase.sol
- src/executor/DelphoHyperLendExecutor.sol
- src/executor/DelphoMorphoExecutor.sol
- src/executor/DelphoStableExecutor.sol
- src/executor/libraries/DelphoExecutorLib.sol
- src/executor/libraries/DelphoSwapLib.sol
- src/executor/types/DelphoExecutorTypes.sol
- src/interfaces/external/ICoreDepositWallet.sol
- src/interfaces/external/ICoreWriter.sol
- src/interfaces/external/IDebtVariableToken.sol

- src/interfaces/external/IGlueAdapter.sol
- src/interfaces/external/IHyperLendPool.sol
- src/interfaces/external/IHyperLoop.sol
- src/interfaces/external/ILiquidSwapAdapter.sol
- src/interfaces/external/IMorphoFlashLoanCallback.sol
- src/interfaces/external/IMorpho.sol
- src/interfaces/external/IMorphoTypes.sol
- src/interfaces/external/ISentimentPositionManager.sol
- src/interfaces/external/ISentimentProtocolLens.sol
- src/interfaces/external/ISwapper.sol
- src/interfaces/external/IWHYPE.sol
- src/interfaces/IDelphoRewardDistributor.sol
- src/interfaces/IDelphoRouter.sol
- src/interfaces/IDelphoSortedDebts.sol
- src/interfaces/IDelphoStabilityPool.sol
- src/interfaces/IDelphoStable.sol
- src/interfaces/IDelphoStaker.sol
- src/interfaces/IDelphoVaultLens.sol
- src/libs/StableManagerLib.sol
- src/libs/VaultMath.sol
- src/periphery/DelphoRouter.sol
- src/types/DelphoTypes.sol

Final Commit Hash

a53984283a02592694518c68ae343a02b23e9a46

Findings

Each issue has an assigned severity:

- High issues are directly exploitable security vulnerabilities that need to be fixed.
- Medium issues are security vulnerabilities that may not be directly exploitable or may require certain conditions in order to be exploited. All major issues should be addressed.

- Low/Info issues are non-exploitable, informational findings that do not pose a security risk or impact the system's integrity. These issues are typically cosmetic or related to compliance requirements, and are not considered a priority for remediation.

Issues Found

High	Medium	Low/Info
3	13	28

Issues Not Fixed and Not Acknowledged

High	Medium	Low/Info
0	0	0

Issue H-1: Accrued interest is silently destroyed during fallback redemption instead of being paid to the fee receiver [RESOLVED]

Source:

<https://github.com/sherlock-audit/2026-04-delpho-update-apr-8th-2026/issues/74>

Summary

When the Stability Pool cannot fully absorb a redemption and the remainder falls through to `_processOneFallbackBorrower`, each processed borrower's accrued `pendingInterest` is zeroed or reduced from the debt ledger without any corresponding payment to `feeReceiver`. The interest was earned protocol revenue and is permanently lost. `repayStable` handles this correctly, proving the omission is unintentional.

Vulnerability Detail

The pool redemption path (`_redeemFromPool`) has no such issue since the SP is interest-exempt. The problem only manifests in the fallback path against individual borrowers, who do accrue interest.

In the full absorption branch, `pendingInterest` is set to zero. In the partial reduction branch, an `interestReduction` is subtracted from `pendingInterest`. In both cases no USDV is minted or transferred to `feeReceiver`.

The redeemer's USDV is burned wholesale in `_redeemFromBorrowers` covering both principal and interest reduction across all processed borrowers. The interest portion is burned along with the principal, then silently wiped from the borrower's ledger, with no revenue collection.

Compare with `repayStable` which correctly separates interest via `StableManagerLib.processRepayment` and transfers the interest portion to `feeReceiver` before burning the principal.

Impact

The protocol permanently loses accrued interest revenue from every borrower touched by a fallback redemption.

Code Snippet

https://github.com/sherlock-audit/2026-04-delpho-update-apr-8th-2026/blob/main/delpho_contracts/src/core/DelphoStableManager.sol#L666-L677

https://github.com/sherlock-audit/2026-04-delpho-update-apr-8th-2026/blob/main/delpho_contracts/src/core/DelphoStableManager.sol#L425-L435

Tool Used

Manual Review

Recommendation

Before zeroing or reducing `pendingInterest` in `_processOneFallbackBorrower`, mint the interest portion to `feeReceiver`.

Discussion

Oxklapouchy

1. The merge of PR 199 looks wrong.

The code on `fix-review` hash diverges from the PR diff in two places inside the `else` branch of `_processOneFallbackBorrower`:

- `interestPaid = interestReduction`; is missing. `interestPaid` defaults to 0, so total `InterestPaid` in `_redeemFromBorrowers` never accumulates the partial-branch contribution and `feeReceiver` is under-minted on every partial fallback iteration.
The issue is not actually closed for the dominant code path.
- `mulDiv` was swapped to `mulDivUp`, which contradicts the adjacent comment ("rounds down, guaranteed $\leq$ `pendingInterest`"). Strong indication of a bad merge rather than an intentional override.

Please re-apply the PR verbatim, and add a regression test that forces `fallbackRedeemPortionBps < MAX_BPS` across multiple borrowers and asserts `feeReceiver` balance change equals $\sum \text{interestReduction} + \text{actualFee}$. That test would have caught both regressions at merge time.

2. Even with the PR re-applied, the `else` branch is the wrong fix.

Proportional split between principal and interest is a new payment ordering invented for this branch only. The protocol's documented ordering is **interest-first** (`repayStable` calls `StableManagerLib.processRepayment`), and the anti-dust projection a few lines above (L711 to L716) already predicts via `processRepayment`, i.e. it predicts interest-first while the actual execution does proportional. The fix ships a prediction/execution mismatch.

Interest-first is also strictly better on revenue, not a tradeoff:

- Today's realized interest: full vs. proportional fraction.
- Future accrual base: with $P = \text{usdvMinted}$, $I = \text{pendingInterest}$, $U = \text{usdApplied}$, interest-first leaves $\text{usdvMinted} = \min(P, P+I-U)$, proportional leaves $P \cdot (P+I-U) / (P+I)$.

I). Interest-first is greater than or equal for all valid inputs, strictly greater whenever $I > 0$.

- Borrower's total debt and LCR are split-invariant, so no fairness regression.
- Removes the up-to-1-wei-per-iteration `feeReceiver` under-credit from rounding.
- Shrinks the residual `pendingInterest` that feeds the unrelated double-mint at L258 to L260 in liquidation.

Suggested patch for the `else` branch:

```
} else {
    (uint256 newBorrowed, uint256 newInterest, , uint256 interestReduced) =
        StableManagerLib.processRepayment(
            borrowerPos.usdvMinted,
            borrowerPos.pendingInterest,
            usdApplied
        );
    borrowerPos.usdvMinted = newBorrowed;
    borrowerPos.pendingInterest = newInterest;
    interestPaid = interestReduced;
}
```

Oxklapouchy

Fix confirmed in [PR#252](#)

Issue H-2: Accrued interest is silently destroyed during liquidation instead of being paid to the fee receiver [RESOLVED]

Source:

<https://github.com/sherlock-audit/2026-04-delpho-update-apr-8th-2026/issues/75>

Summary

When a borrower is liquidated, their accrued `pendingInterest` is zeroed from the debt ledger without any payment to `feeReceiver`. Every liquidated position carries accrued interest, so every liquidation permanently destroys earned protocol revenue.

Vulnerability Detail

`_liquidateUser` first materializes accrued interest via `_updateInterest`, then absorbs debt via the Stability Pool path or the fallback path, and finally wipes the position by setting both `usdvMinted` and `pendingInterest` to zero. The interest is simply discarded with no USDV minted or transferred to `feeReceiver`.

The normal liquidation path treats the full `totalDebt` (including interest) as a monolithic amount when computing the burn/mint split. The interest component is folded into `spUsdvToMint` and `burnAmount` calculations without being separated. Whether the SP absorbs the debt directly or part of it rolls to the fallback path, the interest never reaches the fee receiver.

Compare with `repayStable` which correctly separates interest via `StableManagerLib.processRepayment` and transfers the interest portion to `feeReceiver` before burning principal.

Unlike the fallback redemption case, this flow triggers on EVERY liquidation. The protocol never collects interest from liquidated positions, only from voluntary repayments.

Impact

The protocol permanently loses accrued interest revenue from every liquidated position. Since a significant portion of positions in any CDP protocol close via liquidation (especially during volatile market periods), the cumulative revenue impact is substantial.

Code Snippet

https://github.com/sherlock-audit/2026-04-delpho-update-apr-8th-2026/blob/main/delpho_contracts/src/core/DelphoStableManager.sol#L239-L242

https://github.com/sherlock-audit/2026-04-delpho-update-apr-8th-2026/blob/main/delpho_contracts/src/core/DelphoStableManager.sol#L745-L757

Tool Used

Manual Review

Recommendation

Before zeroing the position in `_liquidateUser`, transfer the accrued `pendingInterest` to `feeReceiver`.

Discussion

Oxklapouchy

The fix for this issue is correct in the healthy-liquidation case but breaks in shortfall scenarios.

Healthy case (collateral covers `totalDebt + penalty`):

The mint at [L258-260](#) is a valid "delayed direct mint" of interest revenue. Supply rebalances cleanly:

- SP burn (`burnAmount`) destroys part of the absorbed debt from circulation immediately.
- SP debt rollover (`spUsdvToMint`) becomes a future supply-reducing obligation against new SP collateral.
- Fallback distribution rolls the residual into absorbers' `usdvMinted` as new principal, and those absorbers will burn it when they repay.
- The [L259](#) mint is funded by the same collateral flow: `feeReceiver` is just receiving USDV now that would otherwise have been minted gradually via voluntary repayments. SP/absorbers carry the corresponding obligation forward.

Tracing `Principal=100`, `Interest=10`, `totalDebt=110`, `burnAmount=55`, `spUsdvToMint=55`:
☒ Supply at liquidation = -45, `feeReceiver` +10. When SP later repays its 55, ☒ Supply = -55.
Cumulative: -100 supply, +10 `feeReceiver`, identical to what `repayStable` would produce.
Fully backed.

Shortfall case:

Collateral doesn't reach `totalDebt + penalty`. The protocol already classifies revenue as junior to principal via `_recordLiquidationShortfall(badDebt, revenueShortfall)` at [L815](#) and [L910](#), where the interest fraction of any uncovered debt is recorded as forgone revenue. The intent is "feeReceiver should NOT count this much interest as collected."

But [L259](#) then mints the full original `pendingInterest` regardless. The same dollars get two contradictory treatments, recorded as forgone in `totalForgoneRevenue` and simultaneously delivered to `feeReceiver`. No SP/absorber took on the corresponding obligation in the shortfall portion, so there is no future supply-reducing leg to balance

this mint. The supply-minus-claims gap that opens here is permanent, exactly equal to `revenueShortfallInterestPortion`.

The proper forfeit hierarchy under stress should be:

1. Penalty forfeited first (protocol upside).
2. Interest revenue forfeited next, where both `feeReceiver` (mint side) and SP (burn side) walk away with nothing for the uncollected interest fraction.
3. Principal forfeited last, recorded as `badDebt`.

(1) and (3) are implemented. (2) is implemented on the SP-burn side (less absorbed means less burned) but broken on the `feeReceiver` side.

Suggested fix:

Track interest actually realized through the absorption paths and mint only that amount at L258-260. Cleanest is to add `interestRealized` to `LiquidationData`, increment it inside `_processNormalLiquidation` for the SP-absorbed portion and inside `_processFallbackLiquidation` for the portion actually rolled onto absorbers, and mint `interestRealized` instead of `userPosition.pendingInterest`. Equivalent simpler patch: subtract the interest portion of the current liquidation's `revenueShortfall` from the mint amount.

After the fix:

- Healthy liquidation: `feeReceiver` gets full `pendingInterest`, supply reconciles via SP/absorbers' eventual repayments.
- Shortfall liquidation: `feeReceiver` and SP both forfeit revenue proportional to the uncovered debt, and `totalForgoneRevenue` matches reality.

Test that would have caught this: liquidate a position whose collateral value falls below `totalDebt`, then `assert ΔfeeReceiver.balanceOf(USDV) + ΔtotalForgoneRevenue.interestPortion == pendingInterestAtLiquidation`. Current code over-shoots by `revenueShortfall.interestPortion`.

Oxklapouchy

Fix confirmed in [PR#255](#)

Issue H-3: HyperLend Rewards Controller Rewards Are Never Claimed or Forwarded [RESOLVED]

Source:

<https://github.com/sherlock-audit/2026-04-delpho-update-apr-8th-2026/issues/111>

Summary

During `initialize`, the executor pre-approves HyperLoop for debt delegation and `hYield` token spend, but never sets up any mechanism to claim rewards from HyperLend's rewards controller. Because the executor holds leveraged supply positions in HyperLend, it continuously accumulates yield incentive rewards. These rewards are never claimed, tracked, or forwarded to the vault or users, resulting in a permanent loss of yield for the protocol.

Vulnerability Details

The executor supplies collateral to the HyperLend pool and opens leveraged long positions via HyperLoop. Lending protocols with a rewards controller (similar to Aave's incentives controller) distribute reward tokens to suppliers and borrowers based on their position size over time. The executor, as the on-chain holder of the `aToken` (`hYieldAsset`) and the variable debt position, is the direct recipient address for these accruing rewards.

No function in the contract calls the rewards controller to claim, sweep, or otherwise handle accumulated reward tokens. There is no `claimRewards`, `harvestRewards`, or equivalent entry point. The reward tokens accumulate at the executor's address indefinitely and cannot be recovered through any existing code path.

Attack Path

1. Executor opens a leveraged supply position in HyperLend via `executeFullEvmFlow`.
2. HyperLend's rewards controller begins accruing reward tokens to the executor's address proportional to the supplied collateral and borrowed debt balance.
3. Over time, as the position remains open, substantial reward tokens accumulate at the executor contract.
4. No function exists to claim these rewards. They sit locked at the executor address permanently.
5. Even after `closeLeveragePosition` is called, the accrued-but-unclaimed rewards remain stranded.

Impact

The protocol permanently loses all HyperLend incentive rewards earned on leveraged supply positions. Depending on the reward token value and the duration and size of open positions, this can represent a significant portion of the expected yield for vault depositors. The loss is irreversible unless a contract upgrade adds a claim path, and any already-accrued rewards before such an upgrade may still be unrecoverable if the rewards controller has a claim window.

Recommendation

Add a function restricted to the executor role that calls the HyperLend rewards controller's claim function, targeting the reward tokens accrued by the executor's supply and borrow positions. The claimed rewards should then be either converted to the collateral asset and returned to the vault as yield, or forwarded to a designated fee or yield distribution address within the protocol.

Discussion

Oxklapouchy

Fix confirmed in [PR#223](#)

Issue M-1: Sorted debt list not updated after collateral deposit allowing unfair targeting in fallback redemptions and liquidations [RESOLVED]

Source:

<https://github.com/sherlock-audit/2026-04-delpho-update-apr-8th-2026/issues/76>

Summary

`Router.depositCollateral()` does not trigger a sorted debt list update after increasing a user's collateral. The user's position remains stale in `DelphoSortedDebts`, causing them to be unfairly targeted first in fallback redemptions and fallback liquidation debt distribution despite having improved their collateral-to-debt ratio.

Vulnerability Detail

Every other Router flow that modifies a user's LCR triggers `updateSortedDebt`: `depositAndMint` updates via `mintStable`, `repayStable` updates inline, and `requestWithdrawal` updates via `Vault.requestWithdrawal`. Only `depositCollateral` (deposit-only path) leaves the sorted position stale.

The fix belongs in the Router, not the Vault. Placing it in `Vault.collateralDeposit` would cause a redundant double sort for `depositAndMint`, since `mintStable` already calls `_updateSortedDebt` internally after the deposit phase completes.

Both `_redeemFromBorrowers` and `_distributeFallbackDebt` walk `DelphoSortedDebts` from the head (riskiest first). A user whose LCR improved via a deposit-only call still appears at their old, riskier position.

The user cannot refresh their position themselves because `updateSortedDebt` is gated by `onlyVaultOrRouter` and the Router has no standalone refresh function. They must perform some unrelated operation to trigger an update.

Impact

A borrower who deposits collateral to improve their position is still treated as the riskiest borrower.

Code Snippet

https://github.com/sherlock-audit/2026-04-delpho-update-apr-8th-2026/blob/main/delpho_contracts/src/periphery/DelphoRouter.sol#L73-L80

Tool Used

Manual Review

Recommendation

Call `updateInterest` and `updateSortedDebt` at the end of `Router.depositCollateral()` for the user, matching the pattern already used by `repayStable` and `requestWithdrawal`. The interest must be up to date before the sorted position is recomputed, since the LCR numerator depends on `pendingInterest`.

