

DOLLARSTORE · SECURITY REVIEW

DollarStore

Security Review.

A security review of DollarStore.

CLIENT	PROJECT	AUDITORS	AUDIT WINDOW	FINAL REPORT
DollarStore	DollarStore	Vlad Zotov, Michael Lett, Ferran Celades, Minato7namikazi, Ali Shehab	October 13 – October 26, 2026	October 27, 2026

§ 00

Contents.

32 pages: overview, confidence ranking, scope, 11 findings, disclaimer, and about Guardian.

§ 01	Overview	P. 03
§ 02	Confidence Ranking	P. 04
§ 03	Scope & Classifications	P. 05
	Files in Scope	P. 06
§ 04	Findings	P. 08
	Main Review · 7 findings	P. 09
	Remediation Review · 4 findings	P. 25
§ 05	Disclaimer	P. 31
§ 06	About Guardian	P. 32

§ 01

Overview.

Guardian reviewed DollarStore across 2 review rounds. The final report includes 4 Low, 7 Informational findings.

AUDIT FIRM	Guardian	REPOSITORY	https://github.com/coasify/dollar
CLIENT	DollarStore	METHODOLOGY	Smart contract · Manual · Main Review · Remediation Review
PROJECT	DollarStore	MAIN REVIEW COMMIT	f78ae3733485
PREPARED BY	Vlad Zotov, Michael Lett, Ferran Celades, Minato7namikazi, Ali Shehab	REMEDATION REVIEW COMMIT	ae25d5a48dce
AUDIT WINDOW	October 13 – October 26, 2026		
FINAL REPORT	October 27, 2026		

1.1 Confidence Ranking

5 / 5

Very High Confidence

Guardian assigns a Confidence Ranking of 5 out of 5, combining finding severity, remediation outcome, and the reviewer’s assessment of latent risk. The full scale is on the next page.



1.2 Findings at a Glance

SEVERITY	TOTAL	PENDING	ACK	PARTIAL	RESOLVED	FIX REVIEW
Critical	0	0	0	0	0	0
High	0	0	0	0	0	0
Medium	0	0	0	0	0	0
Low	4	0	0	0	4	0
Informational	7	0	0	0	3	4
All severities	11	0	0	0	7	4

§ 02

Confidence Ranking.

Guardian assigns a project-level confidence score using the scale below. The ranking combines finding severity, remediation outcome, and the reviewer's assessment of latent risk.

CONFIDENCE RANKING	DEFINITION AND RECOMMENDATION	RISK PROFILE
5 Very High Confidence ASSIGNED	Codebase is mature, clean, and secure. No High or Critical vulnerabilities were found. Follows modern best practices with high test coverage and thoughtful design. Recommendation: code is highly secure at time of audit.	0 High/Critical findings and few Low/Medium severity findings.
4 High Confidence	Code is clean, well-structured, and adheres to best practices. Design patterns are sound and test coverage is strong. Recommendation: suitable for deployment after remediations; consider periodic review with changes.	0-1 High/Critical findings per engagement week and little to no Medium severity findings.
3 Moderate Confidence	Code is functional but shows gaps in defensive programming or test coverage. Recommendation: remediate the findings and re-review the changes before deployment.	Some High/Critical findings, with various Medium severity findings.
2 Low Confidence	Recurring issues suggest systemic gaps rather than isolated mistakes. Recommendation: remediate, then commission a follow-up review at a frozen commit.	Over 6 High and Critical findings across the engagement.
1 Very Low Confidence	The volume and severity of findings indicate the codebase is not ready for production. Recommendation: substantial rework, then a full re-review.	Over 12 High and Critical findings across the engagement.

§ 03

Scope & Classifications.

Guardian reviewed DollarStore in `https://github.com/coasify/dollar`. The scope includes 2,151 source lines across 19 files.

3.1 Severity Matrix

	IMP · HIGH	IMP · MED	IMP · LOW
LIK · HIGH	Critical	High	Medium
LIK · MED	High	Medium	Low
LIK · LOW	Medium	Low	Low

IMPACT · HIGH
Significant loss of assets, harm to a group of users, or core protocol functionality is disrupted.

IMPACT · MEDIUM
Small amount of funds can be lost, or ancillary functionality is affected. Users may experience reduced or delayed receipt.

IMPACT · LOW
Unexpected behavior in some functionality that is notable but does not meet the criteria for higher severity.

LIKELIHOOD · HIGH
Possible with reasonable assumptions that mimic on-chain conditions; cost is low compared to amount gained.

LIKELIHOOD · MEDIUM
Possible only in uncommon cases or requires a large amount of capital relative to disruption caused.

LIKELIHOOD · LOW
Unlikely to ever occur in production.

Files in Scope 1/2.

FILE	NSLOC	TOTAL
MAIN REVIEW		
foundry.toml	37	48
script/CreateSpoke.s.sol	78	113
script/Deploy.s.sol	32	57
script/DeployGovernance.s.sol	83	156
script/Upgrade.s.sol	17	42
src/DLRS.sol	35	68
src/DollarStore.sol	515	898
src/interfaces/AggregatorV3Interface.sol	3	7
src/interfaces/IDollarStore.sol	20	52
src/libraries/NormalizationLib.sol	18	52
src/libraries/PegLib.sol	57	94
src/libraries/QueueLib.sol	105	152
src/libraries/SpokeAdminLib.sol	75	118
src/libraries/SpokeLifecycleLib.sol	201	290
src/libraries/SpokeShareLib.sol	23	60
src/libraries/SwapRouteLib.sol	339	487
src/storage/CoreStorage.sol	23	58
src/storage/QueueStorage.sol	45	127
src/storage/RegistryStorage.sol	46	105
REMEDIATION REVIEW		
foundry.toml	0	48
script/CreateSpoke.s.sol	36	87
script/Deploy.s.sol	0	57
script/DeployGovernance.s.sol	0	156
script/Upgrade.s.sol	0	42
src/DLRS.sol	0	68
src/DollarStore.sol	64	902
src/interfaces/AggregatorV3Interface.sol	0	7
src/interfaces/IDollarStore.sol	36	50
src/libraries/NormalizationLib.sol	0	52
src/libraries/PegLib.sol	34	120
src/libraries/QueueLib.sol	0	151
src/libraries/SpokeAdminLib.sol	8	124
src/libraries/SpokeLifecycleLib.sol	120	320
src/libraries/SpokeShareLib.sol	0	60

Files in Scope 2/2.

