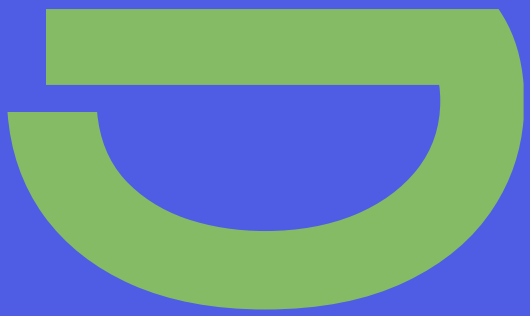
OpenZeppelin | security

DollarStore V3 Audit



August 5, 2026

Table of Contents

Summary

Type	DeFi
Timeline	From 2026-07-21 To 2026-07-24
Languages	Solidity

Total Issues	8 (5 resolved)
Critical Severity Issues	0 (0 resolved)
High Severity Issues	0 (0 resolved)
Medium Severity Issues	2 (1 resolved)
Low Severity Issues	2 (1 resolved)
Notes & Additional Information	4 (3 resolved)
Client Reported Issues	0 (0 resolved)

Scope

OpenZeppelin performed an audit of the `coasify/dollar` repository at the `d4cea9a` commit.

In scope were the following files:

```
contracts
└── src
    ├── DLRS.sol
    └── DollarStore.sol
```

System Overview

DollarStore v3 is an upgradeable (UUPS) exchange for a curated set of stablecoins. Users deposit any listed stablecoin and receive DLRS, a receipt token that represents a 1:1 claim on the protocol's combined reserves which they later redeem DLRS for any listed asset, or swap directly between listed assets at a flat 1:1 rate. All listed assets are treated as interchangeable dollars, so deposits, redemptions, and swaps are priced at par, with no spread or slippage curve.

The core user actions are `deposit`, `withdraw`, `swap`, and `swapExactInput`. `deposit` pulls a listed asset into the protocol, credits its reserve in normalized 6-decimal units, and mints an equal amount of DLRS; `withdraw` burns DLRS and returns the chosen asset in its native units. `swap` fills as much as it can immediately and queues any unfilled remainder, while `swapExactInput` is all-or-nothing and reverts unless the full amount can be filled at once. DLRS itself is a non-transferable (soulbound) 6-decimal token that can only be minted or burned by the `DollarStore` contract, so it serves purely as a redeemable claim rather than a tradable asset.

Liquidity is matched through directed FIFO queues backed by protocol reserves. Each ordered asset pair has its own queue: an incoming swap fills first against the opposite-direction queue, then from protocol reserves provided no positions are already waiting in the same direction, and escrows any remainder as a new position at the back of the pair's queue. `processQueue` later settles queued positions in first-in-first-out order from the available reserves of the wanted asset, moving each escrowed offer into reserves as it fills. A minimum order size and a per-queue position cap bound queue growth.

Security Model and Trust Assumptions

- The contract is not expected to integrate with rebasing and fee-on-transfer tokens.
- Price feeds are assumed to be live, accurate Chainlink-style aggregators that report at most 18 decimals for the lifetime of a listing. `addHubAsset` and `setPriceFeed`

validate that decimals are `18` or less only at configuration time, while `_checkPeg` recomputes `10 ** (18 - decimals())` at runtime with no re-validation. A feed that later reports more than 18 decimals (for example a proxy or migrated feed) panic-reverts and halts inflows that route through `_checkInflow`.

- Reserve assets are assumed to remain freely transferable. An issuer-level blacklist, freeze, or seizure of the protocol's holdings is outside protocol control; the queue-fill and cancel paths degrade gracefully around a failed transfer via `_tryTransfer`, but a frozen reserve asset still blocks the corresponding withdrawals and swaps and cannot be recovered by the protocol.

