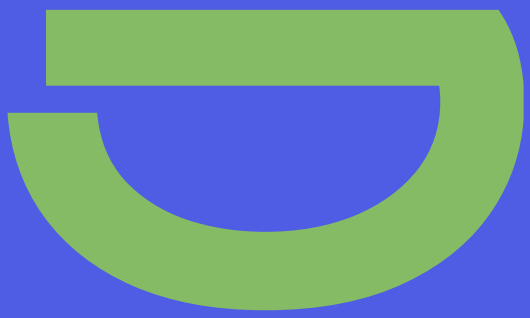
OpenZeppelin | security

DollarStore Audit



March 10, 2026

Table of Contents

Summary

Type	DeFi
Timeline	From 2026-02-16 To 2026-02-19
Languages	Solidity

Total Issues	13 (12 resolved, 1 partially resolved)
Critical Severity Issues	0 (0 resolved)
High Severity Issues	1 (1 resolved)
Medium Severity Issues	2 (1 resolved, 1 partially resolved)
Low Severity Issues	4 (4 resolved)
Notes & Additional Information	6 (6 resolved)

Scope

OpenZeppelin performed an audit of the wandering-soupsmith/dollar repository at commit 091852e.

In scope were the following files:

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

contracts/script
├── DeployMainnet.s.sol
└── Deploy.s.sol
```

System Overview

DollarStore is a zero-fee stablecoin aggregation and swapping protocol. Users deposit any supported stablecoin and receive DLRS, a soulbound ERC-20 receipt token, at a 1:1 ratio. DLRS can then be redeemed for any stablecoin available in the protocol's reserves, swapped directly between stablecoins, or used to enter a FIFO queue for a stablecoin that is not currently in reserve. Queue positions are filled automatically as new supply of the target stablecoin arrives through deposits or swaps. An aggregator-facing swap endpoint supports custom recipients and deadlines for routing integration.

`DollarStore`

This core contract manages reserves, the receipt token lifecycle, and the swapping and queueing infrastructure. Key functionalities include:

- Accepting stablecoin deposits and minting DLRS 1:1 to the depositor.
- Burning DLRS to redeem any supported stablecoin from reserves 1:1.
- Swapping between any two supported stablecoins with no fee, drawing from and replenishing reserves atomically.
- Maintaining a per-stablecoin FIFO doubly-linked list queue, with positions filled as matching supply arrives.
- Providing a `swapExactInput` endpoint for aggregator integration with deadline and recipient parameters.
- Enforcing a dynamic minimum order size for queue positions that scales with queue depth to deter spam.

`DLRS`

DLRS is a minimal ERC-20 token deployed by the `DollarStore` constructor, hardcoded to 6 decimals to match the underlying stablecoins. It is soulbound: `transfer` and `transferFrom` always revert, making it non-transferable between user addresses. Only the `DollarStore` contract can mint or burn tokens. Minting occurs on deposit and on certain swap paths. Burning occurs on withdrawal, queue entry, and DLRS-to-stablecoin swaps.

Security Model and Trust Assumptions

The protocol's security relies on the strict 1:1 accounting relationship between DLRS supply, outstanding queue depth, and total stablecoin reserves. All state-changing user functions are protected by a reentrancy guard and a pausing mechanism. DLRS minting and burning are exclusively controlled by the `DollarStore` contract through an immutable reference set at deployment, eliminating any upgrade or ownership risk on the token side.

The following trust assumptions are critical to the correct and safe operation of the protocol:

- **Admin honesty:** A single EOA holds admin privileges. This address can add or remove supported stablecoins, and pause or unpause the protocol. As such, the security of the entire system depends on this key not being compromised and on the admin acting honestly. There is no timelock or multisig on privileged operations.
- **Stablecoin standard compliance:** The protocol assumes that all whitelisted tokens are standard ERC-20 tokens with 6 decimals. Tokens with non-standard transfer behavior, such as fee-on-transfer and rebasing tokens, are not supported. Adding such a token would cause `_reserves` accounting to diverge from the contract's actual token balance, which would prevent some users from withdrawing and could allow others to withdraw more than their share.
- **Decimal conformity:** All supported stablecoins must use 6 decimal places. The protocol performs all accounting at 1:1 unit parity with no decimal normalization. Adding a token with a different decimal count (for example, an 18-decimal token) would cause exchange rate distortions that could be exploited to drain reserves.