FILE	NSLOC	TOTAL
src/libraries/SwapRouteLib.sol	101	528
src/storage/CoreStorage.sol	0	57
src/storage/QueueStorage.sol	0	122
src/storage/RegistryStorage.sol	0	104

§ 04

Findings.

Grouped by round. Visible IDs reset by severity within each round, so a finding is named the same way in the report as it was in the channel it was written in. Status is where each finding ended up; its own page says how.

ID	TITLE	CATEGORY	SEVERITY	STATUS	PAGE
MAIN REVIEW					
L-01	canInflow panics on a malformed feed instead of returning false	Unexpected Behavior	LOW	RESOLVED	p. 9
L-02	A scheduled governor rescue run mid-swap sweeps pulled-but-unbooked escrow and leaves the queue position unbacked	Reentrancy	LOW	RESOLVED	p. 11
L-03	Stale pool ID in capped-listing batches leaves a new spoke uncapped while overwriting another pool's cap	Validation	LOW	RESOLVED	p. 13
L-04	A ready governor syncReserves run from a refund or payout callback writes reserves down by the in-flight amount a second time	Reentrancy	LOW	RESOLVED	p. 15
I-01	Security NatSpec names the guardian as canceller, the governor as upgrade authority and the guardian as deposit pauser, contradicting the shipped role model	Documentation	INFORMATIONAL	RESOLVED	p. 18
I-02	Relisting a removed asset silently clears a pauser's deposit pause and the probe exemption, bypassing the unpauser	Access Control	INFORMATIONAL	RESOLVED	p. 20
I-03	A single dust queue position makes a dry via revert a spoke-to-spoke swap instead of being skipped	DoS	INFORMATIONAL	RESOLVED	p. 22
REMEDIATION REVIEW					
I-01	Production runbook lists hub assets before activating the launch cap	initialization-ordering	INFORMATIONAL	REPORTED	p. 25
I-02	Donations can invalidate atomic timelock recovery batches and force a fresh delay	denial-of-service	INFORMATIONAL	REPORTED	p. 26
I-03	swapSpokeToSpoke reports a hub pause or an unhealthy via as InsufficientLiquidity	Unexpected Behavior	INFORMATIONAL	REPORTED	p. 27
I-04	The integrator error table names the wrong error for a hub exit above the caller's balance and omits the deposit errors	Documentation	INFORMATIONAL	REPORTED	p. 29

LOW **L-01**

RESOLVED

canInflow panics on a malformed feed instead of returning false

CATEGORY Unexpected Behavior

LOCATION `src/libraries/PegLib.sol:65 (canInflow); same underflow in checkPeg at src/libraries/PegLib.sol:48`

REVIEW Main Review

DESCRIPTION

`PegLib.canInflow` is documented as non-reverting ("a reverting feed reads as false") and exists so callers that must SKIP rather than revert on an unhealthy asset can do so: `SwapRouteLib.triggerSpokeQueues` calls it to skip a deposit-paused/off-peg offer so an unrelated bad asset cannot DoS a legitimate deposit, and `SwapRouteLib.quote` uses it to return 0.

The try/catch in `canInflow` only wraps the external `latestRoundData()` call. The staleness subtraction `block.timestamp - updatedAt` and the scaling `10 ** (PRICE_DECIMALS - uint256(agg.decimals()))` execute inside the try SUCCESS block, so a revert from them is NOT caught by `catch { return false; }`. A feed that reports a future `updatedAt` (`updatedAt > block.timestamp`) therefore underflows and panics (Panic 0x11) instead of returning false. `checkPeg` repeats the same `block.timestamp - updatedAt` underflow.

Impact: for the affected asset, `canInflow` reverts rather than skipping, so `triggerSpokeQueues` (and any caller relying on the non-reverting form) reverts the whole call. A legit spoke deposit that would otherwise settle queues while skipping the bad asset instead reverts, i.e. the exact DoS the non-reverting API was written to avoid.

The blast radius is bounded to the asset with the malformed feed and can be contained by a pauser pausing that asset/pool, which is why this is low severity rather than higher

RECOMMENDATION

Either bound and validate the feed values before the arithmetic (reject `updatedAt > block.timestamp` by returning false), or move the subtraction / scaling into a helper that can revert and be invoked through try / catch so the failure is converted to false.

Apply the same guard to `checkPeg`'s staleness subtraction so a future `updatedAt` reverts with a clear error rather than a bare panic

L-01 canInflow panics on a malformed feed instead of returning false

CONTINUED

— RESOLUTION

Status: Resolved.

DollarStore Team: Fixed in 1b3aae7, wider than recommended. canInflow returns false for a future updatedAt, a reverting decimals(), more than 18 feed decimals and an answer whose scaling would overflow. It also reads the feed with a raw staticcall, so a response that is too short, has out-of-range uint80 or uint8 values, or comes from an address with no code reads as unhealthy instead of reverting. Each feed call, in canInflow and checkPeg, may use at most 100,000 gas, so a feed that never answers fails the check instead of using up the caller's gas; a standard Chainlink read measured about 16,000. As recommended, checkPeg reverts PriceStale for a future updatedAt. A fuzz test checks that, for any feed response, a queue offering that asset is settled exactly when a deposit of it succeeds, so canInflow and checkPeg agree.