- DLRS is soulbound by design: it is a non-transferable, 6-decimal receipt representing a 1:1 claim on the hub reserve basket, mintable and burnable only by the `DollarStore` contract. Value is intended to move only through deposit, withdraw, and swap, never through direct token transfers or secondary markets.

Privileged Roles

- **Governor** controls the asset registry, oracle and peg risk parameters, and reserve accounting; also rotates the Governor and Guardian roles.
- **Guardian** holds emergency authority to halt activity globally, per asset, or per pool, to cancel queue positions, and to tighten launch caps.
- **Upgrader** holds UUPS upgrade authority and rotates the Upgrader role.
- Only the `DollarStore` contract may mint or burn `DLRS`. Both `DLRS.mint` and `DLRS.burn` are gated by `onlyDollarStore`.

Medium Severity

M-01 Below-Minimum First Queue Position Can Disable Reserve-Based Fills for a Swap Route

Directed swaps are matched in `_fillDirected`, which fills against the exact-opposite queue and then from protocol reserves, but only when the same-direction queue is empty (`positionCount == 0`). The queue's escalating minimum order size is enforced in `_enqueueRemainder`, which exempts the first position of an empty queue from the `MIN_ORDER_BASE` check. The issue is that these two rules interact: when a route's reserves are zero and both queues are empty (for example at launch, for a newly listed asset, or after a reserve is fully drained), a `swap` submitted with `minAmountOut = 0` fills nothing and enqueues a dust remainder as the first position. Because that single position sets `positionCount` to a non-zero value, reserve-based instant fills for that direction are disabled and `getSwapQuote` returns `0`. As a result, once reserves refill, same-direction swaps can no longer be filled from reserves: larger swaps are forced into the queue, and swaps whose remainder is below the minimum order size revert with `OrderTooSmall`, degrading liveness for the route until the dust position is cleared by a permissionless `processQueue` call or a cancellation.

Consider decoupling the reserve-fill gate from the raw position count so that a dust first position cannot strand reserves while preserving the legitimate first-position exemption. Alternatively, consider automatically clearing below-minimum positions from reserves when they are available, or integrating a bounded `processQueue` step into the `swap` path.

Update: *Resolved at commit eb61b18 in pull request #22.*

M-02 Peg Check Only Applied To Inflows So Premium Assets Can Be Withdrawn At The Protocol's Expense

The protocol accounts for every listed hub and treats one unit of any asset as exactly one dollar regardless of its market price. The oracle price is consulted only through `_checkInflow`, which applies the `_checkPeg` tolerance band on the assets entering the

protocol: the deposited asset in `deposit`, and the `offerAsset` in `swap`, `swapExactInput`, and `processQueue`.

However, no equivalent check is applied to assets *leaving* the protocol. The `withdraw` function burns `DLRS` and transfers the caller's chosen asset at a fixed 1:1 rate without any peg validation, and `swap` and `swapExactInput` validate only the `offerAsset`, never the `wantAsset` delivered to the recipient. As a result, when a listed asset trades above the upper peg tolerance, an actor can deposit an in-band asset to mint `DLRS` at par, redeem the premium asset 1:1 through `withdraw`, and sell it externally for the premium, repeating until that asset's reserves are exhausted. The extracted premium is borne by the remaining `DLRS` holders, whose claim is rebased onto the lower-value basket assets. Because `withdraw` is not gated by `whenNotPaused` and neither `pause` nor `pausePool` affect it, there is no mechanism to halt the outflow while a premium persists.

Consider enforcing an upper-bound peg check on outflows, rejecting redemptions and `wantAsset` payouts of an asset trading above the tolerance band. Alternatively, consider applying a redemption fee or haircut on premium assets to remove the arbitrage without coupling the exit path to oracle liveness.