Discussion

0xklapouchy

Fix confirmed in [PR#224](#)

Issue M-2: DelphoStaker reward distribution can be frontrun to steal yield from existing stakers [RESOLVED]

Source:

<https://github.com/sherlock-audit/2026-04-delpho-update-apr-8th-2026/issues/77>

Summary

`DelphoStaker.notifyRewardAmount()` instantly increases `_totalAssets`, causing an immediate share price jump with no deposit cooldown, minimum lock period, or reward streaming. An attacker can sandwich the reward notification transaction by depositing a large amount of USDV before and withdrawing immediately after, capturing a disproportionate share of the reward at the expense of existing stakers.

Vulnerability Detail

The `notifyRewardAmount` function applies the full reward to `_totalAssets` in a single atomic update:

```
function notifyRewardAmount(uint256 amount) external override nonReentrant
↪ whenNotPaused {
    if (!IDelphoConfigProvider(delphoConfigProvider).isRewardDistributor(msg.sender)
    ↪ ) revert DS_UNAUTHORIZED();
    IERC20(asset()).safeTransferFrom(msg.sender, address(this), amount);
    _totalAssets += amount; // <- instant share price jump
}
```

The DelphoStaker has no protective mechanisms against sandwich attacks. The reward notification is triggered by `DelphoRewardDistributor.swapAndDistribute()`, which calls `staker.notifyRewardAmount(toStaker)`. This transaction can be sandwiched.

Attack scenario:

Assume the Staker holds 1_000_000 USDV with 1_000_000 sUSDV total supply (1:1 ratio), and a 10_000 USDV reward is about to be distributed.

1. Attacker predicts the `swapAndDistribute` call
2. Attacker front-runs: calls `Router.stakeUSDV(1_000_000)`
 - `_totalAssets = 2_000_000`, `totalSupply = 2_000_000`
 - Attacker receives 1_000_000 sUSDV shares (50% of pool)
3. `notifyRewardAmount(10_000)` executes:
 - `_totalAssets = 2_010_000`, `totalSupply = 2_000_000`
4. Attacker back-runs: calls `Router.unstakeUSDV(1_005_000)`

- Attacker redeems shares worth 1_005_000 USDV
- Profit: 5_000 USDV (half of the 10_000 reward)

5. Existing stakers received only 5_000 instead of 10_000

The attacker's profit scales with the ratio of their deposit to the existing pool. A larger deposit relative to the pool captures a larger fraction of the reward.

Impact

Existing stakers systematically lose yield to frontrunning attackers.

Code Snippet

https://github.com/sherlock-audit/2026-04-delpho-update-apr-8th-2026/blob/main/delpho_contracts/src/DelphoStaker.sol#L119-L126 https://github.com/sherlock-audit/2026-04-delpho-update-apr-8th-2026/blob/main/delpho_contracts/src/periphery/DelphoRouter.sol#L230 https://github.com/sherlock-audit/2026-04-delpho-update-apr-8th-2026/blob/main/delpho_contracts/src/periphery/DelphoRouter.sol#L244

Tool Used

Manual Review

Recommendation

Implement reward streaming instead of instant distribution.

Discussion

Oxklapouchy

Fix for original issue confirmed in [PR#225](#).

The reward is now streamed linearly over `rewardDuration` via `rewardRate` and checkpointed into `_totalAssets` in `_updateReward`, so a single `notifyRewardAmount` no longer produces an instantaneous share-price jump. The deposit-then-withdraw sandwich described in the original report is no longer profitable, since an attacker would have to stay staked across the whole streaming window to capture a meaningful share.

A side effect:

`_updateReward` accrues `rewardRate * elapsed` into `_totalAssets` with no guard on `totalSupply()`. When the staker is empty (`totalSupply == 0`, reachable at bootstrap or after the SP fully unstakeStable) or near-empty (a single dust share), the streamed reward inflates the share price against zero or negligible real shares. The next depositor is priced against

an inflated `_totalAssets`, mints zero shares, and donates their deposit; the streamed reward in an empty window is also stranded.

This is addressable by a simple **dead-share seed at deployment**: mint a small amount of shares to a burn destination against a matching `_totalAssets` seed funded by the deployer, so `totalSupply()` has a permanent non-zero floor that can never be withdrawn. This makes the `totalSupply() == 0` state unreachable and keeps the share price bounded.

Note that overriding `_decimalsOffset()` alone is not sufficient: virtual shares are not real supply, so they do not prevent `totalSupply()` from reaching 0 when all stakers withdraw, and in that state the dilution still scales with the streamed reward magnitude. The dead-share seed constrains `totalSupply()` itself, which is the variable both the dilution and the inflation key on. Adding the offset and `totalSupply() == 0` guards on `notifyRewardAmount / _updateReward / totalAssets()` is complementary, but the dead-share seed is the root fix.

Shivansh070707

@0xklapouchy so we should mint some shares at initialize ? should we take collateral asset too during initialize

0xklapouchy

@Shivansh070707 Yes, it needs to be the same as a standard deposit: 1 share for 1 deposited asset, so that `_totalAssets` correctly reflects this.

You can even use `mintAndStake()` for this, just mint 10e6 shares after deployment, before any `notifyRewardAmount()` call. Even if other users have already deposited, it will not matter. You can then either hold those shares or send them to a burn address.

Shivansh070707

@0xklapouchy added fix in [PR](#)

0xklapouchy

Share inflation risk is addressed in the deployment script (`_mintInitialSusdv` in `02_DeploymentScript.s.sol`): DelphoStaker is seeded at deploy time by depositing 2 WHYPE, minting 50 USDV, staking it, and burning the resulting sUSDV to `0xdead`. This sets a permanent non-zero `totalSupply()` floor backed by real USDV, so the empty/near-empty share-price inflation and first-depositor dilution are no longer reachable.

Issue M-3: Minimum position size not enforced after repayment allowing creation of dust positions that pollute sorted list and escape liquidation [RESOLVED]

Source:

<https://github.com/sherlock-audit/2026-04-delpho-update-apr-8th-2026/issues/80>

Summary

`repayStable` reduces `userPosition.usdvMinted` without enforcing that the remaining position is either zero or at least `minMint`. Users can mint the minimum, repay all but a Szabo, and leave the protocol with a dust position that persists forever. The same gap exists in `deLeverage` (which routes through `repayStable`) and partially in the fallback redemption dust guard when `redeemRemaining < totalDebt`. Combined with the low default `minMint = 1 USDV`, dust positions become a cheap and persistent griefing vector.

Vulnerability Detail

The check in `mintStable` validates the per-mint increment, not the resulting position size:

```
if (usdvAmount < configProvider.minMint()) revert DSM_INVALID_MINT_AMOUNT();
```

So any mint must be $\geq$ `minMint`, but the position that results from subsequent operations is unconstrained. After `repayStable`:

```
userPosition.usdvMinted = newBorrowed;  
userPosition.pendingInterest = newInterest;  
// no check that newBorrowed is 0 or >= minMint
```

A user can:

1. Mint exactly `minMint` (1 USDV)
2. Repay `minMint - 1 Szabo`
3. End up with a 1-Szabo position that persists in the sorted list

The same issue affects `deLeverage`, which computes `repayAmount` from a user-configured `deLeverageBps` of staked assets and forwards to `repayStable`. If the computation leaves a dust remainder, the bot silently creates a dust position.

Fallback redemption has a dust guard (`remainingPrincipal < minMint`) that attempts to absorb the full debt, but it fails when `redeemRemaining < totalDebt` because `targetUsd` gets capped at `redeemRemaining`, leaving a sub-`minMint` remainder.

Dust positions are exploitable because:

- They live permanently in `DelphoSortedDebts`, inflating traversal costs
- They consume `fallbackLiquidateMaxIterations` / `fallbackRedeemMaxIterations` slots with minimal USDV absorbed per iteration
- When they go underwater, liquidator fees (e.g. 1% of dust debt) are dwarfed by gas costs, so no one liquidates them
- An attacker can cheaply Sybil-flood: for 50 addresses, total cost is ~0.15 USDV in fees plus gas to create 50 permanent dust positions

The default `minMint = 1` USDV is also too low given the 6-decimal precision of USDV and the operational cost of on-chain bookkeeping. A minimum closer to 50–100 would materially raise the economic cost of spamming dust positions.

Impact

Attackers can cheaply create permanent dust positions that pollute the sorted list, waste fallback redemption/liquidation iteration budgets, and accumulate as uncollectible bad debt (since no rational liquidator will pay gas to liquidate a sub-cent position). Combined with the HyperEVM gas limits, enough dust positions can effectively freeze fallback operations or render them ineffective as a peg-defense mechanism.

Code Snippet

https://github.com/sherlock-audit/2026-04-delpho-update-apr-8th-2026/blob/main/delpho_contracts/src/core/DelphoStableManager.sol#L142-L156

https://github.com/sherlock-audit/2026-04-delpho-update-apr-8th-2026/blob/main/delpho_contracts/src/core/DelphoStableManager.sol#L666-L677

https://github.com/sherlock-audit/2026-04-delpho-update-apr-8th-2026/blob/main/delpho_contracts/src/configurators/base/DelphoVaultConfig.sol#L95

Tool Used

Manual Review

Recommendation

Enforce a minimum-position invariant after any flow that reduces `usdvMinted`. The position must be either zero (fully closed) or at least `minMint`:

```
userPosition.usdvMinted = newBorrowed;
userPosition.pendingInterest = newInterest;

if (userPosition.usdvMinted != 0 && userPosition.usdvMinted <
    ↪ configProvider.minMint()) {
```

```
    revert DSM_BELOW_MIN_POSITION();  
}
```

Apply the same check in `_processOneFallbackBorrower` partial branch (or force full absorption when the partial would leave dust). Alternatively, centralize the check inside `StableManagerLib.processRepayment`.

Additionally, raise the default `minMint` from `1e6` (1 USDV) to a value that makes dust spam economically unattractive e.g. `50e6–100e6`. The value should at minimum exceed the liquidator's break-even gas cost plus the storage rent implied by keeping a sorted-list node alive indefinitely.

Discussion

Oxklapouchy

The fix in [PR#201](#) only covers the easy half of this issue.

1. `repayStable` is fixed correctly, but silently drops a `PositionClosed` emit.

`DelphoStableManager.sol:158–162` now does:

```
if (userPosition.usdvMinted == 0 && userPosition.pendingInterest == 0) {  
    userPosition.active = false;  
} else if (userPosition.usdvMinted != 0 && userPosition.usdvMinted <  
    ↪ configProvider.minMint()) {  
    revert DSM_BELOW_MIN_POSITION();  
}
```

The PR stripped the `emit PositionClosed(user, currentPositionId, false)`; that lived right next to the `active = false` assignment. Repaying to zero in full now silently flips `active` without emitting `PositionClosed`.

`_processOneFallbackBorrower` full-close has the same gap. `usdApplied == totalDebt` zeros the position and sets `active = false`, but there is no `PositionClosed(borrower, currentPositionId[borrower], false)`. Same fix.

(Liquidation already emits `PositionClosed(user, currentPositionId[user], true)` at the end of `_liquidateUser`, so that path is consistent)

2. `_processOneFallbackBorrower` partial branch still leaks dust through the prediction/execution mismatch.

The dust guard at `DelphoStableManager.sol:708–725` predicts via `StableManagerLib.processRepayment` (interest-first), but the actual write a few lines below is the proportional branch with `mulDivUp`. Both formulas reduce total debt by the same amount, but they put a different number into `usdvMinted`. The guard can be satisfied by the prediction and the write can land below `minMint`.

This is the same prediction/execution mismatch the [#74](#) is pointing at: as long as the partial branch executes proportional while every other site (including this guard's

predictor) executes interest-first, the dust hole stays open against borrowers with non-zero pendingInterest.

Fix is the one the #74 comment proposes, replace the partial branch with a processRepayment call.

3. Default minMint was not raised.

DelphoVaultConfig.__DelphoVaultConfig_init_unchained still sets minMint = 1e6.

Oxklapouchy

Fix confirmed in [PR#259](#)

Issue M-4: Fallback redemption against an underwater borrower can create a stuck active position with debt but zero collateral that cannot be liquidated [RESOLVED]

Source:

<https://github.com/sherlock-audit/2026-04-delpho-update-apr-8th-2026/issues/81>

Summary

When fallback redemption targets an underwater borrower (collateral value less than total debt), `seizeFallbackCollateral` drains all available collateral but the partial-reduction branch runs because `usdApplied < totalDebt`. The borrower is left with `active = true`, `usdvMinted > 0`, and zero collateral. `_updateSortedDebt` removes them from the sorted list via its `coll.totalValue == 0` guard. The position can no longer be cleared by any flow: liquidation reverts on empty-array collateral, and nothing else writes to the stuck state. The residual debt is never recorded as `totalBadDebt`.

Vulnerability Detail

In `_processOneFallbackBorrower`, after `seizeFallbackCollateral` the function branches on `usdApplied == totalDebt`:

```
if (usdApplied == totalDebt) {
  borrowerPos.usdvMinted = 0;
  borrowerPos.pendingInterest = 0;
  borrowerPos.active = false;
} else {
  // partial branch: proportional reduction only
  borrowerPos.usdvMinted -= principalReduction;
  borrowerPos.pendingInterest -= interestReduction;
}
```

For an underwater borrower the seize takes the full collateral value (capped at per-token balances), so `actualUsd + totalValue < totalDebt`, meaning `usdApplied < totalDebt`. The partial branch runs: the position is reduced but not zeroed, and `active` stays true.

`_updateSortedDebt` is then called and hits its `totalValue == 0` guard, silently removing the borrower from the sorted list without touching the position state:

```
if (coll.totalValue == 0) {
  if (sorted.contains(_user)) sorted.remove(_user);
  return;
}
```

The position is now orphaned: active with positive debt, zero collateral, invisible to sorted-list-driven flows.

If someone later calls `liquidate` on this borrower, `validateLiquidation` passes (zero collateral with positive debt is flagged liquidatable), but `_processNormalLiquidation` calls `selectCollateralToRedeem` which returns empty arrays, and `_vault.debitCollateral` reverts with `DV_EMPTY_ARRAY`. The debt cannot be cleared and is never recorded as `totalBadDebt`.

Impact

Every underwater borrower hit by a fallback redemption creates permanent stuck debt:

- `totalBadDebt` understates the actual shortfall, breaking protocol accounting.
- `usdvMinted` ledger no longer reflects backed supply, breaking supply/debt conservation.
- The `_positions` slot is permanently occupied.
- USDV previously minted by this borrower remains in circulation with no collateral and no path to recovery.

Market stress periods produce many underwater positions simultaneously; fallback redemptions processing them cascade into a growing pile of stuck debt, silently undercollateralizing USDV without any on-chain signal.

Code Snippet

https://github.com/sherlock-audit/2026-04-delpho-update-apr-8th-2026/blob/main/delpho_contracts/src/core/DelphoStableManager.sol#L658-L680

https://github.com/sherlock-audit/2026-04-delpho-update-apr-8th-2026/blob/main/delpho_contracts/src/core/DelphoStableManager.sol#L609-L614

https://github.com/sherlock-audit/2026-04-delpho-update-apr-8th-2026/blob/main/delpho_contracts/src/core/DelphoStableManager.sol#L723-L733

Tool Used

Manual Review

Recommendation

Skip underwater borrowers entirely in `_processOneFallbackBorrower` and let the liquidation flow clear them. Liquidation uses `selectCollateralToRedeem`, which does not filter by token support, so it can seize disabled-token collateral that `seizeFallbackCollateral` skips - making it the correct path for underwater positions.

Add an early return when the borrower's total debt exceeds their collateral value:


```
CollateralDetails memory coll = _vault.getUserCollateralDetails(_current);  
if (coll.totalValue == 0 || totalDebt > coll.totalValue) return (0, 0);
```

This leaves the borrower in the sorted list with their collateral intact, ready for a liquidator to call `liquidate(user)`. Normal liquidation absorbs via the Stability Pool, records any shortfall as `totalBadDebt`, and removes the user cleanly. The `coll.totalValue == 0` guard in `_updateSortedDebt` also becomes unreachable and can be removed.

Discussion

Oxklapouchy

The `_isUnderwaterBefore` eligibility filter is correct for the fallback **redemption** path but too aggressive for the fallback **liquidation** path. The two flows have opposite goals, so they should not share the same threshold.

Redemption case (`_processOneFallbackBorrower`):

```
if (_isUnderwaterBefore(_current, lt, _vault)) return (0, 0, 0); // lt =  
↳ liquidationThresholdBps
```

Correct. Redemption must never seize collateral from a position that is already liquidatable; those belong to the liquidation flow, which records bad debt properly. Skipping `LTV >= liquidationThresholdBps` (80%) is the right boundary here.

Liquidation case (`_distributeFallbackDebt`):

```
if (!_isUnderwaterBefore(current, lt, _vault)) { // lt = liquidationThresholdBps  
    eligible[eligibleCount++] = current;  
}
```

Too strict. Fallback liquidation's goal is the opposite of redemption: distribute the liquidated debt + collateral across as many absorbers as possible so it does not become bad debt. The eligible set is hard-capped at 50 borrowers. Filtering out everyone with `LTV >= 80%` shrinks the absorber pool. A borrower at 85% or 90% LTV still has positive equity and can absorb a proportional debt + collateral share, but the current code excludes them. With fewer eligible absorbers, a large liquidation runs out of capacity and leaves debt undistributed, which is then recorded as bad debt unnecessarily.