<https://github.com/coasify/dollar/pull/47>

LOW **L-02**

RESOLVED

A scheduled governor rescue run mid-swap sweeps pulled-but-unbooked escrow and leaves the queue position unbacked

CATEGORY Reentrancy

LOCATION `src/DollarStore.sol:888 (rescueTokens wrapper) -> src/libraries/SpokeAdminLib.sol:113 (rescueTokens body)`

REVIEW Main Review

— DESCRIPTION

`DollarStore.rescueTokens` is `onlyGovernor` but not `nonReentrant`. It is the only token-moving entry point in the contract without the guard (`swap`, `swapExactInput`, `swapSpokeToSpoke`, `deposit`, `withdraw`, `cancelQueue`, `processQueue`, and even the non-token-moving `haircutEscrow` all carry it). `foundry.toml` even documents the intent, excluding `reentrancy-events` because "every function that moves tokens is `nonReentrant`", `rescueTokens` breaks that rule

`SwapRouteLib.swap` performs the following order:

- 1 - `pullExact(offerAsset, nativePulled)` pulls the offer into the contract
- 2 - `fillDirected()` fills from the opposite queue / reserves
- 3 - `IERC20(wantAsset).safeTransfer(msg.sender)` pays the caller the filled `wantAsset`
- 4 - `enqueueRemainder()` only now books the unfilled remainder as queue escrow

Step 3 is an external transfer to the attacker (the swapper) that happens before step 4 records the escrow. A listed asset whose transfer invokes a hook (the repo's own `MockReentrantERC20` models exactly this, and every other token-moving path is guarded against it) executes attacker code here.

`rescueTokens` computes `excess = balanceOf(this) - (escrow + reserves)` and transfers it out. At the reentrancy point the pulled remainder is present in the balance but is not yet in `totalEscrowedByAsset`, so it is counted as excess and swept to the rescue recipient. `swap` then records the queue position, which is now unbacked: `balance < reserves + escrow` by exactly the remainder.

The rescue is a privileged/governor action, but the governor timelock is intentionally configured with an open executor (`executors = [address(0)]`), so any unprivileged account can execute an already-scheduled, ready `rescueTokens` at the moment of its choosing,

L-02 A scheduled governor rescue run mid-swap sweeps pulled-but-unbooked escrow and leaves the queue position unbacked CONTINUED

including from inside the swap's outbound-transfer callback. The attacker does not need to control the rescue's `to` address; the harm is that a legitimate scheduled rescue over-sweeps.

This violates the documented guarantee: "rescueTokens sweeps only balance above what the books account for, and queue escrow is always accounted for" and "no role can take escrow or reserves out of the contract"

Impact: with the escrow aggregate short, a later honest claimant can be parked/unpaid once earlier payouts consume the surviving balance.

Preconditions

A listed asset whose transfer calls back (hook/ERC777-style). The protocol treats such tokens as in scope (it ships MockReentrantERC20 and guards every other token-moving path), and listing enforces only a receiver probe, not absence of hooks

A governor `rescueTokens` operation that is scheduled and past its timelock delay (ordinary governance activity)

— RECOMMENDATION

Add `nonReentrant` to `DollarStore.rescueTokens`, mirroring `haircutEscrow`

— RESOLUTION

Status: Resolved.

DollarStore Team: Fixed in ab78b9c, as recommended. `rescueTokens` is `nonReentrant`, so a rescue executed from a token callback during a swap reverts. The test uses a real `TimelockController` with an open executor and a token that executes the ready rescue from its payout callback; the swap reverts. A second test checks that the same rescue, run after the swap, sweeps only the excess and leaves reserves and escrow fully backed.

Separately, SECURITY.md and DEPLOY.md now state as a listing rule that tokens with transfer hooks are not listed (a50bab4). The guard stays in place either way.

<https://github.com/coasify/dollar/pull/47>

LOW **L-03**

RESOLVED

Stale pool ID in capped-listing batches leaves a new spoke uncapped while overwriting another pool's cap

CATEGORY Validation

LOCATION `script/CreateSpoke.s.sol:55-66, 71-84, 95-113; src/libraries/SpokeAdminLib.sol:52-65; src/DollarStore.sol:937-942`

REVIEW Main Review

DESCRIPTION

The script schedules two actions together: "create the new pool" and "cap it", guessing the new pool's future ID number from today's pool count. That guess is frozen into the payload at schedule time:

```
// script/CreateSpoke.s.sol:55,66 ( id baked in at schedule time )
expectedPoolId = uint16(DollarStore(store).poolCount()); // read NOW
payloads[1] = abi.encodeCall(DollarStore.setLaunchCap, (expectedPoolId, launchCap)); // id frozen
```

That guess is not tied to the pool the batch actually creates. The real ID is assigned only when the transaction runs, based on how many pools exist then:

```
// src/libraries/SpokeAdminLib.sol:57-63 ( real id assigned at EXECUTION time )
uint256 newIndex = r.pools.length; // grows if any other createSpoke lands first
poolId = uint16(newIndex); // != expectedPoolId -> wrong pool capped
```

So if someone else creates a different pool in between, the new pool lands on a different number, the cap is applied to the wrong (older) pool, and the new one launches uncapped.