Update: *Acknowledged, not resolved. The team stated:*

> *M-02 - Peg check only on inflows. Acknowledged by design. Extracting a premium asset at par is ordinary arbitrage on a par-value dollar exchange: it is nominally harmless (the pool never pays out more than a dollar of backing) and peg-restoring. The genuinely harmful direction, a below-par asset entering and displacing good reserves, is already blocked by the inflow peg check plus the guardian deposit pause.*
>
> *Adding an outflow price check would only tie every withdrawal to a live oracle, which we deliberately avoid.*

Low Severity

L-01 Inconsistent `minAmountOut` Units Between `swap` and `swapExactInput`

While the protocol accounts internally in 6-decimal normalized units, `swapExactInput` compares `minAmountOut` against a native-unit `amountOut` yet reverts with `MinAmountNotMet(filled, minAmountOut)` where `filled` is normalized, and the

interface documents `minAmountOut` as normalized for `swap` but leaves it unspecified for `swapExactInput`. As a result, for an asset whose decimals differ from 6, an integrator following the `swap` convention can unintentionally weaken the minimum-output check by up to the scaling factor (e.g. `1e12` for an 18-decimal asset), though the all-or-nothing one-to-one fill means the output is deterministic and no value is at risk.

Consider standardizing `minAmountOut` and the `MinAmountNotMet` payload on the normalized-unit convention and documenting it in the `IDollarStore` NatSpec.

Update: *Resolved in pull request #18.*

L-02 Frozen Reserve Asset Can Create a Redemption Race

`DLRS` is a fungible claim on the whole hub basket, and `withdraw` redeems any single listed asset at a fixed 1:1 rate without a `whenNotPaused` gate. If a listed reserve asset can no longer be transferred out of the contract, for example after an issuer blacklists or pauses it, redemptions of that asset revert while every other asset can still be withdrawn 1:1 by whoever redeems first. `syncReserves` cannot write the frozen balance down, since it only lowers reserves when the actual `balanceOf` decreases, so the frozen asset keeps counting as full backing. Holders who exit early recover in full against the healthy assets, and the remaining holders are left with `DLRS` that can only be redeemed against the frozen asset, with no way for the guardian to stop the outflow. The queue paths use a `_tryTransfer` helper that tolerates failed transfers, but `withdraw` has no equivalent. The trigger is an external issuer action and the loss on the frozen asset is unavoidable, so the concern is that this loss falls entirely on the last holders instead of being shared across everyone.

Consider having each redemption draw proportionally across all listed assets, and letting the governor lower the value credited for an asset that can no longer be transferred, so the loss is shared across all holders instead of falling entirely on whoever withdraws last.

Update: *Acknowledged, not resolved. The team stated:*

> *Accepted as a low, not fixed. The protocol offers no reward or compensation to outside depositors, so we assume any depositor is fully comfortable with the risk associated with the issuer of the asset they are providing. The risk is documented and knowingly accepted, and we did not add a depositor whitelist, to keep the design simple.*

Notes & Additional Information

N-01 Floating Pragma

Pragma directives should be fixed to clearly identify the Solidity version with which the contracts will be compiled.

`DLRS.sol` and `DollarStore.sol` have the `solidity ^0.8.24` floating pragma directive.

Consider using fixed pragma directives.

Update: *Acknowledged, not resolved. The team stated:*

> *Not changed. The deployed bytecode is compiled with a single fixed compiler version configured in the build, so the on-chain artifact is deterministic regardless of the ^0.8.24 pragma; pinning it adds no security benefit here, so we did not consider it necessary.*

N-02 Unsafe ABI Encoding

Generating calldata with `abi.encodeWithSelector` is not type-safe, as it does not verify at compile time that the supplied arguments match the called function's parameters. Within `DollarStore.sol`, `abi.encodeWithSelector` is used to build the ERC-20 transfer calldata; the arguments are correctly typed here, so there is no present encoding error.

Consider replacing `abi.encodeWithSelector` with `abi.encodeCall`, which checks the supplied values against the types expected by the called function and avoids errors caused by typos.