- **Stablecoin contract integrity:** The admin is solely responsible for vetting stablecoin addresses before calling `addStablecoin`. The protocol performs no on-chain validation of contract code, decimals, or transfer behavior at the time of addition. An incorrectly or maliciously configured stablecoin addition represents a complete protocol risk.

High Severity

H-01 Depeg Risk Enables Toxic Flow and Socialized Loss Across DLRS Holders

The `DollarStore` contract settles all conversions at a fixed 1:1 rate in raw token units. `deposit` mints `dlrsMinted = amount`, `withdraw` burns DLRS and transfers amount of any supported stablecoin if `_reserves[stablecoin]` is sufficient, and `swap`/`swapExactInput`/`swapFromDLRS` all assume `amountOut = amountIn` with no valuation logic. This implicitly assumes every supported stablecoin is always worth exactly $1, and applies no oracle pricing, dynamic fee, exposure limit, or circuit breaker. `removeStablecoin` cannot be used as an emergency response while a stablecoin still has reserves.

The protocol offers a free 1:1 redemption on any supported stablecoin into the basket. If one stable depegs, rational actors can deposit the depegged asset and redeem good stables at par until those reserves are gone. Even assuming that "risk is on users", such a claim would be misleading because the system socializes the loss across all DLRS holders, not only the depegged-asset depositor. DLRS is a pro-rata claim on the basket, and if the basket becomes mostly depegged assets, everyone holding DLRS eats the loss.

With multiple stables, the contagion gets worse. Any good stable in reserves becomes a target for redemption by depegged depositors. In other words, the protocol enables conversion of good liquidity into bad especially under stress circumstances. Without depeg guardrails, the protocol is not just exposed to depeg risk, it actively arbitrages it, draining good reserves and leaving honest LPs with bad assets. This represents risk on all DLRS holders, including those who never touched the depegged asset.

Consider adding a depeg-aware intake policy that uses an oracle-derived peg check and a per-asset withdraw-only mode: deposits and swaps into a stablecoin should be blocked when its price is stale or outside conservative bounds, while redemptions out of that stablecoin remain enabled to let users exit. This preserves liveness without allowing toxic inflows. Expose admin controls to toggle per-asset modes and oracle parameters, with pause remaining as a last-resort control.

Update: *Resolved in pull request #14 at commit 0ce2731. The team stated:*

> *- Adds two-layer depeg protection to prevent depegged stablecoins from draining good reserves at par*
>
> *- **Oracle layer**: Chainlink price feed checks on inflows (deposit, swap, swapExactInput). Auto-blocks when price is stale (>1hr) or outside peg bounds (>0.5%)*
>
> *- **Manual layer**: Per-stablecoin `depositPaused` flag for admin override when oracle alone isn't sufficient*
>
> *- Outflows (withdraw, swapFromDLRS, cancelQueue) are never blocked — users can always exit*

Medium Severity

M-01 Unfillable Queue Head Position Can Permanently Block Queue Processing

The `DollarStore` contract uses a per-stablecoin FIFO queue to satisfy redemption demand when reserves are insufficient. Queue positions are created through `joinQueue` and the queue is filled opportunistically by user entrypoints such as `deposit`, `swap`, and `swapExactInput` via the internal `_processQueue` loop.

Queue fills are executed by calling `IERC20(stablecoin).safeTransfer` for the current head position. If this transfer reverts (for example, due to recipient blacklist or freeze), `_processQueue` reverts before it can remove the head position, and the calling user entrypoint reverts as well. For partial fills, the state update to `position.amount` is reverted, so no progress is made. The failing head position is retried on every subsequent attempt, effectively DoSing all flows that process that stablecoin queue. Additionally, `cancelQueue` is restricted to the position owner and there is no admin function to skip or evict positions, so liveness depends on the head owner cooperating.