The only borrowers who should be excluded here are those already in **bad debt** (`LTV >= 100%`): loading more debt onto a position whose collateral no longer covers its existing debt is pointless and creates orphaned debt. Everyone between the liquidation threshold and 100% is a valid absorber.

Reasoning: the function name `_isUnderwaterBefore` implies an "underwater" check (`LTV >= 100%`), but it is being called with `liquidationThresholdBps` (80%), turning it into an "is-liquidatable" check. That semantic is correct for redemption but wrong for liquidation distribution, where we want "is-not-yet-bad-debt."

Suggested fix: in `_distributeFallbackDebt`, pass `MAX_BPS` (100%) instead of `liquidationThresholdBps`:

```
// Liquidation distribution: include every borrower that is not yet in bad debt.
// Only LTV >= 100\% (collateral can no longer cover existing debt) is excluded;
// loading more debt onto a truly underwater borrower would just orphan it.
if (!_isUnderwaterBefore(current, MAX_BPS, _vault)) {
    eligible[eligibleCount++] = current;
}
```

Leave the redemption call at L699 unchanged (`liquidationThresholdBps`).

Trade-off worth acknowledging: including 80%-100% LTV borrowers means absorption can push some of them past 100% into bad debt themselves. That is accepted by design: given the 50-absorber cap, spreading the debt across more borrowers (some of whom may tip into bad debt) is preferable to leaving the liquidated position's debt undistributed and concentrated.

Issue M-5: Unlimited credit delegation and aToken approval to an unverified external HyperLoop contract [RESOLVED]

Source:

<https://github.com/sherlock-audit/2026-04-delpho-update-apr-8th-2026/issues/86>

Summary

DelphoHyperLendExecutor.initialize grants `type(uint256).max` credit delegation on the USDT variable debt token and `type(uint256).max` ERC-20 approval on the hYield (aToken) asset to the HyperLoop contract. The HyperLoop contract deployed at `0x2f5F23ED499ABcDef7116f75e3365C553D2b4913` is not verified on `hyperevmscan` and there is no on-chain confirmation that its bytecode matches the public GitHub source.

Vulnerability Detail

In the initializer:

```
IDebtVariableToken(cfg.debtVariableAsset()).approveDelegation(cfg.hyperLoop(),
↪  type(uint256).max);
IERC20(cfg.hYieldAsset()).forceApprove(cfg.hyperLoop(), type(uint256).max);
```

Both approvals are set once at deployment and persist forever. HyperLoop can, at any time:

- Borrow USDT on behalf of the executor up to the executor's available borrow capacity (via HyperLend credit delegation)
- Transfer the executor's aToken (hYield) balance

The github code (<https://github.com/hyperlendx/looping-contracts/blob/master/contracts/Looping.sol>) suggests the contract is not directly exploitable against a specific user, meaning only HyperLoop caller itself (not an arbitrary attacker) can trigger the callback that consumes the executor's approvals. However, this reasoning depends entirely on the deployed bytecode matching the public source, which cannot be verified because the HyperLoop contract is unverified on the block explorer.

Additional trust assumptions that compound the risk:

- HyperLoop may be upgradeable. If so, an upgrade could introduce logic that withdraws the executor's delegated borrow capacity or pulls aTokens without the initiator guard.
- Even without malicious intent, a bug in a future HyperLoop version could accidentally drain approvals that were left open indefinitely.

Impact

The executor's entire borrow capacity and full aToken balance are permanently exposed to an unverified external contract.

Code Snippet

https://github.com/sherlock-audit/2026-04-delpho-update-apr-8th-2026/blob/main/delpho_contracts/src/executor/DelphoHyperLendExecutor.sol#L40-L46

Tool Used

Manual Review

Recommendation

Do not grant persistent approvals in the initializer. Grant scoped approval for the exact amount needed at the start of each open/close operation, and reset to zero immediately after HyperLoop returns.

Discussion

DemoreXTess

Fix is not correct here: https://github.com/Delpho-labs/delpho_contracts/pull/217

Looping contract borrows `flashLoan` + premium amount of assets, delegation amount won't be enough.

```
debtToken.approveDelegation(hyperLoop, flashloanAmount);
```

<https://github.com/hyperlendx/looping-contracts/blob/101587a52872d3336b1adc5bed22f24ff0b6e4c4/contracts/Looping.sol#L182-L184>

Oxklapouchy

Fix confirmed in [PR#251](#)

Issue M-6: HyperLend close position offsets flash loan by existing USDT balance but does not use it for repayment, leaving an inflated slippage window on the collateral swap [RESOLVED]

Source:

<https://github.com/sherlock-audit/2026-04-delpho-update-apr-8th-2026/issues/87>

Summary

In `DelphoHyperLendExecutor._closePosition`, when the executor's existing USDT balance is positive but below `flashloanAmount`, the code subtracts `usdtBalance` from the flash loan request (`flash -= usdtBalance`) without either using the existing balance for repayment or adjusting `collateralToSwap`. The `minAmountOut` is then computed from the reduced flash, producing a slippage floor that can be arbitrarily loose relative to the actual swap size. On HyperEVM sandwich risk is lower than on mainnet, but the protection is still incorrect by design.

Vulnerability Detail

```
uint256 usdtBalance = IERC20(usdt).balanceOf(address(this));
uint256 flash = flashloanAmount;
if (usdtBalance >= flash) flash = 0;
else if (usdtBalance > 0) flash -= usdtBalance;

uint256 minAmountOut = flash + (flash * 5) / 10_000;

IHyperLoop(cfg.hyperLoop()).closePosition(
    pool, adapter, usdt, collateralAsset, flash, minAmountOut, path,
    ↪ collateralToSwap, deadline
);
```

Two related problems:

1. Existing `usdtBalance` is not used for debt repayment. The Looping contract repays debt and repays the flash loan using only the flash-loaned amount inside its callback; it has no knowledge of USDT already sitting in the Delpho executor. Subtracting `usdtBalance` from the flash loan only reduces how much is borrowed - it does not contribute to closing the debt. The existing balance remains idle in the executor after the close completes.

2. `minAmountOut` is recomputed from reduced flash but `collateralToSwap` is unchanged. The caller sizes `collateralToSwap` to produce enough USDT to repay the full `flashloanAmount` + premium. After reducing flash by `usdtBalance`, the slippage floor becomes `flash +`

5 bps - much smaller than the expected output of the unchanged `collateralToSwap`. The swap could return far less than `collateralToSwap` is worth and still satisfy `minAmountOut`.

Example:

- `flashloanAmount = 100 USDT, collateralToSwap = 100 USDT` worth of collateral
- `usdtBalance = 50 USDT`
- `flash = 100 - 50 = 50`
- `minAmountOut = 50 + 0.025 = ~50.025 USDT`
- The swap of ~100 USDT worth of collateral is allowed to return just 50.025 USDT
- A sandwich or adverse-path swap can drain up to ~50 USDT of value while still passing the `minAmountOut` check

The proportion scales: if the executor holds half the value of `collateralToSwap` as USDT, the slippage window opens to 50 percent.

Impact

- The close path tolerates severe slippage on the collateral swap whenever the executor holds some (but not enough) USDT. A malicious swap path, MEV sandwich (where feasible on HyperEVM), or a misconfigured adapter can extract a meaningful fraction of the collateral value.
- The existing USDT balance does not contribute to repayment.

Code Snippet

https://github.com/sherlock-audit/2026-04-delpho-update-apr-8th-2026/blob/main/delpho_contracts/src/executor/DelphoHyperLendExecutor.sol#L210-L226

Tool Used

Manual Review

Recommendation

- Use the existing balance directly via `pool.repay()` before the flash-loan path.
- Recompute `collateralToSwap` (and the resulting `minAmountOut`) from the new `flash + premium` amount so the slippage floor matches the actual swap size.

Or

- The simplest fix is to remove the `else if (usdtBalance > 0) flash -= usdtBalance` ; branch entirely and always flash-loan the full `flashloanAmount`.

Discussion

Shivansh070707

Before calling `partialClose` or `closePosition`, we call `repayAsset(usdt balance)` off-chain. So the recommendation would work here ?

Oxklapouchy

If you always plan to call `repayAsset()` before the close position calls, then a simple fix - removing `if (usdtBalance > 0) flash -= usdtBalance;` - will be sufficient, as you always want to control the flash loan amount and the `collateralToSwap` directly.

Oxklapouchy

Fix confirmed in [PR#216](#)

Issue M-7: Unbounded $O(n)$ traversal in sorted debt updates forces user operations off HyperEVM small blocks and breaks fallback flows at low borrower counts [RESOLVED]

Source:

<https://github.com/sherlock-audit/2026-04-delpho-update-apr-8th-2026/issues/96>

Summary

`DelphoSortedDebts._findInsertPosition` always falls through to a $O(n)$ walk from the head of the list because `_updateSortedDebt` fetches hints while the user is still present in the list. Every write operation that modifies LCR (mint, repay, withdraw, fallback redemption, fallback liquidation) therefore pays a traversal cost proportional to the number of active borrowers. On HyperEVM this pushes routine user operations off the 2M small-block ceiling (1-second finality) onto the 30M large-block ceiling (60-second finality) at a few hundred borrowers, and can cause fallback liquidation and fallback redemption to revert.

Vulnerability Detail

`_updateSortedDebt` fetches the user's current neighbors and passes them to `findInsertPosition`:

```
bool hasEntry = sorted.contains(_user);
(address prev, address next) = hasEntry ? sorted.getPrevNext(_user) : (address(0),
↪ address(0));
(address insertPrev, address insertNext) = sorted.findInsertPosition(lcr, prev,
↪ next);
```

Because the user has not yet been removed from the list, `data.nodes[prev].next` still points to the user, not to `next`. The hint validation in `_validInsertPosition` checks `data.nodes[prev].next == next` and returns false. Every call falls through to the on-chain fallback traversal, which walks from `data.head` one node at a time. The in-code comment describes this as " $O(1)$ re-insertion when already in the list", but that path is unreachable given the current ordering. Every reinsertion is $O(n)$.

There is also a specific case where the same call double-traverses. When the on-chain fallback traversal stops at the user's own node, it returns `(prevOfUser, user)` as the hint. After `reInsert` calls `remove(user)`, the returned `insertNext == user` is no longer in the list. The inner `insert` runs its own validation, fails, and performs a SECOND full traversal. This arises when the user's new LCR is slightly below their old LCR and no other borrower has LCR in the gap between them, precisely the common case of small interest-accrual drift. In this case a single `_updateSortedDebt` call costs $2 \times O(n)$.

Each traversal step reads two storage slots on a mapping entry (`lcr, next`), costing approximately 4200 gas per cold iteration. Single user operations incur one such traversal. Fallback flows incur many: `_distributeFallbackDebt` runs a deferred reorder loop that calls `_updateSortedDebt` for every touched borrower, and `_redeemFromBorrowers` calls it indirectly via `_processOneFallbackBorrower` for each iteration. Non-traversal overhead per call is also material: oracle reads, position updates, and external calls to the Vault and StableManager.

HyperEVM uses a dual-block architecture: 2M gas small blocks with 1-second finality, 30M gas large blocks with 60-second finality. User transactions that exceed the small-block limit are routed to large blocks automatically. Combining per-step cost and realistic per-call overhead gives the following rough breakpoints:

- A normal user operation (mint, repay, requestWithdrawal) costs approximately $150k + 4200 \times K$ gas, where K is the borrower's position from `data.head`. The 2M small-block ceiling is reached around $K = 440$, and the 30M large-block ceiling around $K = 7100$. Cost scales with position, not with total list size, so borrowers near the head remain unaffected at any list size and only tail borrowers hit the ceilings.
- Fallback redemption and fallback liquidation debt distribution cost approximately $\sim 10M + 210000 \times N$ gas, where N is the number of active borrowers. Both loops reorder most of the list per tx, so cost scales with list size regardless of individual positions. Both always exceed the 2M small-block ceiling and hit the 30M large-block ceiling around $N = 100$.
- Direct liquidation is $O(1)$ on the sorted list (only a `remove` call) and always fits in small blocks.

These are conservative estimates. The per-iteration overhead for fallback flows depends on the number of collateral tokens each borrower holds. Each additional token triggers a Pyth oracle read inside `getUserCollateralDetails`, so multi-collateral positions push the breakpoints lower. The double-traversal edge described above halves the position thresholds for normal user ops when it triggers.

Impact

- **Fallback liquidation and fallback redemption** can revert at the 30M large-block ceiling, disabling bad-debt distribution and peg defense well before the protocol reaches meaningful scale.
- **Normal user operations** are routed off the 2M small block (latency jumps from 1s to 60s) and potentially can revert at the 30M large-block ceiling affects tail borrowers only, who skew toward low-LCR positions most likely to need to act.
- **Interest-accrual double-traversal** halves both thresholds for users whose new LCR drifts slightly below their old LCR, which is the most frequent write scenario.

Code Snippet

https://github.com/sherlock-audit/2026-04-delpho-update-apr-8th-2026/blob/main/delpho_contracts/src/core/DelphoSortedDebts.sol#L147-L175

https://github.com/sherlock-audit/2026-04-delpho-update-apr-8th-2026/blob/main/delpho_contracts/src/core/DelphoStableManager.sol#L618-L624

https://github.com/sherlock-audit/2026-04-delpho-update-apr-8th-2026/blob/main/delpho_contracts/src/core/DelphoStableManager.sol#L1019-L1022

Tool Used

Manual Review

Recommendation

Perform a full gas-modeling pass against HyperEVM's dual-block architecture with the following targets:

- **Normal user operations** (mint, repay, requestWithdrawal, claim) must fit inside the 2M small-block ceiling at realistic list sizes, across all positions in the sorted list, so 1-second finality is preserved for the product's core UX. The budget needs to hold for borrowers at the tail of the list, not just near the head.
- **Fallback liquidation and fallback redemption** must fit inside the 30M large-block ceiling in worst-case conditions: maximum configured `fallbackRedeemMaxIterations`, multi-collateral borrowers with several Pyth reads per iteration, full sorted-list reorder, and the double-traversal edge on interest-accrual drift. These are emergency paths; reverting at scale disables the protocol's peg defense and bad-debt distribution precisely when they are needed.

Discussion

DemoreXTess

Partially addressed in https://github.com/Delpho-labs/delpho_contracts/pull/207.

The new sorted-debt insertion model has been optimized with shortcuts that handle common cases such as small LCR changes by beginning the search from the `prev` or `next` reference points. However, our gas-consumption testing across several core protocol operations indicates that the underlying $O(n)$ search remains a concern. The current implementation is safe to use up to a certain scale, but at larger scale some operations may fail due to excessive gas consumption.

Issue M-8: Sorted Debt List Becomes Stale Due to Price Changes and Interest Accrual [RESOLVED]

Source:

<https://github.com/sherlock-audit/2026-04-delpho-update-apr-8th-2026/issues/100>

Summary

`DelphoSortedDebts` maintains a linked list of borrowers ordered by their Liquidation Collateral Ratio (LCR), with the least-collateralized borrower at the head. The LCR stored per node is a snapshot captured at insert or reinsert time. Because the list is never automatically reordered when collateral prices move or when interest accrues, the ordering degrades continuously and no longer reflects true risk. The protocol's liquidation logic relies on this list to identify the next borrower to liquidate, making stale ordering a direct threat to liquidation correctness.

Vulnerability Details

Each node stores a cached `lcr` value:

```
data.nodes[user].lcr = lcr;
```

This value is set only during `insert` or `reInsert` calls. There is no mechanism to refresh it in response to on-chain price updates from the oracle or to interest that accrues between reinserts.

Two independent sources of staleness exist:

- 1. Collateral price drift.** The LCR is the ratio of a user's collateral value to their debt. Collateral value is denominated in USD and computed from oracle prices. As prices fluctuate, every stored `lcr` in the list becomes incorrect. A borrower whose collateral has dropped significantly may still appear deep in the list (safe) while a healthier borrower sits at the head.
- 2. Interest accrual mismatch.** Interest is accrued and tracked against `usdvMinted` (the principal), but the LCR used for sorting is computed against total debt (principal plus accrued interest). As time passes and interest grows, total debt increases for every borrower, but no reinsert is triggered. The relative ordering among borrowers shifts silently because different borrowers' debt grows at the different rate.

Attack Path

Price-based staleness:

1. Borrower A deposits low-quality collateral (volatile asset) and borrower B deposits high-quality collateral. Both are inserted with an LCR that places A deeper in the list (safer) and B near the head.
2. The volatile collateral price drops sharply. Borrower A's true LCR collapses but their node stays in its original position.
3. A liquidator reads `getHead()` expecting the most at-risk borrower and targets B, who is actually healthier.

Interest-based staleness:

1. All borrowers are inserted with accurate LCRs at a given point in time.
2. Interest accrues. Because interest accrual is based on `usdvMinted` uniformly instead of total debt, total debt rises for everyone at the different rates.

Impact

- Liquidations may target the wrong borrowers, allowing genuinely undercollateralized positions to accumulate bad debt while healthier positions are acted on unnecessarily.
- The protocol's ability to remain solvent depends on timely liquidation of the riskiest positions. A consistently stale list undermines this guarantee.
- Keepers and bots that rely on `getHead()` for efficient liquidation will produce incorrect results without performing their own off-chain LCR validation, increasing operational complexity and risk.