The safety check inside `execute()` is fake: it compares the pool count to itself in the same moment, so it always passes

```
// script/CreateSpoke.s.sol:107-110
require(
    targets.length == 1 || uint16(DollarStore(store).poolCount()) == expectedPoolId, // both
    sides = poolCount()
    "poolCount moved since scheduling: re-schedule"
);
```

The script itself is safe-ish (rebuilding the batch yields a new hash, so the timelock says "not ready"), but the batch already scheduled on-chain stays valid and anyone can run it, since execution is open `executors = [address(0)]`

L-03 Stale pool ID in capped-listing batches leaves a new spoke uncapped while overwriting another pool's cap

CONTINUED

Result: exposure limits get silently messed up: one pool can accept more deposits than intended, another pool's cap is changed unexpectedly

Impact is bounded and operational; it just breaks risk limits until governance fixes the caps

And needs two listings overlapping in time plus someone executing the stale batch

— RECOMMENDATION

Bind creation and its launch cap in one on chain operation using the id actually created, preferably by adding a cap argument to createSpoke (or having it return and atomically apply the cap) so a stale predicted id cannot exist

— RESOLUTION

Status: Resolved.

DollarStore Team: Fixed in bbb13a4, as recommended. createSpoke takes the launch cap as a fourth argument and sets it on the pool it creates, in the same call. CreateSpoke.s.sol queues that one call and no longer predicts a pool id.

<https://github.com/coasify/dollar/pull/47>

LOW **L-04**

RESOLVED

A ready governor syncReserves run from a refund or payout callback writes reserves down by the in-flight amount a second time

CATEGORY Reentrancy

LOCATION `src/DollarStore.sol:881 (syncReserves wrapper) -> src/libraries/SpokeAdminLib.sol:86-102 (syncReserves body); windows at src/libraries/SwapRouteLib.sol:270-271 (cancelPosition) and src/libraries/SwapRouteLib.sol:488-489 (settleSameDirection)`

REVIEW Main Review

DESCRIPTION

This uses the same vector and preconditions as M-01 (a ready governor operation executed through the open timelock executor from a hook-token callback), but on `syncReserves` instead of `rescueTokens`, in different windows (a cancel refund and a settlement payout), and with a different effect: reserves are written down below the real balance instead of escrow being swept.

`DollarStore.syncReserves` is `onlyGovernor` but not `nonReentrant`. Its body sets the reserve from the live token balance:

```
// src/libraries/SpokeAdminLib.sol:92-100 (excerpt)
uint256 actualUnits = IERC20(asset).balanceOf(address(this)) / cfg.scalingFactor;
uint256 escrowUnits = QueueStorage.layout().totalEscrowedByAsset[asset];
uint256 prevReserves = r.reserves[poolId][asset];
if (actualUnits >= prevReserves + escrowUnits) revert
    IDollarStore.ReservesNotDrifted(asset);
uint256 newReserves = actualUnits - escrowUnits;
r.reserves[poolId][asset] = newReserves;
```

Two paths pay tokens out before they update the books:

- 1 - `SwapRouteLib.cancelPosition` refunds the owner, and only then `QueueLib.remove` lowers escrow
- 2 - `SwapRouteLib.settleSameDirection` pays the owner, and only then `applyReserveFill` lowers reserves (reached from `processQueue`, inline settlement in swaps, and the spoke-deposit trigger)

```
// src/libraries/SwapRouteLib.sol:259-271 (cancelPosition)
/// Safe because the wrapper is nonReentrant, so the token call cannot re-enter a
/// mutator.
...
if (tryTransfer(offerAsset, owner_, NormalizationLib.toNative(amount, scaling))) {
```

L-04 A ready governor syncReserves run from a refund or payout callback writes reserves down by the in-flight amount a second time CONTINUED

```
QueueLib.remove(qs, positionId);
```

The code relies on the guard to make this ordering safe ("the token call cannot re-enter a mutator"), but `syncReserves` is a mutator the guard does not cover, and it derives its write from the very balance the in-flight transfer has just changed.

The governor timelock is configured with an open executor (`executors[0] = address(0)`, `script/DeployGovernance.s.sol:84`), so any unprivileged account can execute a ready `syncReserves` operation at the moment of its choosing, including from inside the refund or payout callback of its own position.

At that moment the balance is already down by the in-flight amount A while the books still hold it, so the drift check passes on the transfer alone and reserves are written to `balance - escrow`. The outer path then lowers escrow (cancel) or reserves (settle) by A again:

- Cancel window: reserves end at $R - L - A$ against a balance of $R - L$, where R is the booked reserve and L the real loss. A tokens stay in the contract outside the books, and since `syncReserves` only ever lowers reserves, no function books them back.

- Settle window: the same payout is subtracted twice, and reserves end at $R - L - 2A$ against a balance of $R - L - A$.

No real loss is needed. A ready operation whose normal execution reverts `ReservesNotDrifted` is not consumed by that revert, and OpenZeppelin timelock operations never expire, so it stays ready and succeeds inside the window, writing reserves to $R - A$ with zero loss.

Example: $R = 1,000,000$, $L = 50,000$, $A = 200,000$. The honest sync writes 950,000. Run from the cancel refund callback it writes 750,000, and a holder's withdrawal of 950,000 that succeeds without the attack reverts `InsufficientReserves(asset, 950_000e6, 750_000e6)`. With $A = R - L$ the reserve is written to 0.

This violates the documented guarantee: "`syncReserves` only ever decreases reserves, to true them up to the real balance after an external loss" (SECURITY.md, Reserve accounting) and "Sync recorded reserves down to the actual balance minus queue escrow" (IDollarStore.syncReserves).

Impact: up to the whole booked reserve of the targeted asset stops backing DLRS (hub asset) or the spoke's LP shares (spoke asset) while the tokens stay in the contract, redemptions revert `InsufficientReserves` once the gap is reached, and the governor's write-down operation is consumed.