Consider making queue fulfillment blacklist-resilient by decoupling queue progress from a single successful ERC-20 transfer. At a high level, this means attempting fills in a way that can fail without reverting the entire queue, tracking repeated transfer failures per position, and introducing a time or failure-bounded skip/age-out path that allows the queue to advance and eventually eject stuck positions with DLRS refunded to the owner. This preserves liveness in the presence of blacklisted recipients while keeping funds recoverable and behavior deterministic.

Update: *Resolved in pull request #11 at commit c76ee45.*

M-02 Minimum Queue Order Size Can Be Inflated via Cheap `positionCount` Manipulation

The `DollarStore` contract supports a FIFO queue per stablecoin entered through `joinQueue` when immediate redemption is not possible. `joinQueue` burns DLRS as escrow and relies on a minimum and increasing order size check to limit spam while `cancelQueue` mints the burned amount of DLRS back to the position owner.

The minimum order size is computed in `getMinimumOrderSize` from `MIN_ORDER_BASE` and `MIN_ORDER_SCALE_POSITIONS` as `MIN_ORDER_BASE * 10**(queue.positionCount / MIN_ORDER_SCALE_POSITIONS)` (integer division), and it is enforced by both `joinQueue` and `_createQueuePosition`. Since the only hard bound is `MAX_QUEUE_POSITIONS` (150), a single actor can open enough minimum-sized positions to push `queue.positionCount` across a tier boundary, raising the minimum for all subsequent users. Since cancellation refunds the burned amount, the primary cost is temporary capital lock and gas, which makes sustained access restriction against smaller orders economically feasible.

Consider decoupling the minimum order size from the mutable `positionCount`, or using an anti-spam mechanism that is not cheaply reversible (for example, a non-refundable fee or rate limiting). Consider also enforcing the next-tier minimum on the boundary-creating order (for example, validate against `positionCount + 1` rather than the pre-increment value) to prevent attackers from cheaply stepping over the boundary with a small order and then raising the minimum for everyone else. While this does not address all the possibilities for manipulation, it does eliminate a simple edge case.

Update: *Partially Resolved in pull request #12 at commit 6abb4be. The boundary-creating order is now forced into the next tier via `positionCount + 1`, but the core manipulation vector (cheaply inflating `positionCount` with refundable positions) still exists. The team stated:*

> *Regarding the broader structural recommendations (decoupling from `positionCount`, non-refundable fees): we believe the residual manipulation risk is sufficiently mitigated by the economics of the queue itself. Positions entered into the queue are not inert locked capital — they are valid FIFO redemption orders that get filled by incoming liquidity in order. Any attacker attempting to sustain elevated tier minimums must continuously cycle real capital through legitimate swap fulfillments.*

Low Severity

L-01 `_reserves` Drift Strands Funds and Enables Self-Transfer Mismatch

The `DollarStore` contract maintains internal reserves in the `_reserves` mapping. Availability checks and payouts in `withdraw`, `swap`, `swapFromDLRS`, and `swapExactInput` use `_reserves` as the sole cap, while increments to `_reserves` occur only in protocol code paths (e.g., after `safeTransferFrom` in `deposit`).

As a result, any supported stablecoin that reaches the contract without incrementing `_reserves` (direct ERC-20 transfers, airdrops, or other forced transfers) becomes stranded since no function reconciles balances to reserves. In addition, `swapExactInput` allows `recipient == address(this)` (recipient validation) and still decrements `_reserves[toStablecoin]` before transferring. For standard ERC-20 implementations, a transfer from the contract to itself does not reduce its balance, creating permanently unaccounted excess tokens that cannot be used because future withdrawals and swaps remain capped by `_reserves`.

Consider adding a `syncReserves(stablecoin)` function that reconciles `_reserves[stablecoin]` to `IERC20(stablecoin).balanceOf(address(this))` and a sweep mechanism for unexpected balances, but only allow sweeping when there are no outstanding obligations. This means checking that DLRS has no supply and that all queue positions are empty, since queue entries burn DLRS and represent separate claims. Consider also forbidding `recipient == address(this)` in `swapExactInput` and documenting that `_reserves` is not the same as the actual token balance.