Recomendation

Fix is not trivial for this issue.

Discussion

Shivansh070707

We have `getDebtLCR` available on `DelphoVaultLens` which returns real-time LCR including current oracle prices and accrued interest. An off-chain keeper can walk the sorted list, compare cached vs fresh LCR for each position, and call `reInsert` for any stale nodes. Would this be an acceptable fix?

DemoreXTess

Yes this works. To be honest, sorted debts are always problem even for Liquity due to staleness of state variables. I believe there is no best way to resolve this. I just wanted to share this issue just so you know. There is a trade-off since you need to call `reInsert` constantly and it will cost you gas.

0xklapouchy

Acknowledged and accepted as Operational risk.

The on-chain list cannot be kept perfectly fresh because price drift and interest accrual are continuous, so any snapshot is stale the moment it is written. Refreshing the top region of the list (where liquidations and fallback flows enter) is sufficient for safety and cheap enough for a keeper to do every few blocks.

Shivansh070707

We made fix in PR , can u check updated sorteddebt contract , just to be sure no new bugs introduced with the fix @0xklapouchy

DemoreXTess

Fixed in https://github.com/Delpho-labs/delpho_contracts/pull/244

Issue M-9: Same Borrower Can Be Redeemed Multiple Times in a Single Fallback Redemption Pass [RESOLVED]

Source:

<https://github.com/sherlock-audit/2026-04-delpho-update-apr-8th-2026/issues/101>

Summary

In `_redeemFromBorrowers`, the `nextUser` pointer is captured from the sorted list before the current borrower is processed. Inside `_processOneFallbackBorrower`, `_updateSortedDebt` reorders the list based on the borrower's new LCR. If a partially redeemed borrower is re-inserted at a position that falls back in front of a node whose `next` pointer the loop will read next, that borrower will be visited again in the same pass and redeemed a second time.

Vulnerability Details

The loop reads:

```
address nextUser = sortedDebts.getNext(current);
```

before calling `_processOneFallbackBorrower`. That function calls `_updateSortedDebt` at the end, which calls `reInsert` on the current borrower based on their updated LCR. If after the partial redemption the borrower's LCR is still low enough to be placed before a node that the loop will visit next, and the loop's already-captured `nextUser` happens to point to a node whose own `next` is now the re-inserted borrower, the loop will reach that borrower a second time.

Concretely: the list is A (head) -> B -> C. `nextUser = B` is saved. A is processed and partially redeemed. `_updateSortedDebt` reinserts A between B and C. List becomes B -> A -> C. Loop sets `current = B`, saves `nextUser = sortedDebts.getNext(B) = A`, processes B, then sets `current = A` and processes A again.

Attack Path

1. Sorted list: A (worst) -> B -> C.
2. Redemption loop saves `nextUser = B`, processes A (reduces A's debt). `_updateSortedDebt` reinserts A at position between B and C: B -> A -> C.
3. Loop sets `current = B`, saves `nextUser = getNext(B) = A`. Processes B.
4. Loop sets `current = A`. A is processed a second time, seizing more collateral from A.

5. A's collateral is drained beyond what a single-pass redemption should allow.

Impact

A single redemption call can seize collateral from the same borrower more than once within one transaction. The borrower loses more collateral than their proportional debt share warrants. The redeemer receives excess collateral.

Recommendation

Capture all target addresses and skip them if we see the same user again in the list.

Discussion

Oxklapouchy

Fix confirmed in [PR#210](#)

Issue M-10: Remaining Collateral Returned to Liquidated Borrower Instead of Being Redistributed [RESOLVED]

Source:

<https://github.com/sherlock-audit/2026-04-delpho-update-apr-8th-2026/issues/104>

Summary

After `_processFallbackLiquidation` distributes debt and collateral across borrowers, any leftover seized collateral is credited back to the liquidated user via `_creditRemaining`. This means the liquidated borrower retains excess collateral while their debt has already been socialized to other borrowers, creating an asymmetric outcome that harms the absorbing borrowers' LTV positions.

Vulnerability Details

In `_processFallbackLiquidation`:

```
_creditRemaining(_data, _vault);
```

`_creditRemaining` reads `pendingSeizedCollateral` for each token and credits whatever remains back to `_data.user` (the liquidated borrower). Absorbing borrowers received new debt (`usdvMinted` increased) but did not receive the proportional collateral that the leftover represents. The liquidated borrower keeps both the leftover collateral and has their debt cleared.

Attack Path

1. User X is liquidated with 1000 USDV debt and 1100 USD worth of collateral.
2. Fallback distributes 1000 USDV of debt equally to 10 borrowers (100 USDV each).
3. After penalty and fee allocation, 80 USD worth of collateral remains unassigned.
4. `_creditRemaining` sends the 80 USD back to user X.
5. User X has no debt but holds collateral. Absorbing borrowers now carry higher LTV with no offsetting benefit.

Impact

Borrowers who absorb fallback debt see their LTV increase with no additional collateral benefit for the portion returned to the liquidated user. The liquidated user is effectively

bailed out with excess collateral. Over multiple liquidation events, this systematically weakens absorbing borrowers' positions and degrades overall protocol solvency.

Recommendation

Remaining collateral after fallback distribution should be allocated to the absorbing borrowers proportionally rather than returned to the already-liquidated borrower.

Discussion

Shivansh070707

In case the Stability Pool does not have enough to cover the debt, we distribute the debt + collateral to other users in fallback liquidation. Any remaining collateral will be given back to the liquidated user.

DemoreXTess

Hey @Shivansh070707, yes you're correct. This issue is business logic related. Using current liquidation config values, first 50 borrowers will get 9% more collateral value but in fact this is very bad thing for them because liquidation threshold is already 80%. Let say \$10,000 debt is liquidated at fallback, it will return \$10,900 worth of collateral. If you calculate the LTV value of this distribution, it makes $\approx 92\%$ which extremely high.

In case of a big liquidation scenario, other borrowers will suffer from this a lot and they're not guilty in this incident. That's why I believe we should distribute 100% of the collateral to these 50 borrowers in fallback scenario in order to protect other borrowers as much as possible.

Actually, there is much better way to design it. Using 2 different liquidation method would be efficient. First one is this with 100% collateral redistribution to borrowers and second one is AAVE style liquidation. Liquidator pays whole debt, fee is sent to fee receiver and remaining debt is burnt. Liquidator receives debt + penalty worth of collateral and leftover can be given back to liquidated borrower. In this way, liquidator can choose both ways to liquidate.

Oxklapouchy

Fix confirmed in [PR#243](#)

Issue M-11: Oracle staleness and price-bound reverts DoS liquidation and redemption during market stress [RESOLVED]

Source:

<https://github.com/sherlock-audit/2026-04-delpho-update-apr-8th-2026/issues/116>

Summary

The Pyth to Chainlink migration added a staleness check and a per-token `[minPrice, maxPrice]` circuit breaker that revert inside `_getTokenPrice`.

Liquidation and redemption read collateral prices through the same path (`getUserCollateralDetails / getUserTotalCollateralValue -> getUsdValues -> _getTokenPrice`), so any token whose price is stale or outside its configured band reverts the entire operation.

Both reverts fire precisely during the crashes and de-pegs when liquidation and redemption are most needed, and unlike the previous Pyth pull-oracle there is no way for a liquidator to force a fresh price.

Vulnerability Details

`_getTokenPrice` now reverts on staleness and on the price band:

```
if (block.timestamp - updatedAt > stalenessAge) revert DO_STALE_PRICE();
...
if ((minPrice > 0 && normalizedPrice < minPrice) || (maxPrice > 0 &&
↪ normalizedPrice > maxPrice)) {
    revert DO_PRICE_OUT_OF_BOUNDS();
}
```

`getUsdValues` loops the user's basket and calls `_getTokenPrice` for each token; any single revert propagates and aborts the whole call, so one bad price on one token reverts the entire liquidation. Two distinct problems make this dangerous during stress:

1. **Price-bound revert on a real crash.** `minPrice/maxPrice` is meant to reject obviously-bogus feed data, but it cannot distinguish bogus data from a genuine crash or de-peg. When collateral legitimately falls below `minPrice`, every price read for that token reverts. The prod configuration registers WHYPE/KHYPE with `[5e6, 500e6]` (5–500 USD) and USDT/USDC with `[0.9e6, 1.1e6]` (0.90–1.10 USD) - bands a real HYPE crash below \$5 or a stablecoin de-peg below 0.90 USD will cross. The old Pyth oracle had no magnitude bound and would have accepted the crash price and liquidated normally, so this DoS is introduced by the migration. The bounds are also immutable post-registration (no setter), so the band cannot be widened mid-incident.

2. **Staleness revert with no liquidator recourse.** Chainlink push feeds update only on their heartbeat/deviation schedule and can lag or pause during volatility. If the lag exceeds `stalenessAge`, the read reverts. With Pyth (pull), a liquidator could submit a fresh price update in the same transaction before liquidating; with Chainlink (push) the liquidator cannot force an update and is stuck waiting for the feed.

The fallback path amplifies this: the eligibility scan reads `getUserTotalCollateralValue` for every eligible borrower, so a stale or out-of-bounds price on any one borrower's token reverts the entire fallback distribution of an unrelated liquidation.

Path:

1. A collateral asset crashes (HYPE below the configured 5 USD floor) or a stablecoin de-pegs (USDT/USDC below 0.90 USD).
2. The affected positions are now undercollateralized and must be liquidated.
3. `liquidate(user)` calls `getUserCollateralDetails(user)`, which reverts `DO_PRICE_OUT_OF_BOUNDS` (or `DO_STALE_PRICE` if the feed also lagged) on the crashing token.
4. Liquidation cannot proceed; redemption against the same collateral reverts identically. Bad debt accrues with no on-chain remedy, because the bounds have no setter to widen mid-crash.

Impact

Undercollateralized positions cannot be liquidated and USDV cannot be redeemed exactly during the crashes and de-pegs these mechanisms exist to handle, leaving the protocol accruing unrecoverable bad debt.

Code Snippet

https://github.com/Delpho-labs/delpho_contracts/blob/fac75aeeb8c589745ac6d7d620beee464432821a/src/core/DelphoOracle.sol#L124 https://github.com/Delpho-labs/delpho_contracts/blob/fac75aeeb8c589745ac6d7d620beee464432821a/src/core/DelphoOracle.sol#L130 https://github.com/Delpho-labs/delpho_contracts/blob/fac75aeeb8c589745ac6d7d620beee464432821a/src/core/DelphoOracle.sol#L64

Tool Used

Manual Review

Recommendation

Make the price read used by liquidation and redemption non-reverting on the price-quality checks: clamp the value to the breached `minPrice/maxPrice` (or fall back to a secondary source) rather than reverting, so liquidation can still proceed at a

conservative price during a genuine crash, and apply a liquidation-context degradation for staleness instead of a hard revert.

Discussion

JayusJay

@0xklapouchy should we even have min / max prices in the first place? We could just take it out, does that raise any new concerns?

DemoreXTess

@JayusJay If this is related with min price / max price suggestion from Chainlink docs, we don't need to check that most of the time because it's not even used anymore for almost every Chainlink price feed.

To make it guarantee, you can directly fetch min/max price from feed's aggregator but you need to make some changes in this case:

1. Normalizing min/max price will be needed because min/max price returned by aggregator may not match with our decimal base.
2. When feed updated, min/max price should be updated again based on new aggregator.

0xklapouchy

Removing min/max on its own does not introduce any new critical risk. They are only a hardening check, and they really only make sense in one place: the borrow / withdrawal flow. If an asset prints far outside a sane range there, you arguably should not keep treating it as valid collateral to borrow against, so reverting and blocking new exposure is acceptable.

For liquidation and redemption they are not needed and are actually harmful. There you want to act at any price, including a stale-but-present one, because not liquidating is worse than liquidating at a slightly off price. For example Aave works exactly this way: `AaveOracle.getAssetPrice` reads `latestAnswer`, has no staleness check and no min/max bounds, and only falls back to a secondary oracle when the price is ≤ 0 . It never reverts the liquidation path on price quality, it just returns the last Chainlink price and lets the liquidation proceed.

So the real point of the finding is not "keep or drop min/max." It is that the price read should be flow-aware:

- Liquidation / redemption: never revert on price quality (no min/max, no staleness revert). Use whatever price is reported so these flows always work.
- Borrow / withdrawal: revert when the collateral value cannot be validated. This is where min/max and the staleness check belong.

If you split it that way you can drop min/max from the shared read path entirely and keep a guard only on the borrow/withdrawal side. Dropping them outright is also fine, but note it does not fully resolve this issue by itself: the staleness revert still blocks

liquidation the same way, so that check also has to be made flow-aware (skipped in the liquidation/redemption path), otherwise a stale feed can still DoS liquidations.

JayusJay

Alright agreed and also instead of our own min / max prices which will be hard to reasonably define, we'll utilize the aggregator min/max to check

Oxklapouchy

@JayusJay Don't use aggregator min/max; it will never be correct. To my understanding, CL deprecated this and always sets it to a size range of [1, uint.max]. Either use internal min/max hardening settings or drop this entirely.

JayusJay

oh... if that's the case I lean towards removing it completely. How can we reliably set the minimum or max price for Hype? It is a volatile asset, at least that is my reasoning for arguing for removing it.

Oxklapouchy

As for the fix in the [PR#274](#)

`_getTokenPriceStrict` reverts when the price moves more than the allowed deviation vs the last stored price:

```
if (delta * BPS_DENOMINATOR * SECONDS_PER_HOUR
    > stored * uint256(bounds.maxDeviationBps) * (block.timestamp -
        ↪ _lastPriceAt[token])) {
    revert DO_PRICE_DEVIATION_TOO_HIGH();
}
```

The allowed move is `maxDeviationBps` per hour scaled from **zero** at `elapsed = 0`. But Chainlink updates in steps of its deviation threshold (HYPE ~0.5%, and the realized step is often a bit higher, ~0.55%). So right after a normal CL update the allowance is far below the step and strict ops revert until enough time passes.

With prod config (HYPE `maxDeviationBps = 2000`, 20%/hr) the allowance reaches the step size only after `step / 20\% * 1h`: a 0.5% step is blocked for ~90s, a 0.55% step for ~99s (a larger step blocks longer), repeating on every update. This DoSes mint/borrow/withdraw during ordinary volatility (liquidation/redemption use the lenient path and are unaffected). Two extra problems: the strict path is now non-view (stores price, costs SSTOREs), and `getUsdValuesStrict` is permissionless, so anyone can reset `_lastPriceAt` to grief the window.

It would be better to **drop the deviation guard entirely**. Chainlink push feeds are aggregated and cannot be flash-manipulated, so there is no manipulation vector for it to defend. The only residual risk is a user borrowing against a HYPE spike that then reverts (HYPE go UP and short after DOWN), and that is already covered by the LTV buffer (65% max borrow vs 80% liquidation threshold) plus the lenient liquidation path. Removing it

also lets the LTV validation go back to `view` with no stored price and no anchor-griefing surface, matching other lending protocols that rely on the feed plus the LTV buffer.

If you prefer to keep the current per-update mechanism, then:

(a) Add a base allowance equal to the CL deviation threshold plus margin so a normal step is always within bounds:

```
if (delta * BPS_DENOMINATOR * SECONDS_PER_HOUR
    > stored * baseDeviationBps * SECONDS_PER_HOUR
    + stored * uint256(bounds.maxDeviationBps) * (block.timestamp -
        ↪ _lastPriceAt[token])) {
    revert DO_PRICE_DEVIATION_TOO_HIGH();
}
```

with `baseDeviationBps` set per token to the feed's CL deviation threshold plus margin (e.g. 55 bps for a 0.5% feed).

(b) Gate the anchor update so `_lastPriceAt` cannot be advanced by a permissionless call.

Oxklapouchy

Fix confirmed in [PR#274](#) (a539842). The deviation guard was removed.

Issue M-12: Disabling a collateral token reverts price reads and blocks liquidation and withdrawal of existing holders [RESOLVED]

Source:

<https://github.com/sherlock-audit/2026-04-delpho-update-apr-8th-2026/issues/117>

Summary

`_getTokenPrice` reverts `DO_UNSUPPORTED_TOKEN` when a token's config is disabled. Disabling a token is intended to stop new deposits, but because every valuation flows through the price read, disabling a token that users still hold reverts all valuation, liquidation, withdrawal, and redemption involving any user whose basket contains it. The token's existing holders become stuck, and undercollateralized positions among them cannot be liquidated.

Vulnerability Details

The price read reverts whenever the token is disabled:

```
if (!tokenConfig.enabled || tokenConfig.decimals == 0 || tokenConfig.priceFeed ==  
    ↪ address(0)) {  
    revert DO_UNSUPPORTED_TOKEN();  
}
```

`disableToken` flips `enabled` to `false` without removing users' existing balances. After it is called, any operation that values a holder of that token (`getUserCollateralDetails` / `getUserTotalCollateralValue` -> `getUsdValues` -> `_getTokenPrice`) reverts on that token, and because the valuation loops the whole basket, one disabled token blocks every operation on that user. Liquidation, withdrawal, and redemption all revert. The `enabled` flag already correctly gates new deposits via `_isTokenSupported` in the deposit path, so reverting again at the valuation layer is harmful: it punishes existing holders rather than only preventing new exposure.