Preconditions: a listed asset whose transfer calls back the recipient (hook/ERC777-style), a class the protocol treats as in scope (it ships `MockReentrantERC20`), guards every

L-04 A ready governor syncReserves run from a refund or payout callback writes reserves down by the in-flight amount a second time CONTINUED

user-facing token-moving path, justifies the cancel path's ordering in code by the reentrancy guard, and its listing preconditions in SECURITY.md and DEPLOY.md exclude only sender-only freezes and tokens that reject zero-value transfers, not tokens with transfer hooks); and a governor `syncReserves` operation that is scheduled and past its timelock delay, which is ordinary loss-reconciliation activity.

— RECOMMENDATION

Add `nonReentrant` to `DollarStore.syncReserves`, and apply the same guard to any other governor function that derives a write from the live token balance or from pool state that a user action changes mid-flight (for example `removePool`), so no such operation can execute while a payout or refund is in progress. Timelock batches are unaffected because each call in a batch enters and leaves the guard separately.

— RESOLUTION

Status: Resolved.

DollarStore Team: Fixed in 58e9e9a, as recommended. `syncReserves` is `nonReentrant`, so a sync executed from a cancel refund or a queue payout callback reverts. If the token passes that revert on, the transfer fails and the position is parked with its escrow; if it swallows it, nothing was synced. As suggested, `addHubAsset`, `createSpoke`, `windDownSpoke` and `removePool` are `nonReentrant` too, so no governor action that changes pool or asset state runs inside a token transfer. Timelock batches are unaffected.

Separately, SECURITY.md and DEPLOY.md now state as a listing rule that tokens with transfer hooks are not listed (a50bab4). The guards stay in place either way.

<https://github.com/coasify/dollar/pull/47>

INFORMATIONAL I-01

RESOLVED

Security NatSpec names the guardian as canceller, the governor as upgrade authority and the guardian as deposit pauser, contradicting the shipped role model

CATEGORY	Documentation
LOCATION	src/DollarStore.sol:29 (contract notice); src/DollarStore.sol:52-57 (MAX_STALENESS rationale); src/storage/RegistryStorage.sol:41 (depositPaused); src/interface s/IDollarStore.sol:4, 260, 266-268 (deposit, NotEnabled)
REVIEW	Main Review

DESCRIPTION

The NatSpec on the core contract, its interface and its storage still describes milestone-era role assignments that the shipped code and SECURITY.md contradict. The 3-day `MAX_STALENESS` ceiling is justified as harmless because a wrong governor action is "cancellable by the guardian". However, the guardian holds no canceller role: SECURITY.md:66 says "The guardian is not a canceller", DeployGovernance.s.sol:29 says "The guardian holds no cancel power", and on mainnet `checkSeparation` requires the guardian to differ from the veto Safe and both proposer Safes so that it holds none. The contract notice calls upgrades "governor-gated", but `_authorizeUpgrade` is `onlyUpgrader` (DollarStore.sol:988), a separate role behind a 14-day timelock. `AssetConfig.depositPaused` is documented as "set by guardian", while `pauseDeposits` is `onlyPauser` (DollarStore.sol:847) and `unpauseDeposits` is `onlyUnpauser` (:855). `IDollarStore.deposit`, the interface integrators compile against, is documented as "Hub-only in this milestone", returning "DLRS minted", and that notice is inherited into the implementation through `@inheritdoc`. In fact, spoke deposits return receipt shares and emit only `SpokeLiquidityAdded` (DollarStore.sol:584, 617), and `NotEnabled` is never thrown.

```
// src/DollarStore.sol:28-29
/// @notice Establishes the upgradeable core: ERC-7201 namespaced storage, two-step
///     governor/guardian roles, pause/unpause, and governor-gated UUPS upgrades.

// src/DollarStore.sol:54-57
///     a ceiling below what a listed feed needs bricks swaps and takes an upgrade to lift,
///     while a
///     generous one only matters if a governor action is wrong, which is timelocked and
///     cancellable
///     by the guardian.
uint256 private constant MAX_STALENESS = 3 days;

// src/storage/RegistryStorage.sol:41
```

I- Security NatSpec names the guardian as canceller, the governor as upgrade authority and the guardian as deposit pauser, contradicting the shipped role model CONTINUED

```
/// @notice Per-asset deposit pause (set by guardian in M5).

// src/interfaces/IDollarStore.sol:266-268
/// @notice Deposit `amount` (native units) of a hub asset and receive DLRS 1:1 in
normalized
/// units. Sub-unit dust stays with the caller. Hub-only in this milestone.
/// @return receiptUnits The normalized 6dp amount credited (DLRS minted).
```

These comments are what block explorers, integrators and incident responders read, and each one points at the wrong actor for a safety control:

- A reader of the staleness rationale assumes the guardian can veto a mistaken `setMaxStaleness`, when only the veto Safe or the proposing Safe can cancel it.
- A reader of the contract notice watches the governor timelock for upgrades instead of the upgrader timelock.
- An operator following the storage docs during an incident sends a per-asset pause to the guardian Safe, and the call reverts `OnlyPauser`.
- An integrator building against `IDollarStore` treats a spoke deposit's return value as DLRS units, when it is a share amount whose value diverges from units after any write-down.

— RECOMMENDATION

Rewrite these comments to match the deployed role model:

- The `MAX_STALENESS` rationale should name the veto Safe as the canceller.
- The contract notice should name the upgrader as upgrade authority and the pausers and unpauser as the pause roles.
- `depositPaused` should name the pauser and unpauser.
- `IDollarStore.deposit` should document both the hub path (DLRS minted) and the spoke path (receipt shares, `SpokeLiquidityAdded`).

Also remove the unused `NotEnabled` error and the "Milestone M1" headers on `IDollarStore` and `DollarStore`.

— RESOLUTION

Status: Resolved.