Update: *Resolved in pull request #6 at commit 1949c93. The team stated:*

> *- Add syncReserves(stablecoin): admin-only, syncs _reserves DOWN to match balanceOf after external events like token seizure*
>
> *- Add rescueTokens(stablecoin, to): admin-only, sweeps excess tokens (balanceOf - _reserves) that arrived outside protocol flows (direct transfers, airdrops)*
>
> *- Block recipient == address(this) in swapExactInput to prevent self-transfer creating permanently unaccounted tokens*

L-02 `removeStablecoin` Can Delist a Stablecoin While Its Queue Is Non-Empty, Leaving Positions Unfillable

Users create redemption positions via `joinQueue`, which burns DLRS and appends a `QueuePosition` to `_queues[stablecoin]`. Positions are only filled upon stablecoin inflows: `deposit`, `swap`, and `swapExactInput` call the `internal` `_processQueue` for the input stablecoin.

`removeStablecoin` only checks `_reserves[stablecoin] == 0` and then sets `_isSupported[stablecoin] = false`, without validating that `_queues[stablecoin]` is empty. After delisting, all external entry points that could call `_processQueue` revert due to their `_isSupported[stablecoin]` checks, and `_processQueue` is `internal`. This makes existing positions for the removed stablecoin unfillable until the stablecoin is re-added and forces users to exit via `cancelQueue`, changing the expected redemption asset.

Consider preventing delisting while `_queues[stablecoin].positionCount > 0` (or `totalDepth > 0`), or adding a close-out mechanism that cancels/settles all remaining positions before flipping `_isSupported[stablecoin] = false`.

Update: *Resolved in pull request #8 at commit 69f65d7. The team stated:*

> *Adds a totalDepth > 0 check in removeStablecoin to prevent delisting a stablecoin while queue positions are still active, which would leave those positions unfillable.*

L-03 `Swap` Event Reports `amountQueued` Even When No Queuing Occurs

The `Swap` event is defined in the `IDollarStore` interface and includes an `amountQueued` field. Actual queuing is represented on-chain via `_createQueuePosition` and the `QueueJoined`/`QueueFilled`/`QueueCancelled` events. This makes `amountQueued` a natural field for off-chain indexers to interpret as the amount that became a queued obligation.

However, both `swap` and `swapFromDLRS` emit `Swap(..., amount - received)` unconditionally. In the `queueIfUnavailable == false` partial-fill branch of `swap`, the remainder is minted as DLRS and no call to `_createQueuePosition` occurs. In the `queueIfUnavailable == false` partial-fill branch of `swapFromDLRS`, only `received` is burned, and the remainder remains in the caller's DLRS balance, with no queue position

created. In both cases, `amountQueued` is non-zero even though nothing was queued, which can break off-chain accounting and integrations that rely on `Swap.amountQueued`.

Consider emitting `amountQueued` as the amount actually passed to `_createQueuePosition` (and emitting `0` when no position is created). In addition, consider documenting the intended meaning of `Swap.amountQueued` in `IDollarStore` to reduce ambiguity for indexers.

Update: *Resolved in pull request #9 at commit 15c3ede. The team stated:*

> *When queueIfUnavailable=false and a partial fill occurs, the Swap event was emitting a non-zero amountQueued even though no queue position was created. Now emits 0 when no position is created, preventing off-chain indexers from overcounting queue volume.*

L-04 Partial `swap` Can Mint Non-Transferable DLRS Tokens to Contract Callers, Trapping Value

The `DollarStore` contract exposes `swap`, which always pays outputs to `msg.sender` and provides a `queueIfUnavailable` switch. When the output reserve is partially available and `queueIfUnavailable` is `false`, the function transfers the available `toStablecoin` amount and mints the remaining as DLRS to `msg.sender`. Since DLRS tokens are non-transferable, router-based integrations can cause the minted remainder to become practically unrecoverable for the end user unless the router implements explicit DLRS redemption logic (e.g., `withdraw`/`swapFromDLRS` from the router itself).