Impact

A token disabled while still held leaves those positions un-liquidatable and un-withdrawable, which can turn into stranded positions and unrecoverable bad debt if such a position is undercollateralized.

Code Snippet

https://github.com/Delpho-labs/delpho_contracts/blob/fac75aeeb8c589745ac6d7d620beee464432821a/src/core/DelphoOracle.sol#L108

https://github.com/Delpho-labs/delpho_contracts/blob/fac75aeeb8c589745ac6d7d620beee464432821a/src/configurators/base/DelphoTokenConfig.sol#L46

Tool Used

Manual Review

Recommendation

Gate the `enabled` flag only at the deposit boundary (where `_isTokenSupported` already enforces it), not at the valuation layer. Continue to return a price for existing holdings of a disabled token so those positions remain liquidatable, withdrawable, and redeemable. Do not return a zero price for disabled tokens, as zero-valuing collateral would itself trigger unfair liquidations.

Discussion

0xklapouchy

Fix confirmed in [PR#273](#)

Issue M-13: Chainlink feed parameters cannot be safely updated after token registration [RESOLVED]

Source:

<https://github.com/sherlock-audit/2026-04-delpho-update-apr-8th-2026/issues/118>

Summary

A token's Chainlink parameters are set once in `addToken` and cannot be safely changed afterward. `updateToken` swaps the `priceFeed` but never refreshes the cached `feedDecimals`s, so pointing a token at a feed with different decimals silently misprices it by orders of magnitude. The `minPrice`/`maxPrice` circuit-breaker bounds have no setter at all, so a too-tight or stale band cannot be corrected without redeploying the oracle config.

Vulnerability Details

`feedDecimals` is cached at registration from the feed and used on every read to normalize the price:

```
// addToken
_tokenConfigs[token] = TokenConfig({ ..., feedDecimals:
  ↪ AggregatorV3Interface(priceFeed).decimals(), minPrice: minPrice, maxPrice:
  ↪ maxPrice });
```

```
// updateToken - changes the feed but NOT feedDecimals, minPrice, or maxPrice
cfg.enabled = enabled;
cfg.priceFeed = priceFeed;
```

Two problems:

1. **Stale feedDecimals on a feed swap.** `_normalizePrice` scales the answer using the cached `feedDecimals`. If `updateToken` points the token at a feed whose decimals differ from the original (Chainlink USD feeds are commonly 8, ETH-denominated feeds 18), the cached value no longer matches the live feed and `_normalizePrice` is wrong by `10(diff)`. The collateral is silently mispriced by orders of magnitude, which flows straight into LTV checks, liquidation, and redemption. Unlike the previous Pyth integration (where the exponent arrived with each price), there is no self-correction.
2. **Immutable minPrice/maxPrice.** There is no setter for the circuit-breaker bounds. If a band is set too tight, or the asset's legitimate price range moves over time, the bound cannot be widened. Combined with the price-bound revert, a breached bound bricks the collateral (no valuation, liquidation, withdrawal, or redemption) with no on-chain recovery short of a config upgrade.

Impact

A feed change can silently misprice a collateral asset, and a too-tight bound permanently bricks an asset, both with no on-chain remediation path.

Code Snippet

https://github.com/Delpho-labs/delpho_contracts/blob/fac75aeeb8c589745ac6d7d620beee464432821a/src/configurators/base/DelphoTokenConfig.sol#L35
https://github.com/Delpho-labs/delpho_contracts/blob/fac75aeeb8c589745ac6d7d620beee464432821a/src/configurators/base/DelphoTokenConfig.sol#L22

Tool Used

Manual Review

Recommendation

Make `updateToken` (or a dedicated setter) update the full Chainlink parameter set atomically: when the `priceFeed` changes, re-read and store `feedDecimals` from the new feed, and allow `minPrice`/`maxPrice` to be updated. This keeps the cached decimals consistent with the live feed and lets a misconfigured or stale circuit-breaker band be corrected without a redeploy.

Discussion

Oxklapouchy

Fix confirmed in [PR#272](#)

Issue L-1: Re-leverage mint undershoots targetBps due to mixed rounding on LTV vs mint size [RESOLVED]

Source:

<https://github.com/sherlock-audit/2026-04-delpho-update-apr-8th-2026/issues/78>

Summary

`_reLeverageUser` derives `usdvToMint` from $(\text{targetBps} - \text{ltv}) * \text{totalValue} / \text{MAX_BPS}$ where `ltv` uses `mulDivUp` (ceiling) and the product uses `mulDiv` (floor). That double rounding systematically mints less than the debt implied by `targetBps.mulDiv(totalValue, MAX_BPS)`. A user can execute a successful re-leverage and still remain strictly below their configured `targetBps` in the same LTV definition.

Vulnerability Detail

```
uint256 ltv = borrowedAmount.mulDivUp(MAX_BPS, totalValue);
if (ltv >= targetBps) return;
uint256 usdvToMint = (targetBps - ltv).mulDiv(totalValue, MAX_BPS);
```

`mulDivUp` on debt-to-bps overstates current LTV whenever `borrowedAmount * MAX_BPS` is not divisible by `totalValue`. Subtracting that inflated `ltv` from `targetBps` shrinks the numerator before the mint path floors again in `mulDiv`. The result is not, in general, equal to the gap to the floored target debt cap `targetBps.mulDiv(totalValue, MAX_BPS) - borrowedAmount`.

Numerical example:

- `totalValue = 9999, targetBps = 5000 (50%), borrowedAmount = 3333`.
- `ltv = mulDivUp(3333, 10_000, 9999) = 3334`; the exact ratio is about 3333.333 bps.
- `3334 < 5000`, so the early return on `ltv >= targetBps` does not trigger.
- `usdvToMint = (5000 - 3334) * 9999 / 10_000 = 1666 * 9999 / 10_000 = 1665`.
- Floored target debt at 50%: `5000 * 9999 / 10_000 = 4999`; gap to that cap is `4999 - 3333 = 1666`, one wei more than the minted amount.
- After `mintStable`, debt is `3333 + 1665 = 4998`; new `ltv = mulDivUp(4998, 10_000, 9999) = 4999`.

So `mintStable(1665)` runs (non-zero re-leverage), but post-hoc LTV is 4999, which is still below `targetBps` of 5000. The user's opt-in target is not reached in one pass. A keeper or leverage bot that keys off the same LTV read will still see the position as under target and will try to re-leverage the user again on a later batch call.

Impact

Users with re-leverage enabled can sit slightly under their chosen `reLeverageTargetBps`. External bots or keepers that repeatedly invoke re-leverage paths for such positions waste gas on calls that do not reliably land on the configured target.

Code Snippet

https://github.com/sherlock-audit/2026-04-delpho-update-apr-8th-2026/blob/main/delpho_contracts/src/periphery/DelphoRouter.sol#L419-L428

Tool Used

Manual Review

Recommendation

Derive mint size from the floored target debt cap so it matches the usual “bps of collateral” conversion and avoids mixing `ceil(LTV)` into the mint:

```
uint256 targetDebt = targetBps.mulDiv(totalValue, MAX_BPS);  
if (borrowedAmount >= targetDebt) return;  
uint256 usdvToMint = targetDebt - borrowedAmount;
```

Discussion

0xklapouchy

Fix confirmed in [PR#227](#)

Issue L-2: USDV sent directly to the DelphoStaker contract is permanently locked [RESOLVED]

Source:

<https://github.com/sherlock-audit/2026-04-delpho-update-apr-8th-2026/issues/79>

Summary

DelphoStaker tracks deposits via an internal `_totalAssets` counter. USDV tokens sent directly to the Staker contract (bypassing `deposit` or `notifyRewardAmount`) are not reflected in `_totalAssets` and are not recoverable, since `recoverERC20` explicitly blocks recovery of the vault asset.

Vulnerability Detail

Any USDV transferred directly to the Staker address is stuck: `totalAssets()` returns `_totalAssets` so the tokens never affect share price, and `recoverERC20` reverts with `DS_WITHDRAW_STAKING_TOKEN` when called with the asset token.

Impact

Accidental direct transfers of USDV to the Staker contract are permanently lost.

Code Snippet

https://github.com/sherlock-audit/2026-04-delpho-update-apr-8th-2026/blob/main/delpho_contracts/src/DelphoStaker.sol#L111-L113

https://github.com/sherlock-audit/2026-04-delpho-update-apr-8th-2026/blob/main/delpho_contracts/src/DelphoStaker.sol#L146-L150

Tool Used

Manual Review

Recommendation

Add an admin-only `sweepExcessAsset` function that transfers `balanceOf(this) - _totalAssets` to a specified address.

Discussion

0xklapouchy

Fix confirmed in [PR#236](#)

Issue L-3: Unbounded loop in `_cancelExcessPendingWithdrawals` can grief liquidation and fallback redemption [RESOLVED]

Source:

<https://github.com/sherlock-audit/2026-04-delpho-update-apr-8th-2026/issues/82>

Summary

`_cancelExcessPendingWithdrawals` in the Vault iterates over `userActiveEpochs[user]` with no upper bound. A user can accumulate many distinct target epochs over time through small `requestWithdrawal` calls, then self-position at the head of the sorted debt list with multiple accounts. Any future `debitCollateral` against this user (liquidation or fallback redemption) triggers the cancel loop, which can exceed the HyperEVM large-block 30M gas limit and revert the entire operation.

Vulnerability Detail

Every seize path in the protocol calls `debitCollateral` in the Vault, which unconditionally invokes `_cancelExcessPendingWithdrawals(user, token)` after reducing the user's collateral. When pending withdrawals exceed remaining collateral, that function walks `userActiveEpochs[user]` and trims requests newest-first until the invariant holds:

```
uint256 epochCount = userActiveEpochs[user].length();
// ...
for (uint256 i = epochCount; i > 0; i--) {
    uint48 epochId = uint48(userActiveEpochs[user].at(i - 1));
    // ~2 cold SLOADs + conditional updates per iteration
}
```

`userActiveEpochs` is an `EnumerableSet` keyed by `epochId`. Entries are added in `_queueWithdrawal` each time a user queues a request for a distinct target epoch:

```
targetEpoch = currentEpoch + configProvider.withdrawalWaitEpochs();
if (block.timestamp + withdrawalRequestCutoff > epochData[currentEpoch].endTime)
    ++targetEpoch;
userActiveEpochs[user].add(targetEpoch);
```

Within a single epoch, the target is fixed, so a user accumulates new entries only across epoch boundaries. But there is no maximum size, and the only way to remove entries is by claiming matured withdrawals via `withdrawCollateral(epoch)` one epoch at a time. A user who never claims can grow the set indefinitely.

1. The attacker maintains a position with a healthy LCR (positioned safely in the middle/tail of `DelphoSortedDebts`).

2. Over many epochs, the attacker makes tiny `requestWithdrawal` calls - 1 wei is allowed. Each new epoch adds a fresh entry to `userActiveEpochs[attacker]`.
3. Attacker deliberately does not claim matured withdrawals.
4. After accumulating N entries, attacker performs a large `requestWithdrawal` that drains most of their collateral via instant withdrawal when possible.
5. The attacker's LCR is now at the head of the sorted-debt list -> first target of fallback redemptions and a prime liquidation candidate.
6. When any seize operation tries to debit collateral from the attacker:
 - `debitCollateral` -> `_cancelExcessPendingWithdrawals` -> iterates N epochs
 - At ~5,000-10,000 gas per iteration (two cold SLOADs + branches + writes), N around 3000 can exhaust the 30M large-block gas limit
 - The whole seize reverts

The attacker is griefing the protocol (fallback redemption is blocked when attacker positions are at the head). On a volatile day, this is enough to freeze peg defense.

The attack is slow to set up (requires presence across many epochs), but it's also slow to detect because the cost is invisible until a seize actually fires. Even non-malicious heavy users could organically accumulate enough epoch entries to become accidentally gas heavy to liquidate.

Impact

A user can accumulate arbitrary entries in `userActiveEpochs` through normal `requestWithdrawal` calls spread across epochs. Once the set grows large enough, any operation that seizes collateral from them may exceed the block gas limit and revert.

Code Snippet

https://github.com/sherlock-audit/2026-04-delpho-update-apr-8th-2026/blob/main/delpho_contracts/src/core/DelphoVault.sol#L603

https://github.com/sherlock-audit/2026-04-delpho-update-apr-8th-2026/blob/main/delpho_contracts/src/core/DelphoVault.sol#L526

Tool Used

Manual Review

Recommendation

Cap `userActiveEpochs[user].length()` at a protocol-defined maximum (e.g. 10-20 active epochs per user) enforced in `_queueWithdrawal`. When the cap is reached, `requestW`

`ithdrawal` should revert with a clear error, forcing the user to claim matured withdrawals before queuing new ones.

Discussion

0xklapouchy

Fix confirmed in [PR#228](#)

Issue L-4: HyperLend executor cannot close position when executor already holds enough USDT to cover the debt [RESOLVED]

Source:

<https://github.com/sherlock-audit/2026-04-delpho-update-apr-8th-2026/issues/83>

Summary

In `DelphoHyperLendExecutor._closePosition`, when the executor's existing USDT balance is greater than or equal to the required `flashloanAmount`, the code sets `flash = 0` and proceeds to call `HyperLoop.closePosition(..., flash=0, ...)`. This initiates an HyperLend flash loan with `amount=0`, which HyperLend allows, but the callback then calls `pool.repay(debtAsset, 0, 2, user)` which reverts with `INVALID_AMOUNT`. The close flow is bricked in exactly the scenario where it could have succeeded trivially via a direct repay.

Vulnerability Detail

The offending branch:

```
uint256 usdtBalance = IERC20(usdt).balanceOf(address(this));
uint256 flash = flashloanAmount;
if (usdtBalance >= flash) flash = 0;    // ← sets flash to zero
// ...
IHyperLoop(cfg.hyperLoop()).closePosition(pool, adapter, usdt, collateralAsset,
    ↪ flash, ...);
```

`HyperLoop.closePosition` issues `pool.flashLoanSimple(..., amount=flash, ...)`. HyperLend `validateFlashloanSimple` does not check for zero amount, so the flash loan proceeds with `amount=0`. The `executeOperation` callback then calls `pool.repay(debtAsset, repaymentAmount, 2, user)` where `repaymentAmount` is derived from the flash-loan amount (zero). HyperLend `validateRepay` explicitly reverts on zero:

```
require(amountSent != 0, Errors.INVALID_AMOUNT);
```

The entire close transaction reverts. The executor cannot close its position until either the existing USDT balance drops below `flashloanAmount` or an alternative close path is introduced.

Impact

Close operations fail whenever the executor happens to hold USDT greater than or equal to the debt being closed.

Code Snippet

https://github.com/sherlock-audit/2026-04-delpho-update-apr-8th-2026/blob/main/delpho_contracts/src/executor/DelphoHyperLendExecutor.sol#L210-L226 <https://github.com/hyperlendx/looping-contracts/blob/master/contracts/Looping.sol#L211>

Tool Used

Manual Review

Recommendation

When `usdtBalance >= flashloanAmount`, skip the flash-loan path entirely and call `pool.repay` directly with the executor's existing USDT, then withdraw and swap collateral without invoking `HyperLoop.closePosition`.

Discussion

Shivansh070707

For such cases , in order to withdraw collateral , either we need seperate function to withdraw or pass some flashloan amount ?

Oxklapouchy

The fix removes the auto-offset that caused the original revert, but the recommended behavior (skip the flash loan when the executor already has enough debt-asset) was not implemented in `DelphoHyperLendExecutor._closePosition`.

The offset was removed without a replacement, so a handler that already holds enough debt-asset and passes a non-zero `flashloanAmount` now pays the full `FLASHLOAN_PREMIUM_TOTAL` on every close. The original code's intent was to avoid the premium in that case.

Minimal patch: add an `if (flashloanAmount == 0)` branch at the top of `_closePosition` that does `pool.repay(debtAsset_, balanceOf(this), 2, this)` and returns (or proceeds to a swap-only path if `collateralToSwap > 0`), mirroring `DelphoMorphoExecutor._closeLeveragePosition`. Update `partialClosePosition`'s early-return condition accordingly.

Shivansh070707

Added fix in [PR](#) . @Oxklapouchy

Oxklapouchy

Fix confirmed in [PR#253](#)

Issue L-5: Redundant interest accrual writes when borrow index delta is zero waste gas in multi-step flows [RESOLVED]

Source:

<https://github.com/sherlock-audit/2026-04-delpho-update-apr-8th-2026/issues/84>

Summary

`_updateInterest` performs storage writes to `pendingInterest`, `borrowIndexSnapshot`, and `lastUpdatedAt` even when `userBorrowIndexDiff == 0`, which means no interest has accrued. The case is hit repeatedly in composite flows that call `updateInterest` more than once per transaction, such as `depositAndMint` and `repayAndWithdraw`.