DollarStore Team: Fixed in 3f6e005, as recommended. The NatSpec names the veto Safe as canceller, the upgrader as upgrade authority, and the pausers and unpauser as the pause roles, and documents both deposit paths (DLRS minted on the hub; receipt shares and `SpokeLiquidityAdded` on a spoke). The `NotEnabled` error and the milestone labels are removed.

<https://github.com/coasify/dollar/pull/47>

INFORMATIONAL I-02

RESOLVED

Relisting a removed asset silently clears a pauser's deposit pause and the probe exemption, bypassing the unpauser

CATEGORY Access Control

LOCATION `src/libraries/SpokeLifecycleLib.sol:68-85 (removePool); src/DollarStore.sol:330-338 (addHubAsset); src/libraries/SpokeAdminLib.sol:69-77 (createSpoke); src/DollarStore.sol:820-860 (setProbeExempt, pauseDeposits, unpauseDeposits)`

REVIEW Main Review

DESCRIPTION

`removePool` unlists the spoke asset by flipping only `listed` to false (`SpokeLifecycleLib.sol:79`), so the asset's `depositPaused` and `probeExempt` flags survive. Once the asset is unlisted, `pauseDeposits`, `unpauseDeposits` and `setProbeExempt` all revert `AssetNotListed`, so no role can inspect or carry those flags across the relist. Relisting, which `PROTOCOL.md` "Relisting" and `SECURITY.md` "Fully impaired spoke recovery" describe as the recovery path, goes through `addHubAsset` or `createSpoke`. Both overwrite the whole `AssetConfig` with both flags set to false, and both emit only `AssetListed` or `PoolCreated`.

```
// src/libraries/SpokeLifecycleLib.sol:79 (removePool)
r.assetConfig[sa].listed = false;           // depositPaused and probeExempt are left as
  they were

// src/DollarStore.sol:330-338 (addHubAsset); src/libraries/SpokeAdminLib.sol:69-77
// (createSpoke) is identical
r.assetConfig[asset] = RegistryStorage.AssetConfig({
  poolId: 0,
  decimals: dec,
  scalingFactor: sf,
  priceFeed: priceFeed,
  listed: true,
  depositPaused: false,                    // a pauser's deposit pause is lifted here
  probeExempt: false                       // the governor's exemption is dropped here
});
```

Suppose a pauser paused deposits of an asset during an incident and the asset is later retired and relisted. Deposits and swaps offering it reopen the moment anyone executes the relist through the open governor executor, with no decision by the unpauser. That breaks the role split `SECURITY.md:83-106` and `:197` sets out ("Pausing and lifting a pause are separate roles", "The unpauser lifts the pause. No pauser can"), and the pauser can only re-pause after inflows have resumed. Dropping `probeExempt` brings back the zero-value issuer probe on exits. For a token that rejects zero-value transfers, which is the case the exemption exists for, exits that pay none of that asset revert `ExitBlockedByIssuer`.

I- 02 Relisting a removed asset silently clears a pauser's deposit pause and the probe exemption, bypassing the unpauser CONTINUED

(SpokeLifecycleLib.sol:295) until a new `setProbeExempt` executes. Because no `DepositsPausedSet(asset, false)` or `ProbeExemptSet(asset, false)` is emitted, monitors and frontends that rebuild state from events keep showing the asset as deposit-paused and probe-exempt after the relist.

— RECOMMENDATION

Preserve `depositPaused` across a relist: in `addHubAsset` and `createSpoke`, carry over the existing value instead of writing false, so only `unpauseDeposits` can lift a pause. Either carry `probeExempt` over the same way, or emit `ProbeExemptSet(asset, false)` when the relist resets it, and emit `DepositsPausedSet` whenever a listing changes that flag.

— RESOLUTION

Status: Resolved.

DollarStore Team: Fixed in cdf9380, as recommended. `addHubAsset` and `createSpoke` keep the stored `depositPaused` and `probeExempt` values when an asset is relisted, so only the unpauser can lift a deposit pause and the exemption survives. Since neither flag changes on a relist, no new event is needed for monitors to stay correct.

<https://github.com/coasify/dollar/pull/47>

INFORMATIONAL I-03

RESOLVED

A single dust queue position makes a dry via revert a spoke-to-spoke swap instead of being skipped

CATEGORY DoS

LOCATION src/libraries/SwapRouteLib.sol:204-212 (swapSpokeToSpoke via loop); src/libraries/SwapRouteLib.sol:515-517 (enqueueRemainder first-position minimum waiver)

REVIEW Main Review

DESCRIPTION

`swapSpokeToSpoke` walks the `via` list and sizes each via's spoke-to-hub leg (`got`) without regard to what the paired hub-to-spoke leg can pay. It skips a via only when `got == 0`. When `got > 0` and the second leg pays less, it reverts the whole call instead of moving on to the next via. The first leg matches the opposite queue before reserves, and `enqueueRemainder` waives the minimum order size for the first position in an empty line. So, while the offered asset's spoke has no reserve to fill it, anyone can plant a 1-unit position in the `via -> offerAsset` line, and that turns an otherwise dry via into one with `got = 1`.

```
// src/libraries/SwapRouteLib.sol:204-208 (swapSpokeToSpoke)
uint256 got = fillDirected(sell[i], offerAsset, via[i], remaining, true);
if (got == 0) continue; // only a fully dry first leg is
    skipped
uint256 paid = fillDirected(buy[i], via[i], wantAsset, got, true);
if (paid < got) revert IDollarStore.InsufficientLiquidity(amountUnits - remaining + paid,
    amountUnits);

// src/libraries/SwapRouteLib.sol:517 (enqueueRemainder)
if (remaining < minOrder && q.positionCount != 0) revert
    IDollarStore.OrderTooSmall(remaining, minOrder);
```

Take a swap of spoke asset S for spoke asset W routed through the launch hub assets `[USDC, USDS]`, at a moment when S's spoke holds no S reserve and no DLRS-side reserve above its minimum, W's spoke holds no W reserve, and the USDS route can complete the full amount through waiting USDS to S and W to USDS positions, so the USDC route fills nothing in either leg. Planting a 1-unit USDC to S position, which queues because no S reserve can fill it, makes the USDC leg take that unit. USDC cannot pay it out, so the call reverts `InsufficientLiquidity(0, amountUnits)`, and the USDS route that would have filled is never tried. Because the call reverts, the dust is never consumed. It stays in the line until it is filled or cancelled, so one planted unit breaks every call through that `via` order, not just

I-03 A single dust queue position makes a dry via revert a spoke-to-spoke swap instead of being skipped CONTINUED

one transaction. The attacker can cancel it at any time for a full refund, so the attack costs only gas.