Consider reverting (or always queueing) when reserves are insufficient, instead of minting DLRS as implicit change. In addition, consider adding a `recipient` parameter (for both the stablecoin output and any DLRS change) and documenting that integrations should prefer `swapExactInput` when a custom recipient is required.

Update: *Resolved in pull request #10 at commit ea2e5ea. The team stated:*

> *Previously, a partial fill with queueIfUnavailable=false would silently mint DLRS as change (swap) or leave unburned DLRS with the caller (swapFromDLRS). This traps value when the caller is a router contract, since DLRS is non-transferable. Now reverts with InsufficientReservesNoQueue, consistent with the zero-reserves path.*

Notes & Additional Information

N-01 DLRS ERC-20 Adds Surface Area Without Practical Composability

The current architecture uses a separate ERC-20 receipt token (DLRS) that is explicitly non-transferable and only minted/burned by `DollarStore`. Since DLRS cannot be transferred, it does not allow for composability or external utility as of now, yet it introduces additional code paths (mint/burn, ERC-20 logic, allowance UX) and state that the protocol must reason about alongside its own queue bookkeeping. A simple internal accounting model inside the `DollarStore` contract would provide the same functionality with fewer moving parts and a smaller attack surface.

Consider replacing the DLRS token with internal accounting for balances plus the existing queue state. This reduces complexity, eliminates unused ERC-20 features (like allowances), and keeps all balance semantics localized to the core contract.

Update: *Resolved in pull request #7 at commit b94a3fe. The team stated:*

> *Address audit finding N-01: document why DLRS is retained as a separate ERC-20 rather than internal accounting. The token provides wallet visibility, block explorer indexing, and a foundation for future composability. The additional surface area concern is mitigated by N-02 (blocking approve).*

N-02 DLRS Reverts on Transfers but Still Allows `approve`

The DLRS token is intentionally non-transferable with `transfer` and `transferFrom` reverting with `NonTransferable`. However, `approve` is inherited from ERC-20 and still succeeds, allowing users to set allowances that can never be used. This creates a confusing and non-standard ERC-20 surface where approvals appear to work, but any downstream transfer will always revert. Some tooling and contracts assume that a successful `approve` implies eventual `transferFrom` will succeed, and may not handle the hard-revert semantics gracefully. It also invites wasted gas and false expectations.

Consider overriding `approve` in the `DLRS` contract (and optionally `increaseAllowance`/`decreaseAllowance`) to revert with `NonTransferable`, consistent with the non-transferable design. This makes the token's behavior explicit and avoids misleading allowance state.

Update: *Resolved in pull request #5 at commit 646f241. The team stated:*

> *Address audit finding N-02: override approve in DLRS.sol to revert with NonTransferable, consistent with transfer and transferFrom. Approve was inherited from ERC20 and succeeded silently, allowing users to set allowances that could never be used. Update test to cover new approve behavior.*

N-03 Unclear Documentation for `minAmountOut` Parameter of `swapExactInput`

The aggregator-focused `swapExactInput` function intentionally mirrors common router signatures, including `minAmountOut`. However, the current documentation states that `minAmountOut` is for "slippage protection" and implies that it is enforced, whereas the function body never reads the parameter and always sets `amountOut = amountIn`. As such, callers may assume a revert-on-slippage guarantee that does not exist, even if the 1:1 design makes slippage irrelevant.

Consider updating the documentation to clearly state that `minAmountOut` is unused and included only for interface compatibility, and that swaps are always 1:1 by design.

Update: *Resolved in pull request #4 at commit ba1841b. The team stated:*