Vulnerability Detail

After the early-return checks, the function computes:

```
uint256 userBorrowIndexDiff;  
unchecked {  
    userBorrowIndexDiff = borrowIndex - previousIndex;  
}  
uint256 interest = userPosition.usdvMinted.mulDivUp(userBorrowIndexDiff,  
    ↪ previousIndex);  
  
userPosition.pendingInterest += interest;  
userPosition.borrowIndexSnapshot = borrowIndex;  
userPosition.lastUpdatedAt = uint48(block.timestamp);
```

When `userBorrowIndexDiff == 0`, the interest is zero and the snapshot already equals `borrowIndex`. The three subsequent `SSTOREs` write the same values they already hold (or write `block.timestamp` to a field that no on-chain logic reads).

This occurs whenever:

1. `_updateInterest` is called more than once in the same transaction for the same user (the second call finds the snapshot already current).
2. The interest rate is set to zero for a period, so `borrowIndex` does not advance despite time passing.

Affected composite flows:

- `Router.depositAndMint`: `Vault.collateralDeposit` calls `updateInterest(user)` first, then `StableManager.mintStable` calls `_updateInterest(user, true)` - the second call hits `diff == 0`.

- Router.repayAndWithdraw: StableManager.repayStable calls _updateInterest(user, false), then Vault.requestWithdrawal calls updateInterest(user) again - the second call hits diff == 0.

Impact

Gas is wasted on every execution of the affected flows.

Code Snippet

https://github.com/sherlock-audit/2026-04-delpho-update-apr-8th-2026/blob/main/delpho_contracts/src/core/DelphoStableManager.sol#L576-L587

Tool Used

Manual Review

Recommendation

Add an early return when userBorrowIndexDiff is zero:

```
unchecked {
    userBorrowIndexDiff = borrowIndex - previousIndex;
}
if (userBorrowIndexDiff == 0) {
    return (userPosition, 0);
}
```

Discussion

Oxklapouchy

Fix confirmed in [PR#230](#)

Issue L-6: Borrow index rounds down while per-user interest rounds up, creating inconsistent and non-standard rounding policy [RESOLVED]

Source:

<https://github.com/sherlock-audit/2026-04-delpho-update-apr-8th-2026/issues/85>

Summary

`VaultMath.calculateBorrowIndex` uses `mulDiv` (rounds DOWN) twice when computing the new global `borrowIndex`, while `_updateInterest` uses `mulDivUp` (rounds UP) when computing the per-user interest share. The directions conflict and deviate from the convention of rounding borrow indexes up to favor the protocol.

Vulnerability Detail

The index update rounds down:

```
uint256 scaledFactor = FixedPointMathLib.mulDiv(PRECISION, numerator, denominator);
uint256 multiplier = PRECISION + scaledFactor;
borrowIndex = FixedPointMathLib.mulDiv(prevIndex, multiplier, PRECISION);
```

Per-user interest rounds up:

```
uint256 interest = userPosition.usdvMinted.mulDivUp(userBorrowIndexDiff,
↪ previousIndex);
```

The mixed directions pull in opposite ways: the global index grows slightly slower than ideal (user-favor), while the per-user calculation charges slightly more per unit of index growth (protocol-favor). Industry convention rounds borrow indexes up to guarantee the protocol never undercharges.

Impact

Inconsistent rounding policy that leaks fractional-wei interest and deviates from standard CDP conventions.

Code Snippet

https://github.com/sherlock-audit/2026-04-delpho-update-apr-8th-2026/blob/main/delpho_contracts/src/libs/VaultMath.sol#L90-L97

https://github.com/sherlock-audit/2026-04-delpho-update-apr-8th-2026/blob/main/delpho_contracts/src/core/DelphoStableManager.sol#L581

Tool Used

Manual Review

Recommendation

Switch both `mulDiv` calls in `calculateBorrowIndex` to `mulDivUp` for consistency with the per-user `mulDivUp`.

Discussion

0xklapouchy

Fix confirmed in [PR#231](#)

Issue L-7: DelphoStableManager initialize does not emit BorrowDetailsUpdated [RESOLVED]

Source:

<https://github.com/sherlock-audit/2026-04-delpho-update-apr-8th-2026/issues/88>

Vulnerability Detail

`_updateBorrowState()` emits `BorrowDetailsUpdated` whenever `borrowIndex` and `borrowIndexTimestamp` are updated, but `initialize()` sets the initial values (`borrowIndex = 1e27` and `borrowIndexTimestamp = block.timestamp`) without emitting the same event. Off-chain indexers and monitoring systems that rely on `BorrowDetailsUpdated` to reconstruct borrow-state history can miss the initial baseline values and start from incomplete state.

Recommendation

Emit `BorrowDetailsUpdated(1e27, uint48(block.timestamp))` at the end of `initialize()` after assigning the initial borrow parameters.

Discussion

Oxklapouchy

Fix confirmed in [PR#233](#)

Issue L-8: Short leg sizing uses inconsistent rounding between HyperLend and Morpho executors [RESOLVED]

Source:

<https://github.com/sherlock-audit/2026-04-delpho-update-apr-8th-2026/issues/89>

Vulnerability Detail

The HyperLend and Morpho executors both compute the USDT short leg size from the same portfolio-value formula but round differently:

DelphoHyperLendExecutor.sol (line 82):

```
uint256 shortCol = (portfolio * shortSizeInBps / MAX_BPS) / shortLeverage;
```

Both divisions round down.

DelphoMorphoExecutor.sol (line 94):

```
uint256 shortCol = (portfolio.mulDivUp(shortSizeInBps, MAX_BPS)) / shortLeverage;
```

First step rounds up, second rounds down.

For the same portfolio, shortSizeInBps, and shortLeverage, the Morpho executor borrows up to 1 wei more than the HyperLend executor. The two executors are meant to implement the same economic strategy on different lending venues, so their debt sizing should match by construction.

Recommendation

Pick one convention and apply it to both executors. Either `mulDivUp` on the first step in both (slightly larger short, protocol-favorable) or plain `mulDiv` in both (slightly smaller short, user-favorable).

Discussion

0xklapouchy

Fix confirmed in [PR#232](#)

Issue L-9: Liquidator fee cap is too high given the liquidator provides no capital [RESOLVED]

Source:

<https://github.com/sherlock-audit/2026-04-delpho-update-apr-8th-2026/issues/90>

Summary

The default `maxLiquidatorFee = 4006e6` (~4006 USD) is calibrated against a Liquity(BOLD)-style model where the liquidator provides the USDV to repay the debt. In Delpho, the Stability Pool absorbs the debt from its staked balance and the liquidator only calls the function - they risk nothing beyond gas. The incentive is overpaid by orders of magnitude and erodes protocol revenue on every liquidation.

Vulnerability Detail

`computeLiquidationValues` in `VaultMath`:

```
liquidatorFee = totalDebt.mulDiv(config.liquidatorRewardBps(),  
    ↪ MAX_BPS).min(config.maxLiquidatorFee());
```

With defaults `liquidatorRewardBps = 100` (1%) and `maxLiquidatorFee = 4006e6` (4006 USD), any liquidation where debt exceeds ~400600 USD pays the liquidator 4006 in USDV.

In Liquity the liquidator burns their own BOLD to cancel the liquidated debt and receives the seized collateral. The fee compensates for capital provisioning, unwinding collateral on mainnet DEXs during stress, and bearing smart-contract/price risk. A high cap is justified.

In Delpho:

- The Stability Pool covers the debt by burning its own staked USDV.
- The liquidator does not provide USDV, does not hold collateral, and bears no slippage or price risk.
- The liquidator's only cost is the gas to call `liquidate(user)` - on HyperEVM this is fractions of a cent.
- The fee is paid in USDV directly from the SP's staker withdrawal, which is protocol-owned reserves.

Paying up to 4006 USDV for a transaction that costs pennies in gas represents a significant leak of protocol reserves on every liquidation of a large position.

Impact

Protocol reserves are drawn from the Stability Pool's staked USDV to pay an oversized bot fee on every liquidation.

Code Snippet

https://github.com/sherlock-audit/2026-04-delpho-update-apr-8th-2026/blob/main/delpho_contracts/src/libs/VaultMath.sol#L142

https://github.com/sherlock-audit/2026-04-delpho-update-apr-8th-2026/blob/main/delpho_contracts/src/configurators/base/DelphoVaultConfig.sol#L79

Tool Used

Manual Review

Recommendation

Lower `maxLiquidatorFee` to around 50 USDV. The 1% `liquidatorRewardBps` can stay as-is (it is a reasonable proportional incentive for small positions), but the cap should reflect the actual work: the liquidator just triggers the function while the SP provides the capital.

Discussion

0xklapouchy

Fix confirmed in [PR#222](#)

Issue L-10: HyperLend executor's flash loan premium calculation rounds down and hardcodes 5 bps [RE-SOLVED]

Source:

<https://github.com/sherlock-audit/2026-04-delpho-update-apr-8th-2026/issues/91>

Summary

In `DelphoHyperLendExecutor._closePosition`, the `minAmountOut` passed to the collateral swap is computed as `flash + (flash * 5) / 10_000`. Integer division rounds down, while `HyperLend PercentageMath.percentMul` uses round-half-up via the `HALF_PERCENTAGE_FACTOR = 5000` offset. The two calculations can differ by 1 wei at rounding boundaries. The premium rate is also hardcoded at 5 bps rather than read from the pool's `FLASHLOAN_PREMIUM_TOTAL`, so any future rate change on the HyperLend pool would permanently break the close path.

Vulnerability Detail

The executor uses:

```
uint256 minAmountOut = flash + (flash * 5) / 10_000;
```

HyperLend premium on the repay side computes:

```
premium = (flash * 5 + 5000) / 10000;
```

The two diverge by 1 wei whenever $(\text{flash} * 5) \% 10000 \geq 5000$. In that case, Delpho's `minAmountOut` is exactly the amount the swap must produce to satisfy the floor, but HyperLend requires 1 wei more for flash loan repayment. If the swap returns the bare minimum, the flash loan repayment fails and the entire close transaction reverts.

Additionally, 5 is a magic number. HyperLend exposes `FLASHLOAN_PREMIUM_TOTAL` which is configurable by the pool owner. If HyperLend ever changes it (e.g., to 9 bps), Delpho's hardcoded 5 undercounts the premium systematically and every close transaction reverts until the constant is updated and the executor redeployed or reconfigured.

Impact

Close transactions may revert at the exact rounding boundary when the swap returns precisely `minAmountOut`. In practice the adapter typically returns more than the floor, providing natural slack, so the rounding divergence rarely manifests. The hardcoded rate dependency is the bigger latent risk: an external protocol's configuration change can brick the close path without code review or warning.

Code Snippet

https://github.com/sherlock-audit/2026-04-delpho-update-apr-8th-2026/blob/main/delpho_contracts/src/executor/DelphoHyperLendExecutor.sol#L216

Tool Used

Manual Review

Recommendation

Read the premium rate from the pool and match HyperLend rounding convention:

```
uint256 premiumBps = IHyperLendPool(pool).FLASHLOAN_PREMIUM_TOTAL();  
uint256 premium = (flash * premiumBps + 5000) / 10000;  
uint256 minAmountOut = flash + premium;
```

Discussion

Shivansh070707

with the current FLASHLOAN_PREMIUM_TOTAL = 4 , does rounding down have any issues ? or do we scaleup ?

0xklapouchy

Fix confirmed in [PR#221](#)

Issue L-11: DelphoStableManager.validateLTV returns stale results because it does not accrue pending interest [RESOLVED]

Source:

<https://github.com/sherlock-audit/2026-04-delpho-update-apr-8th-2026/issues/92>

Summary

`DelphoStableManager.validateLTV` is a publicly callable view function that reads `userPosition.pendingInterest` from storage without first advancing the global `borrowIndex` or materializing the user's accrued interest. Any external on-chain caller or off-chain consumer reading this function gets a debt figure based on the last time the user's position was touched, which understates the real total debt whenever interest has accrued since.

Vulnerability Detail

`validateLTV` reads debt directly from storage:

```
function validateLTV(address user) external view override {
    UserPosition memory position = _positions[user][currentPositionId[user]];
    uint256 debt = position.usdvMinted + position.pendingInterest;
    // ...
    VaultMath.validateLtv(debt, collateralDetails.totalValue,
        ↪ configProvider.maxLtvBps());
}
```

`pendingInterest` is only materialized when `_updateInterest` runs as part of a state-changing operation (`mintStable`, `repayStable`, `redemption`, `liquidation`, etc.). Between such operations, the stored value is frozen while real interest continues to accrue via the global `borrowIndex`. A view caller therefore sees a snapshot that is stale by however long the user has been idle.

Impact

Off-chain and external on-chain consumers of `validateLTV` receive a debt figure that may be materially lower than the user's true debt, leading to incorrect health assessments. Bots or integrations that trust the result can miss liquidatable positions, mis-size redemptions, or display incorrect LTVs in user interfaces.

Code Snippet

https://github.com/sherlock-audit/2026-04-delpho-update-apr-8th-2026/blob/main/delpho_contracts/src/core/DelphoStableManager.sol#L450-L458

Tool Used

Manual Review

Recommendation

Either document the coupling explicitly, or make the view self-sufficient by computing a live debt value inside the function using the current `borrowIndex` and the user's `borrowIndexSnapshot`, without writing to storage. A pure-view variant that replicates `_updateInterest`'s arithmetic produces an accurate health reading for any external consumer without requiring a prior state-changing call.

Discussion

Oxklapouchy

Fix confirmed in [PR#220](#)

Issue L-12: Redemption peg defense becomes ineffective under stress because the shared 5% buffer empties quickly and remaining capital is locked in executor positions [RESOLVED]

Source:

<https://github.com/sherlock-audit/2026-04-delpho-update-apr-8th-2026/issues/93>

Summary

When a user deposits collateral, 95% goes to `fundsForExecutor` and only 5% goes to `globalBufferFunds`. The 5% buffer is the only source of instant-settled collateral for both voluntary withdrawals and redemption arbitrage. During peg stress, redeemers drain this buffer within a handful of transactions, and subsequent redeemers receive only queued epoch promises maturing days later. Because executor funds are deployed into HyperLend, Morpho, and HyperCore positions that require bot-orchestrated multi-step unwinds, the buffer cannot be refilled fast enough to sustain peg defense.

Vulnerability Detail

`_handleDeposit` splits every collateral deposit:

```
uint256 bufferAmount = (amountAfterFee * configProvider.bufferBps()) / MAX_BPS;
uint256 executorAmount = amountAfterFee - bufferAmount;
tokenAccounting[token].globalBufferFunds += bufferAmount;
tokenAccounting[token].fundsForExecutor += executorAmount;
```

With `bufferBps = 500` (5%), the protocol routinely holds only 5% of total collateral as instant-withdrawable liquidity. The remaining 95% is earmarked for executors and rapidly deployed into external protocols.

The redemption flow (`Router.redeem` → `_handleRemainingCollateralAfterRedemption` → `Vault.requestWithdrawal` → `_drainBuffer`) serves the redeemer from the same `globalBufferFunds` pool used by voluntary withdrawals. There is no dedicated reserve for redemption. The buffer is also continuously consumed by normal user operations during regular market conditions, so at the moment peg stress arrives, the buffer is often already partially depleted.

When the buffer runs out, the redeemer's remaining collateral gets queued for a future epoch via `_queueWithdrawal`. The redeemer cannot arbitrage the depeg with queued IOUs because the arbitrage economics assume immediate settlement, so effective peg-defense capacity drops to zero once the buffer is dry.

Refilling the buffer during stress requires admin to coordinate executor unwinds. Each executor has its own close flow: HyperLend requires flash loans, swaps, and Aave V3

interactions; Morpho requires flash loan callbacks and multi-step repayment; DelphoStableExecutor requires sequential HyperCore actions across 2s small blocks and 60s large blocks. A full unwind of a meaningful fraction of deployed capital takes tens of minutes to hours under ideal conditions and depends entirely on bot availability. During an ongoing depeg event where peg needs to be restored, this is too slow.

Impact

The protocol's primary peg-defense mechanism, redemption arbitrage, fails to respond to stress at the scale it needs to.

Code Snippet

https://github.com/sherlock-audit/2026-04-delpho-update-apr-8th-2026/blob/main/delpho_contracts/src/core/DelphoVault.sol#L459-L469 https://github.com/sherlock-audit/2026-04-delpho-update-apr-8th-2026/blob/main/delpho_contracts/src/core/DelphoVault.sol#L489-L506 https://github.com/sherlock-audit/2026-04-delpho-update-apr-8th-2026/blob/main/delpho_contracts/src/periphery/DelphoRouter.sol#L315-L343

Tool Used

Manual Review

Recommendation

Raise the default `bufferBps` materially (e.g., from 500 to 1500-2000, i.e., 15-20%) to give the buffer enough headroom to absorb typical redemption stress without immediately falling through to epoch queueing. The trade-off is lower executor yield on the reserved capital, but this is the cost of having a functional peg-defense mechanism during the scenarios where it actually matters.