The documented advice to simulate multi-via swaps (docs/PROTOCOL.md) does not protect against this. A simulation that passes is still broken by dust planted before the transaction executes, and routers or integrations with a fixed `via` order that do not re-simulate every call keep failing for as long as the dust stays. This conflicts with the project's own rule that "A single dust entry must not be able to strand a route" (docs/QUALITY.md §8). For affected callers, the call reverts and they pay the gas.

— RECOMMENDATION

Document in PROTOCOL.md and the `swapSpokeToSpoke` NatSpec that a via whose first leg fills but whose second leg falls short reverts the whole call instead of being skipped, so callers should put the most liquid via first and simulate immediately before sending.

PROOF OF CONCEPT <https://github.com/GuardianOrg/SentryRepo-DollarStore--Round-1-01a134086a3577dca5cc4eeaad9c621d/blob/refinery/poc-01a13408-7e90-75dc-8852-863b5f15106d/test/refinery-poc/dust-escrow-can-disable-dry-via-skipping-in-fixed-order-cros-ac612381.sol>

— RESOLUTION

Status: Resolved.

DollarStore Team: Fixed in 2d2e3d6, with a code change rather than the documentation recommended, because a planted dust position should not be able to strand a route. In `swapSpokeToSpoke`, each hub asset in via now carries at most what both of its legs can fill for the caller, using the same calculation as `getSwapQuote`. A hub asset that can carry nothing is skipped without settling anyone on its lines, and one that fails the inflow check is skipped instead of reverting the call. In the scenario from the finding, the dry hub asset is now skipped and the swap fills through the next one (`test_s2s_dustBuyerOnShortHubAsset_skippedForNext`).

Two limits remain and are documented in the NatSpec and PROTOCOL.md, with worked examples. Hub assets are used greedily in the caller's order, and they share the offered spoke's hub claim and the wanted spoke's reserve, so a list can revert where another order would fill; routers choose the order off-chain and simulate. Sizing assumes every payout succeeds, so a position waiting ahead whose owner is frozen counts against a hub asset until it is parked; `processQueue` parks it first.

Tests include a fuzz test against a brute-force best split of the amount across hub assets.

I - A single dust queue position makes a dry via revert a spoke-to-spoke
03 swap instead of being skipped

CONTINUED

<https://github.com/coasify/dollar/pull/47>

INFORMATIONAL I-01

Production runbook lists hub assets before activating the launch cap

CATEGORY	initialization-ordering
LOCATION	packages/dollar/docs/DEPLOY.md:L229-L288
REVIEW	Remediation Review

— DESCRIPTION

The production runbook executes initial hub listings before scheduling the hub launch cap. The deployed hub starts unpaused with cap zero, meaning unlimited, so following that sequence admits permissionless deposits throughout the subsequent cap timelock delay. A checklist asking operators to verify caps before their first deposit cannot prevent third-party deposits. Both submitted tests pass using the real deployment routine and timelock. This is a runbook/bootstrap sequencing defect, not a bypass of an already-active cap: the existing hub can alternatively be capped before listing.

Impact: A third party can establish exposure above the intended launch budget before the cap activates; the PoC deposits four times that budget. The later cap leaves existing exposure unchanged and blocks further deposits while exposure remains above it. No theft or underbacking is demonstrated.

— RECOMMENDATION

Configure the existing hub's cap before executing the first listing, or atomically execute cap configuration and listing in one timelock batch. Alternatively keep the protocol paused throughout bootstrap and unpaue only after configuration. Update the production runbook accordingly.

PROOF OF CONCEPT <https://github.com/GuardianOrg/SentryRepo-DollarStore--Round-1-01a134086a3577dca5cc4eeaad9c621d/blob/refinery/poc-01a16dc8-6fcc-7180-8441-9eaae4a3d299/test/refinery-poc/production-runbook-lists-hub-assets-before-activating-the-la-fcd69d91.sol>

— RESOLUTION

Status: Reported during the fix review period.

INFORMATIONAL I-02

Donations can invalidate atomic timelock recovery batches and force a fresh delay

CATEGORY denial-of-service

LOCATION packages/dollar/src/libraries/SpokeAdminLib.sol:syncReserves:L98-L103

REVIEW Remediation Review

— DESCRIPTION

syncReserves reverts when actual custody no longer has negative drift. An attacker can repair an affordable deficit by transferring tokens directly to DollarStore after governance schedules an atomic recovery batch containing that asset and other still-required actions. The repaired asset's sync now reverts, rolling back the entire batch, including reconciliation of unrelated larger losses. Exact deficit repair creates no rescuable excess. The batch remains Ready, not consumed, but cannot execute while that sync has no negative drift. Removing the repaired call requires a newly scheduled payload and its configured delay; changing only the salt does not fix the stale payload. Both revised PoC tests pass, including rollback, operation-state assertions, replacement execution failing immediately before maturity and succeeding exactly at maturity, and a no-donation control.