Update: *Resolved in pull request #19.*

N-03 Magic Numbers in the Code

Throughout the codebase, there are occurrences of literal values with unexplained meanings.

- The `6` literal number in `DLRS.sol`.

- The `18` literal number in `DollarStore.sol` (1, 2, 3).
- The `50` literal number in `DollarStore.sol`.
- The `500` literal number in `DollarStore.sol`.
- The `3600` literal number in `DollarStore.sol`.
- The `1e18` literal number in `DollarStore.sol` (1, 2).
- The `10_000` literal number in `DollarStore.sol` (1, 2).

Consider defining and using `constant` variables instead of these literals to improve readability, noting that the `18` occurrences represent two distinct values (the maximum supported feed decimals and the fixed-point normalization target) and should not be collapsed into a single constant.

Update: *Resolved in pull request #20.*

N-04 Unused Named Returns in `getQueuePosition`

The `getQueuePosition` function declares named return variables (`owner`, `offerAsset`, `wantAsset`, `amount`, `timestamp`) but leaves them unassigned and returns an explicit tuple instead, leaving the named returns unused and the `return` statement redundant.

Consider removing either the named return variables or the explicit `return` statement so the function uses a single, consistent return convention.

Update: *Resolved in pull request #21.*

Conclusion

The audit covered the `DollarStore` and `DLRS` contracts of the DollarStore v3 rebuild which is an upgradeable, hub-and-spoke exchange for listed stablecoins in which users deposit into a shared reserve basket, receive the soulbound `DLRS` receipt token as a 1:1 claim, and redeem or swap between listed assets at a flat 1:1 rate. No critical- or high-severity issues were identified. Two medium-severity issues were reported, relating to the directed-queue fill logic and the peg check being enforced only on inflows, alongside two low- and four note-severity issues recommending further correctness and code-quality improvements. We thank the DollarStore team for their participation during this audit.

Appendix

Issue Classification

OpenZeppelin classifies smart contract vulnerabilities on a 5-level scale:

- Critical
- High
- Medium
- Low
- Note/Information

Critical Severity

This classification is applied when the issue's impact is catastrophic, threatening extensive damage to the client's reputation and/or causing severe financial loss to the client or users. The likelihood of exploitation can be high, warranting a swift response. Critical issues typically involve significant risks such as the permanent loss or locking of a large volume of users' sensitive assets or the failure of core system functionalities without viable mitigations. These issues demand immediate attention due to their potential to compromise system integrity or user trust significantly.

High Severity

These issues are characterized by the potential to substantially impact the client's reputation and/or result in considerable financial losses. The likelihood of exploitation is significant, warranting a swift response. Such issues might include temporary loss or locking of a significant number of users' sensitive assets or disruptions to critical system functionalities, albeit with potential, yet limited, mitigations available. The emphasis is on the significant but not always catastrophic effects on system operation or asset security, necessitating prompt and effective remediation.

Medium Severity

Issues classified as being of medium severity can lead to a noticeable negative impact on the client's reputation and/or moderate financial losses. Such issues, if left unattended, have a moderate likelihood of being exploited or may cause unwanted side effects in the system.

These issues are typically confined to a smaller subset of users' sensitive assets or might involve deviations from the specified system design that, while not directly financial in nature, compromise system integrity or user experience. The focus here is on issues that pose a real but contained risk, warranting timely attention to prevent escalation.

Low Severity

Low-severity issues are those that have a low impact on the client's operations and/or reputation. These issues may represent minor risks or inefficiencies to the client's specific business model. They are identified as areas for improvement that, while not urgent, could enhance the security and quality of the codebase if addressed.

Notes & Additional Information Severity

This category is reserved for issues that, despite having a minimal impact, are still important to resolve. Addressing these issues contributes to the overall security posture and code quality improvement but does not require immediate action. It reflects a commitment to maintaining high standards and continuous improvement, even in areas that do not pose immediate risks.