Pair the larger buffer with active off-chain monitoring: alert the admin when `globalBufferFunds` drops below a threshold (e.g., 50% of target) and begin pre-emptive executor unwinds before the buffer is fully depleted. During observed peg stress (USDV price deviation beyond some threshold), the admin should halt further executor deployments via `reallocateProtocolFunds` and redirect incoming deposits entirely to the buffer until the event passes.

Discussion

Oxklapouchy

Issue addressed by increasing the default `bufferBps` to 10% in [PR#237](#)

Issue L-13: Stability Pool redemption seizes collateral greedily token-by-token while the fallback path seizes proportionally [RESOLVED]

Source:

<https://github.com/sherlock-audit/2026-04-delpho-update-apr-8th-2026/issues/94>

Summary

Delpho has two redemption paths that hit different functions for choosing which collateral tokens to take. The fallback borrower path uses `StableManagerLib.seizeFallbackCollateral`, which splits the seize across all tokens in the borrower's basket pro rata to each token's USD share. The Stability Pool path uses `StableManagerLib.selectCollateralToRedeem`, which sorts the basket descending by USD value and drains the highest-value token first, only moving to the next token when the first is exhausted. The two paths therefore treat multi-token baskets inconsistently: the fallback path spreads pressure across every token, the SP path concentrates it on one.

Vulnerability Detail

`_redeemFromPool` selects SP collateral via the greedy helper:

```
(address[] memory redeemTokens, uint256[] memory redeemAmounts,) =  
    StableManagerLib.selectCollateralToRedeem(vault.getUserCollateralDetails(_stabilityPool), netRedeem);
```

Inside `selectCollateralToRedeem`, the basket is sorted descending and consumed one token at a time:

```
collateralDetails.sort();  
...  
uint256 tokensNeeded = usdRemaining.mulDivUp(decimalScale,  
    collateralDetails.tokenPrices[i]);  
uint256 tokensTaken = tokensNeeded.min(collateralDetails.amounts[i]);
```

`seizeFallbackCollateral`, used for fallback borrower redemption, takes the opposite approach:

```
uint256 usdShare = coll.usdValues[i].mulDiv(params.targetUsd, coll.totalValue);  
uint256 amount = usdShare.mulDiv(decimalScale, coll.tokenPrices[i]);
```

Because `Vault.globalBufferFunds` is per-token (`tokenAccounting[token].globalBufferFunds`) and every `requestWithdrawal` drains only that token's reserve, the SP path concentrates buffer consumption on whichever token happens to rank highest by USD

value in the SP basket. Other tokens in the same basket keep their buffer untouched even when the top token's buffer is already empty and the redeemer is being queued.

Impact

Redemption against the SP can hit epoch queueing on one token while other tokens in the same SP basket still have full buffer capacity. In aggregate the Vault holds enough instant liquidity to settle the trade, but the greedy selector routes all demand at a single token's slice of it. Under peg stress this amplifies the effective shortfall because the usable buffer is not the sum of all token buffers but the buffer of whichever token the SP basket happens to list first by value.

There is also a composition effect: repeated SP redemptions systematically deplete the highest-value token in the SP basket, leaving the pool over time with a residual mix skewed toward lower-grade collateral. The fallback path does not have this drift because it draws pro rata.

Code Snippet

https://github.com/sherlock-audit/2026-04-delpho-update-apr-8th-2026/blob/main/delpho_contracts/src/core/DelphoStableManager.sol#L344-L348

https://github.com/sherlock-audit/2026-04-delpho-update-apr-8th-2026/blob/main/delpho_contracts/src/libs/StableManagerLib.sol#L140-L185

https://github.com/sherlock-audit/2026-04-delpho-update-apr-8th-2026/blob/main/delpho_contracts/src/libs/StableManagerLib.sol#L193-L236

Tool Used

Manual Review

Recommendation

Align the SP path with the fallback path. Replace the call to `selectCollateralToRedeem` in `_redeemFromPool` with a proportional selector modelled on `seizeFallbackCollateral`, so SP redemption also takes each token in the ratio `usdValues[i] / totalValue`. This keeps the two redemption paths symmetric, spreads buffer consumption across all token-level `globalBufferFunds` pools, and preserves the SP basket's composition over time.

Discussion

Oxklapouchy

Fix confirmed in [PR#239](#)

Issue L-14: Disabling a token traps its executor-deployed principal inside the executor and blocks admin from rebalancing it back to users [RESOLVED]

Source:

<https://github.com/sherlock-audit/2026-04-delpho-update-apr-8th-2026/issues/95>

Summary

When admin flips `isTokenSupported` to false for a collateral token, two Vault paths that exist precisely to move executor-deployed capital back toward user withdrawals begin to revert. `reportReturns` rejects disabled tokens, so any executor still holding deployed principal in that token cannot return it. `reallocateProtocolFunds` rejects disabled tokens in every mode, so even the unused slice that already sits in `fundsForExecutor` cannot be moved into `globalBufferFunds` or `availableForWithdrawRequest` to serve withdrawal requests. Users holding the disabled token as collateral keep their LTV intact (oracle still prices disabled tokens) and can continue minting USDV against it, but they can only withdraw up to the 5% buffer slice that happened to exist at the moment of disabling. The remaining 95% is permanently locked behind an admin-only path that the same admin action just disabled.

Vulnerability Detail

`reportReturns` is the only path by which an executor transfers deployed principal back to the Vault:

```
if (!configProvider.isExecutor(msg.sender)) revert DV_UNAUTHORIZED();
if (!configProvider.isTokenSupported(token)) revert DV_UNSUPPORTED_TOKEN();

TokenState storage ts = tokenAccounting[token];
if (ts.deployedByExecutor < principal) revert DV_INSUFFICIENT_DEPLOYED_FUNDS();
ts.deployedByExecutor -= principal;
IERC20(token).safeTransferFrom(msg.sender, address(this), principal);
ts.fundsForExecutor += principal;
```

Once the token is disabled, the executor's own unwind bot cannot complete even if it has already closed the underlying HyperLend, Morpho, or HyperCore positions and holds the principal. The Vault refuses to accept the tokens back.

`reallocateProtocolFunds` is the admin's only lever to move liquidity between the buffer, the executor pool, and the withdrawal-request pot:

```
if (!configProvider.isTokenSupported(token)) revert DV_UNSUPPORTED_TOKEN();

if (mode == ReallocationMode.BufferToExecutor) { ... }
```

```
else if (mode == ReallocationMode.ExecutorToBuffer) { ... }  
else { /* ExecutorToWithdrawRequest */ ... }
```

The disable gate is mode-agnostic. BufferToExecutor is the only mode that grows new exposure on the disabled token and therefore legitimately belongs behind the gate. ExecutorToBuffer and ExecutorToWithdrawRequest both move capital in the user-friendly direction and reduce exposure. They are blocked regardless.

Impact

Users holding the disabled token can only withdraw the slice that was already in `globalBufferFunds` at the moment of disabling (5% by default). Executors cannot return deployed principal via `reportReturns`, and admin cannot move `fundsForExecutor` back into the buffer via `reallocateProtocolFunds(ExecutorToBuffer)`.

Code Snippet

https://github.com/sherlock-audit/2026-04-delpho-update-apr-8th-2026/blob/main/delpho_contracts/src/core/DelphoVault.sol#L280

https://github.com/sherlock-audit/2026-04-delpho-update-apr-8th-2026/blob/main/delpho_contracts/src/core/DelphoVault.sol#L299

Tool Used

Manual Review

Recommendation

Drop the `isTokenSupported` check from `reportReturns`. In `reallocateProtocolFunds`, move the check inside the `BufferToExecutor` branch only, so the unwind modes (`ExecutorToBuffer`, `ExecutorToWithdrawRequest`) remain usable for disabled tokens.

Discussion

Oxklapouchy

Fix confirmed in [PR#219](#)

Issue L-15: Buffer can be drained cheaply via a deposit-then-withdraw cycle [RESOLVED]

Source:

<https://github.com/sherlock-audit/2026-04-delpho-update-apr-8th-2026/issues/97>

Summary

`collateralDeposit` adds to `globalBufferFunds[token]` and `requestWithdrawal` drains it, with no temporal separation between the two. Any caller can deposit and immediately withdraw, pulling the entire buffer at a cost equal only to the platform fee on the deposit. The planned `withdrawFeeBps = 0` adds no friction on the way out.

Vulnerability Detail

A deposit of X allocates $\text{bufferBps} \times (1 - \text{platformFeeBps}) \times X$ to the buffer. A same-block `requestWithdrawal` drains up to the full buffer via `_drainBuffer`. At default config (`platformFeeBps = 0.1\%`, `bufferBps = 5\%`, `withdrawFeeBps = 0\%`) the attacker spends roughly \$1 in deposit fee per \$50 of pre-existing buffer drained. If admin ever sets `platformFeeBps = 0`, the attack becomes free.

Impact

Buffer can be held near zero by a low-cost grief, disabling instant withdrawals for legitimate users and breaking redemption arbitrage before any actual depeg event arrives.

Code Snippet

https://github.com/sherlock-audit/2026-04-delpho-update-apr-8th-2026/blob/main/delpho_contracts/src/core/DelphoVault.sol#L460-L469

https://github.com/sherlock-audit/2026-04-delpho-update-apr-8th-2026/blob/main/delpho_contracts/src/core/DelphoVault.sol#L489-L506

Tool Used

Manual Review

Recommendation

Two main options:

- Implement a delay between deposit and withdrawal.
- Accept the risk and acknowledge the issue.

Discussion

Oxklapouchy

Fix confirmed in [PR#242](#)

Issue L-16: DelphoStabilityPool cannot stake USDV already held in its own balance [RESOLVED]

Source:

<https://github.com/sherlock-audit/2026-04-delpho-update-apr-8th-2026/issues/98>

Summary

`DelphoStabilityPool.stakeStable` requires USDV to be transferred from `msg.sender` (admin EOA) via `safeTransferFrom`, even when the SP already holds USDV in its own balance from `mintStable`, `unstakeStable`, or `deposit(mint=true, stake=false)`. Idle USDV cannot be re-staked without first calling `withdrawToken` to pull it out to the admin and then `stakeStable` to push it back in.

Vulnerability Detail

`mintStable(usdvAmount)`, `unstakeStable(amount)`, and `deposit(..., mint=true, stake=false, mintAmount)` all leave USDV in the SP's balance. `stakeStable` then runs `delphoStable.safeTransferFrom(msg.sender, address(this), amount)`, which targets the admin's wallet rather than the SP's own balance, so re-staking idle USDV requires a 3-step round-trip (withdraw → approve → stake).

Impact

Operational inefficiency.

Code Snippet

https://github.com/sherlock-audit/2026-04-delpho-update-apr-8th-2026/blob/main/delpho_contracts/src/core/DelphoStabilityPool.sol#L102-L111

Tool Used

Manual Review

Recommendation

Add a `stakeStableFromBalance(uint256 amount)` helper that approves and forwards to `Router.stakeUSDV` without `safeTransferFrom`, or change `stakeStable` to use the SP's own balance when sufficient.

Discussion

0xklapouchy

Fix confirmed in [PR#218](#)

Issue L-17: `setEpochDuration` Does Not Advance the Epoch Before Changing Duration [RESOLVED]

Source:

<https://github.com/sherlock-audit/2026-04-delpho-update-apr-8th-2026/issues/99>

Summary

When an admin calls `setEpochDuration`, the new duration takes effect immediately without first advancing or finalizing the current epoch. This leaves the vault in an inconsistent epoch state where the timing of the active epoch no longer aligns with user expectations or previously submitted withdrawal requests.

Vulnerability Details

`setEpochDuration` updates `epochDuration` in-place with no synchronization against the vault's current epoch state:

```
epochDuration = _epochDuration;
```

The vault tracks epochs using a start timestamp and the configured `epochDuration` to compute epoch boundaries. When the duration changes mid-epoch, all downstream calculations that derive the current epoch end and consequently the withdrawal scheduling are immediately invalidated. The current epoch's effective end time silently shifts without any on-chain event that users or the withdrawal queue can react to.

Impact

- Withdrawal requests submitted by users under the old epoch timing assumptions may be processed in the wrong epoch, either delaying fund access or releasing them earlier than the protocol intends.
- Loss of predictability for users and integrators relying on epoch timing for their withdrawal planning.

Recommendation

Before assigning the new `epochDuration`, `setEpochDuration` should invoke the vault's `autoAdvance` (or equivalent epoch-finalization logic) to close out the current epoch under the old duration. Only after the epoch is cleanly advanced should the new duration be stored. This ensures epoch boundaries remain consistent and no withdrawal requests are stranded in a mis-timed epoch.

Discussion

0xklapouchy

Fix confirmed in [PR#238](#)

Issue L-18: proportionBps Rounding to Zero Allows SP to Absorb Debt With No Liquidation Penalty [RE-SOLVED]

Source:

<https://github.com/sherlock-audit/2026-04-delpho-update-apr-8th-2026/issues/103>

Summary

In `_processNormalLiquidation`, `proportionBps` is computed as `spUsdvAbsorbed * 1e4 / totalDebt`. When `spUsdvAbsorbed` is small relative to `totalDebt`, this rounds down to zero. `stabilityPenalty` and `stabilityLiquidatorFee` are then computed as zero, meaning the SP absorbs debt and seizes collateral at a 1:1 ratio with no liquidation bonus, undermining the liquidation incentive design.

Vulnerability Details

```
uint256 proportionBps = _liqData.spUsdvAbsorbed.mulDiv(1e4, _liqData.totalDebt);
```

If `spUsdvAbsorbed = 5` and `totalDebt = 100000`, `proportionBps = 0`. Then:

```
stabilityPenalty = _liqData.liquidationPenalty.mulDiv(0, MAX_BPS) = 0;  
stabilityLiquidatorFee = _liqData.liquidatorFee.mulDiv(0, MAX_BPS) = 0;
```

`selectCollateralToRedeem` is called with `spUsdvAbsorbed + 0 = spUsdvAbsorbed`. The SP absorbs debt and takes proportional collateral with no penalty premium. The liquidation penalty that is supposed to reward absorbers is silently dropped.

Attack Path

1. SP has absorbed a tiny fraction of a very large debt position.
2. `proportionBps` rounds to zero due to integer division.
3. `stabilityPenalty = 0`, `stabilityLiquidatorFee = 0`.
4. SP absorbs debt without receiving the configured liquidation bonus.
5. The absorbed penalty is effectively lost, reducing incentives for SP participants.

Impact

Liquidation incentives are silently bypassed for small SP absorption fractions.

Recommendation

Use a minimum precision floor for `proportionBps` or compute penalties using the absorbed absolute amount directly rather than scaling through a BPS ratio, to avoid losing value from integer rounding.

Discussion

0xklapouchy

Fix confirmed in [PR#240](#)

Issue L-19: Proportional Interest Reduction May Leave Non-Zero Interest on a Fully Seized Position [RESOLVED]

Source:

<https://github.com/sherlock-audit/2026-04-delpho-update-apr-8th-2026/issues/107>

Summary

In `_processOneFallbackBorrower`, the proportional reduction formula can leave a small residual `pendingInterest` due to rounding while clearing whole `usdvMinted` amount.

Vulnerability Details

```
uint256 interestReduction = borrowerPos.pendingInterest.mulDiv(usdApplied,
↳ totalDebt);
uint256 principalReduction = usdApplied - interestReduction;
```

`mulDiv` rounds down, so `interestReduction` may be slightly less than the proportional share. A dust amount of `pendingInterest` remains on a position where all the debt cleared.

Impact

Dust interest entries remain open and it will create new user position in the next operation because principal is completely removed.

Recommendation

Use `mulDivUp()` for this particular rounding.

Discussion

Oxklapouchy

Fix confirmed in [PR#235](#)

Issue L-20: Shortfall Calculation Records Interest Portion of Absorbed Debt as Bad Debt [RESOLVED]

Source:

<https://github.com/sherlock-audit/2026-04-delpho-update-apr-8th-2026/issues/108>

Summary

In `_processNormalLiquidation`, the shortfall is computed from `spUsdvAbsorbed`, which is derived from `totalDebt = usdvMinted + pendingInterest`. When collateral is insufficient to cover the full absorbed amount, the entire shortfall including the interest component is recorded as bad debt. However, `pendingInterest` is protocol fee revenue, not principal that backs USDV in circulation. Forgoing it in a distressed scenario does not affect USDV solvency, so recording it as bad debt overstates the protocol's true solvency deficit.

Vulnerability Details

```
uint256 shortfall = (_liqData.spUsdvAbsorbed + stabilityPenalty) - actualSeized;
if (shortfall != 0) _recordBadDebt(shortfall);
```

`spUsdvAbsorbed` is capped at `totalDebt.min(spUsdvStaked)`, and `totalDebt` is the sum of the borrower's `usdvMinted` (principal, which directly backs outstanding USDV) and `pendingInterest` (accrued fee revenue owed to the protocol). When seized collateral falls short of `spUsdvAbsorbed`, the recorded shortfall includes whatever fraction of `pendingInterest` went unmet. Since `pendingInterest` is a protocol revenue claim, not a liability that leaves USDV unbacked, its shortfall has no impact on USDV collateralization. Only the unmet `usdvMinted` portion represents a genuine bad debt threat.