> *Address audit finding N-03: update NatSpec in DollarStore.sol and IDollarStore.sol to clarify that minAmountOut is unused and included only for router interface compatibility. Swaps are always 1:1 by design. External documentation site also updated.*

N-04 Missing Security Contact

Providing a specific security contact (such as an email address or ENS name) within a smart contract significantly simplifies the process for individuals to communicate if they identify a vulnerability in the code. This practice is quite beneficial as it permits the code owners to dictate the communication channel for vulnerability disclosure, eliminating the risk of miscommunication or failure to report due to a lack of knowledge on how to do so. In addition, if the contract incorporates third-party libraries and a bug surfaces in those, it becomes easier

for their maintainers to contact the appropriate person about the problem and provide mitigation instructions.

Throughout the codebase, multiple instances of contracts not having a security contact were identified:

- The `DLRS` contract
- The `DollarStore` contract

Consider adding a NatSpec comment containing a security contact above each contract definition. Using the `@custom:security-contact` convention is recommended as it has been adopted by the OpenZeppelin Wizard and the ethereum-lists.

Update: *Resolved in pull request #3 at commit 5e091bd.*

N-05 Missing Docstrings

Throughout the codebase, multiple instances of missing docstrings were identified:

- In `DLRS.sol`, the `dollarStore` state variable
- In `DollarStore.sol`:
  - The `MAX_QUEUE_POSITIONS` state variable
  - The `MIN_ORDER_BASE` state variable
  - The `MIN_ORDER_SCALE_POSITIONS` state variable
  - The `AdminTransferInitiated` event
  - The `AdminTransferCompleted` event
  - The `addStablecoin` function
  - The `removeStablecoin` function
  - The `transferAdmin` function
  - The `acceptAdmin` function
  - The `pause` function
  - The `unpause` function

Consider thoroughly documenting all functions (and their parameters) that are part of any contract's public API. Functions implementing sensitive functionality, even if not public, should be clearly documented as well. When writing docstrings, consider following the Ethereum Natural Specification Format (NatSpec).

Update: *Resolved in pull request #2 at commit 085b91b.*

N-06 Redundant Return Statements

To improve the readability of the contract, it is recommended to remove redundant return statements from functions that have named returns.

Within `DollarStore.sol`, multiple instances of redundant return statements were identified:

- The return statement of `getQueuePosition`
- The return statement of `getUserQueuePositions`
- The return statements of `getSwapQuote`
- The return statement of `_processQueue`

Consider removing the redundant return statement in functions with named returns to improve the readability of the contract. Alternatively, if the named return variable is unused, consider removing it from the function signature and use explicit returns instead.

Update: *Resolved in pull request #1 at commit 5508a03.*

Conclusion

The DollarStore codebase implements a simple 1:1 stablecoin swapping and queueing system with a DLRS receipt token. While the design keeps on-chain logic compact, it relies heavily on stablecoin behavior and operational policies (asset support, queue handling, and reserve accounting) to maintain safety and liveness.

The most significant risk identified is a depeg scenario that enables toxic flow. Under stress, value can migrate from stronger assets into weaker ones, socializing losses across all receipt-token holders. Two medium-severity issues relate to queue liveness and access control: a single unfillable position can stall the queue, and the minimum order threshold can be pushed upward at low cost, restricting smaller users.

Several lower-severity issues were also observed around accounting drift from unaccounted transfers, delisting assets with pending queue positions, misleading event semantics, and partial fills that can strand value for contract-based callers. Additional notes highlight design and UX considerations around the receipt token and interface documentation.

The DollarStore team is appreciated for their support and responsiveness during the 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.