Impact

`totalBadDebt` is overstated in every collateral-shortfall liquidation where the absorbed debt includes accrued interest. Governance, risk monitoring, and any circuit-breakers keyed on `totalBadDebt` will react to an inflated figure, potentially triggering protective actions that are not warranted by the protocol's true solvency state.

Recommendation

Before recording the shortfall, separate the interest component from `spUsdvAbsorbed`. Only the portion of the shortfall attributable to unmet principal (`usdvMinted`) should be passed to `_recordBadDebt`. The interest shortfall can be silently forgone or emitted as a distinct event representing foregone protocol revenue.

Discussion

0xklapouchy

Fix confirmed in [PR#241](#)

Issue L-21: Stability Pool Receives New Debt During Liquidation Without LTV Validation [ACKNOWLEDGED]

Source:

<https://github.com/sherlock-audit/2026-04-delpho-update-apr-8th-2026/issues/109>

This issue has been acknowledged by the team but won't be fixed at this time.

Summary

In `_processNormalLiquidation`, the SP's `usdvMinted` is increased by `spUsdvToMint` to represent its share of the liquidated debt. This is done unconditionally without checking whether the resulting debt-to-collateral ratio on the SP's position exceeds the liquidation threshold.

Vulnerability Details

```
stabilityPoolPosition.usdvMinted += spUsdvToMint;
```

The SP accumulates debt each time it participates in a liquidation. Because no LTV check is performed on the SP's position after debt is added, the SP's own borrow ratio can exceed the liquidation threshold. When `spLiquidationDisabled = false`, this makes the SP itself eligible for liquidation. A large liquidation event or a sequence of smaller ones can push the SP's accumulated debt past its collateral value without any warning or prevention.

Impact

The SP, which is intended to be the first-line absorber of liquidated debt, can itself become undercollateralized through the normal operation of the liquidation mechanism. If SP liquidation is enabled, this could trigger a cascading redistribution of the SP's own position across all active borrowers, precisely the high-impact systemic event the `setSpLiquidationDisabled` flag is designed to prevent.

Recommendation

After incrementing `stabilityPoolPosition.usdvMinted`, validate that the SP's resulting LTV does not exceed the liquidation threshold. If it does, either cap the absorption or emit a warning event so the protocol can respond before the SP becomes liquidatable.

Discussion

0xklapouchy

Issue L-22: Redundant Exclusion Checks in `_distributeFallbackDebt` Loop [RESOLVED]

Source:

<https://github.com/sherlock-audit/2026-04-delpho-update-apr-8th-2026/issues/110>

Summary

In `_distributeFallbackDebt`, the loop explicitly excludes `_user` (the liquidated borrower) and `_stabilityPool` from receiving fallback debt. Both exclusions are unnecessary: the liquidated user is removed from `DelphoSortedDebts` before `_distributeFallbackDebt` is called, and the Stability Pool is never inserted into the sorted list.

Vulnerability Details

```
if (current != _user && current != _stabilityPool) {
```

The liquidated user is removed via `IDelphoSortedDebts(delphoContracts.delphoSortedDebts).remove(user)` in `_liquidateUser` before the fallback path is entered. The Stability Pool is explicitly skipped at insertion time in `DelphoSortedDebts.insert`. Neither address should appear in the list during this traversal.

Impact

No functional impact. The checks are dead code that add minor gas overhead per iteration. However, they create misleading documentation suggesting these addresses might be present in the sorted list, which could confuse future maintainers.

Recommendation

Remove the redundant checks and update the NatSpec comment to clarify that the sorted list structurally guarantees neither address will be present at this point.

Discussion

Oxklapouchy

Fix confirmed in [PR#247](#)

Issue L-23: No Standalone Collateral Withdrawal Function for Morpho Positions [ACKNOWLEDGED]

Source:

<https://github.com/sherlock-audit/2026-04-delpho-update-apr-8th-2026/issues/112>

This issue has been acknowledged by the team but won't be fixed at this time.

Summary

The executor has no dedicated function to withdraw collateral from Morpho without going through the full close or partial-close flow. Every collateral withdrawal is bundled inside `_handleClose`, which is the flash loan callback and always involves repaying debt and swapping tokens. Operators have no way to withdraw free collateral independently, limiting operational flexibility and forcing unnecessary flash loan costs in scenarios where a simple withdrawal would suffice.

Vulnerability Details

`_handleClose` is called exclusively from `onMorphoFlashLoan` in response to a flash loan initiated by `_closeLeveragePosition`. Within it:

```
morpho.withdrawCollateral(mp, withdrawAmt, address(this), address(this));
```

This withdrawal is coupled to debt repayment and a collateral-to-USDT swap. There is no entry point in the contract that calls `morpho.withdrawCollateral` directly without these preconditions. Situations where a withdrawal is legitimately needed without a swap include: collateral rebalancing after the debt has already been fully repaid via `repayAsset`, retrieval of excess collateral after a position is partially unwound, or recovery of collateral when no debt exists.

The existing `_repayAndWithdraw` internal helper does handle the no-flash-loan case when `flashloanAmount == 0`, but it is only reachable through `_closeLeveragePosition` and still requires going through the close flow with its full parameter set.

Attack Path

This is not an exploitable attack vector. The risk is operational: the absence of a standalone withdrawal path means operators are forced to construct a close or partial-close call with a flash loan and swap parameters even when they only need to retrieve free collateral. This creates unnecessary flash loan fee costs and introduces swap slippage exposure in situations where no swap is needed.

Impact

Protocol operators cannot withdraw collateral from Morpho without executing a full close flow. In distressed or time-sensitive scenarios (e.g., a rapid rebalance or emergency collateral retrieval after debt repayment), the lack of a direct withdrawal path increases operational complexity and cost.

Recommendation

Add a dedicated function restricted to the executor role that calls `morpho.withdrawCollateral` directly with a specified amount, without requiring a flash loan or swap. This mirrors the existing `borrowAsset` and `repayAsset` standalone functions and provides a complete and symmetric set of position management primitives.

Discussion

JayusJay

This is valid but appears to be a duplicate of <https://github.com/sherlock-audit/2026-04-delpho-update-apr-8th-2026/issues/83>

Oxklapouchy

#112 is a sibling finding to #83

Fix confirmed in [PR#253](#)

Issue L-24: redeem Has No Slippage Protection for Minimum USDV Burned or Minimum Collateral Received [ACKNOWLEDGED]

Source:

<https://github.com/sherlock-audit/2026-04-delpho-update-apr-8th-2026/issues/113>

This issue has been acknowledged by the team but won't be fixed at this time.

Summary

`redeem` accepts a `usdvAmount` but provides no parameters for the caller to specify a minimum amount of collateral they expect to receive or a maximum amount of USDV they are willing to burn. The actual collateral output depends on oracle prices and borrower position states at execution time, both of which can change between transaction submission and inclusion. A user can submit a redemption expecting a certain collateral value and receive significantly less with no protection.

Vulnerability Details

The `redeem` function passes `usdvAmount` directly to `stableManager.redeem` and auto-requests withdrawal for all newly credited collateral:

```
uint256 fallbackRedeemed = stableManager.redeem(msg.sender, usdvAmount,
↪ normalRedeemAmount);
```

The collateral received is determined inside `StableManager` based on:

- Live oracle prices at the time of execution for each collateral token seized.
- How much of the redemption goes to the Stability Pool versus the fallback borrower path.
- The partial absorption behavior in the fallback path, which can result in less collateral than expected if borrower positions are partially depleted.

None of these are predictable by the caller at submission time. There is no `minCollateralOut` parameter to revert if the received collateral value falls below an acceptable threshold, and no `maxUsdvBurned` to limit how much USDV is consumed if the fallback path delivers less collateral per USDV burned than expected. A sandwiched or delayed transaction can result in the user burning their full `usdvAmount` while receiving collateral at an unfavorable exchange rate.

Attack Path

1. User submits `redeem(1000e6)` expecting to receive approximately 1000 USD worth of collateral at current prices.
2. Between submission and inclusion, oracle prices for the collateral assets drop or a large redemption changes borrower position states.
3. The user burns 1000 USDV and receives collateral worth materially less than 1000 USD with no on-chain protection to abort.

Impact

Users have no on-chain recourse against price movement or sequencing between transaction submission and execution. In a volatile market or a high-activity redemption period, slippage in the collateral received per USDV burned can be significant. This is especially relevant for the fallback redemption path where collateral is seized proportionally from individual borrowers based on live prices.

Recommendation

Add `minCollateralUsd` and/or `minCollateralAmounts` parameters to `redeem` so the function reverts if the collateral credited falls below the caller's acceptable threshold. Alternatively, expose the expected output via a view function so callers can compute and enforce slippage off-chain before calling. This follows the standard pattern used by AMM routers.

Discussion

Oxklapouchy

Acknowledged

Issue L-25: redeem Auto-Requests Withdrawal Without Giving User an Option to Keep Collateral in the Vault [ACKNOWLEDGED]

Source:

<https://github.com/sherlock-audit/2026-04-delpho-update-apr-8th-2026/issues/114>

This issue has been acknowledged by the team but won't be fixed at this time.

Summary

After a successful redemption, `_handleRemainingCollateralAfterRedemption` automatically converts all newly credited collateral into a withdrawal request on behalf of the user. This removes the user's choice to keep the collateral inside the vault as a deposited balance, which they may want to do to maintain their LTV, open a new borrow, or use the collateral for another purpose without incurring an extra withdrawal epoch delay.

Vulnerability Details

```
_handleRemainingCollateralAfterRedemption(vault, allTokens, collateralBefore);
```

This function unconditionally calls `vault.requestWithdrawal` for every token whose balance increased during the redemption. The user is given no flag or parameter to opt out of the automatic withdrawal request. A user who redeems collateral and wants to immediately use that collateral as backing for a new borrow position, or who simply wants to hold it in the vault to improve their collateral ratio, is forced through the withdrawal queue with its associated waiting period.

Since the withdrawal request queues the collateral for a future epoch, the collateral is locked and unavailable for vault-level operations until the epoch completes and `claimWithdrawal` is called. A user who would have preferred to leave it deposited now faces an unnecessary epoch delay to re-deposit it.

Impact

Redeemers cannot choose to retain newly credited collateral inside the vault. This limits composability and forces an unnecessary epoch-delay roundtrip for users who want to cycle back into a vault position after a redemption. It also means every redemption unconditionally generates a withdrawal request entry, adding queue state even when the user has no intention of withdrawing.

Recommendation

Add a boolean parameter (e.g. `autoWithdraw`) to `redeem`. When `false`, skip the `_handleRemainingCollateralAfterRedemption` call and leave the credited collateral in the user's vault balance. When `true`, preserve the current behavior. This gives users full control over what happens to their newly received collateral post-redemption.

Discussion

Oxklapouchy

Acknowledged

Issue L-26: Default oracle staleness age is too low for Chainlink push feeds [RESOLVED]

Source:

<https://github.com/sherlock-audit/2026-04-delpho-update-apr-8th-2026/issues/119>

Vulnerability Details

defaultStalenessAge is initialized to 600 seconds (5 minutes):

```
defaultStalenessAge = 600; // 5 minutes
```

This value was reasonable for the previous Pyth pull-oracle, but Chainlink push feeds update only on their heartbeat/deviation schedule, and many feeds have heartbeats of 1 to 24 hours. Any token that relies on this default (no per-token `tokenStalenessAge` override) will have its price read revert `DO_STALE_PRICE` in normal markets, because the feed simply will not have updated within 5 minutes during calm periods. Such a token is effectively bricked (no deposits, borrows, withdrawals, or liquidations) from launch. The per-token override mitigates this only when an operator remembers to set it for every token.

The same tightness carries into the production deploy. The dev deploy script was updated to set every token's per-token staleness to 1 day (`setTokenStalenessAge(token, 86400)`) for `KHYPE`, `WHYPE`, `USDT`, `USDC`), but the prod deploy script was not updated in the same way and still uses much tighter values: `KHYPE 600`, `WHYPE 600`, `USDT 3600`, `USDC 3600`. If any of these Chainlink feeds update less frequently than the configured age (likely for the `HYPE/KHYPE` feeds at 600s), the corresponding collateral is bricked from launch on production while the dev environment is not, so the problem will not surface in dev testing.

Code Snippet

https://github.com/Delpho-labs/delpho_contracts/blob/fac75aeeb8c589745ac6d7d620beee464432821a/src/configurators/base/DelphoOracleConfig.sol#L17

https://github.com/Delpho-labs/delpho_contracts/blob/fac75aeeb8c589745ac6d7d620beee464432821a/script/prod/02_DeploymentScript.s.sol#L119

https://github.com/Delpho-labs/delpho_contracts/blob/fac75aeeb8c589745ac6d7d620beee464432821a/script/dev/02_DeploymentScript.s.sol#L131

Recommendation

Set the default staleness to a value consistent with realistic Chainlink heartbeats, or require a per-token `tokenStalenessAge` to be provided when a token is registered.

Discussion

0xklapouchy

Fix confirmed in [PR#271](#)

Issue L-27: Incorrect storage-gap comments and reserve totals in oracle/token config modules [RESOLVED]

Source:

<https://github.com/sherlock-audit/2026-04-delpho-update-apr-8th-2026/issues/120>

Vulnerability Details

The storage-gap accounting comments in two config modules are wrong, and the reserved totals no longer sum to the intended 50 slots.

DelphoTokenConfig has two own storage slots (`_tokenConfigs` mapping and `supportedTokens` array) but the comment claims three, and $2 + 47 = 49$, not 50:

```
/// @dev Reserved storage gap for future upgrades (3 own slots + 47 gap = 50 total).
uint256[47] private __gap;
```

DelphoOracleConfig has two own storage slots (`defaultStalenessAge` and `tokenStalenessAge`) but the comment claims one, and $2 + 49 = 51$, not 50 (the `pyth` slot was removed in the Chainlink migration without adjusting the gap):

```
/// @dev Reserved storage gap for future upgrades (1 own slot + 49 gap = 50 total).
uint256[49] private __gap;
```

These are inaccurate today and break the per-module 50-slot reserve convention, which matters for predictable storage layout across the inherited config modules in `DelphoConfigProvider`.

Code Snippet

https://github.com/Delpho-labs/delpho_contracts/blob/fac75aeeb8c589745ac6d7d620beee464432821a/src/configurators/base/DelphoTokenConfig.sol#L18

https://github.com/Delpho-labs/delpho_contracts/blob/fac75aeeb8c589745ac6d7d620beee464432821a/src/configurators/base/DelphoOracleConfig.sol#L14

Recommendation

The correct fix depends on whether the proxy has already been deployed with the pre-migration (Pyth) storage layout:

- **Not yet deployed with the old layout:** simply correct the comments to reflect the actual own-slot counts and size each `__gap` so the module reserves exactly 50 slots (`__gap[48]` in `DelphoTokenConfig`, `__gap[48]` in `DelphoOracleConfig`).

- **Already deployed (on-chain layout matches the old comments):** do NOT shift slots, or an upgrade will reinterpret old storage as new variables (for example the old `pyth` address would be read back as `defaultStalenessAge`). Instead, reintroduce each removed slot as a placeholder that retains its original position and type, prefixed `__deprecated_` (e.g. `address private __deprecated_pyth;` in `DelphoOracleConfig`), declare the new variables after the preserved slots, and shrink the `__gap` so the per-module total stays at the same 50 slots that are already on-chain. This keeps the live storage layout intact across the upgrade.

If the proxies are already live, the same caution applies to the `TokenConfig` struct change (fields removed/added inside `_tokenConfigs`): re-add removed struct fields as deprecated placeholders to preserve the struct's field ordering, or migrate existing entries, otherwise stored token configs will be read back corrupted.

Discussion

Oxklapouchy

Fix confirmed in [PR#270](#)

Issue L-28: New errors missing NatSpec and an unused error declaration [RESOLVED]

Source:

<https://github.com/sherlock-audit/2026-04-delpho-update-apr-8th-2026/issues/121>

Vulnerability Details

The errors added for the Chainlink migration have no NatSpec documentation, while the pre-existing errors in the same interface do, and one of the new errors is declared but never used anywhere in the oracle:

```
error DO_ZERO_OR_NEGATIVE_PRICE();
error DO_UNINITIALIZED_ROUND();
error DO_STALE_ROUND();          // declared but never used
error DO_STALE_PRICE();
error DO_PRICE_OUT_OF_BOUNDS();
```

DO_STALE_ROUND appears to be the vestige of an answeredInRound-based staleness check that was (correctly) not implemented, since answeredInRound is deprecated for OCR feeds.

Code Snippet

https://github.com/Delpho-labs/delpho_contracts/blob/fac75aeeb8c589745ac6d7d620beee464432821a/src/core/interfaces/IDelphoOracle.sol#L41

Recommendation

Add NatSpec to the new errors to match the documentation style of the existing ones, and remove the unused DO_STALE_ROUND declaration.

Discussion

Oxklapouchy

Fix confirmed in [PR#275](#)

Disclaimers

Sherlock does not provide guarantees nor warranties relating to the security of the project.

Usage of all smart contract software is at the respective users' sole risk and is the users' responsibility.