Impact: A 1-unit donation blocks a particular batch reconciling a separate 400-unit loss and delays that work by another governor delay: seven days in the demonstrated reference configuration. Other bundled actions are also delayed. This requires a batch containing an affordable-to-repair deficit and still-needed work, not malicious governance. It does not establish a protocol-wide deadlock, cheap indefinite repetition, or additional principal loss. Invalidating a reduced replacement requires repairing its remaining deficit.

— RECOMMENDATION

Make the no-negative-drift case an idempotent no-op while retaining escrow-impairment checks. Alternatively, schedule independent reconciliation operations so a repaired asset cannot block unrelated recovery work.

PROOF OF CONCEPT <https://github.com/GuardianOrg/SentryRepo-DollarStore--Round-1-01a134086a3577dca5cc4eeaad9c621d/blob/refinery/poc-01a16dc8-6fcc-7180-8441-9eaae4a3d299/test/refinery-poc/donations-can-invalidate-atomic-timelock-recovery-batches-an-52bd32a8.sol>

— RESOLUTION

Status: Reported during the fix review period.

INFORMATIONAL I-03

swapSpokeToSpoke reports a hub pause or an unhealthy via as InsufficientLiquidity

CATEGORY Unexpected Behavior

LOCATION `src/libraries/SwapRouteLib.sol:198` (unhealthy via skipped); `src/libraries/SwapRouteLib.sol:219-224` (shortfall revert); `src/libraries/PegLib.sol:73-77` (canInflow)

REVIEW Remediation Review

— DESCRIPTION

Since 2d2e3d6, `swapSpokeToSpoke` skips a hub asset in `via` that fails `PegLib.canInflow`: the asset is deposit-paused, the hub pool is paused, or its feed is missing, stale or off-peg.

```
// src/libraries/SwapRouteLib.sol:198
if (!PegLib.canInflow(via[i])) continue;
```

When the skipped entries leave the amount unfilled, the call ends in the generic shortfall revert (:219-224). A single-via call, or one where every via is unhealthy, therefore reverts `InsufficientLiquidity(wantAsset, 0, fullOutput)` instead of naming the pause or price failure. Before 2d2e3d6, the route loop called `PegLib.checkInflow(via[i])` (`SwapRouteLib.sol:194` at f78ae37), which reverted with the specific error (`DepositsPaused`, `PoolPaused(0)`, `PriceStale`, ...).

The docs describe these cases as pauses and price failures with their own errors:

- SECURITY.md:210 and :212 list `swapSpokeToSpoke` among the calls an asset pause or a hub pool pause blocks.
- docs/PROTOCOL.md:274-275 map price failures and pauses to their own errors.
- docs/PROTOCOL.md:267 tells integrators to answer `InsufficientLiquidity` by swapping a smaller amount or trying another `via` order.

No funds are at risk, and the swap is correctly refused. During a pause or oracle incident, though, routers retry smaller amounts, which fail the same way, or reorder `via`. Monitoring also records the pause as a liquidity shortage.

Found during the fuzzing campaign, while classifying expected reverts for the spoke-to-spoke handler.

— RECOMMENDATION

I-03 swapSpokeToSpoke reports a hub pause or an unhealthy via as InsufficientLiquidity

CONTINUED

Keep skipping unhealthy entries. When the call ends short and an entry was skipped for failing the inflow check, revert with that cause: for example, call `PegLib.checkInflow` on the first skipped via before the shortfall revert, or revert a dedicated error such as `NoHealthyVia(via)`.

Alternatively, document at docs/PROTOCOL.md:267 that `InsufficientLiquidity` with `fillableAmountOut == 0` can mean every via was skipped because of a pause or a price failure, and state the same at SECURITY.md:210 and :212.

— RESOLUTION

Status: Reported during the fix review period.

INFORMATIONAL I-04

The integrator error table names the wrong error for a hub exit above the caller's balance and omits the deposit errors

CATEGORY Documentation

LOCATION `src/libraries/SpokeLifecycleLib.sol:186` (hub exit burns DLRS, reverts `ERC20InsufficientBalance`); `src/libraries/SpokeLifecycleLib.sol:190` (spoke exit reverts `InsufficientReceiptShares`); `src/DollarStore.sol:519, :523, :539, :813, :995` (`AssetNotListed`, `WrongPool`, `FeeOnTransferNotSupported`, `LaunchCapExceeded`)

REVIEW Remediation Review

DESCRIPTION

`ae25d5a` adds an error table for integrators. Its row for `InsufficientReceiptShares` reads "More DLRS or shares than the caller holds". Only a spoke exit reverts that way (`SpokeLifecycleLib.sol:190`). A hub exit burns the DLRS first, so a `receiptAmount` above the caller's balance reverts with the token's

`ERC20InsufficientBalance(caller, balance, needed)`:

```
// src/libraries/SpokeLifecycleLib.sol:184-190
if (poolId == 0) {
    DLRS(CoreStorage.layout().dlrs).burn(msg.sender, receiptAmount); //
    ERC20InsufficientBalance
} else {
    ...
    if (owned < receiptAmount) revert IDollarStore.InsufficientReceiptShares(receiptAmount,
        owned);
}
```

`previewWithdraw(0, ...)` does revert `InsufficientReceiptShares`, but against the total supply, not the caller's balance (:64). An integrator that decodes `InsufficientReceiptShares` to show "withdraw less" never sees it on a hub exit, and the table's last row, "An error from the token itself", describes a failed transfer, not a burn.

The table also omits the errors a deposit can revert with. `docs/PROTOCOL.md:51` lists the conditions in prose, but the errors are not named anywhere in the document: `AssetNotListed` (`DollarStore.sol:519`), `WrongPool` (:523, :539), `FeeOnTransferNotSupported` (:813) and `LaunchCapExceeded` (:995). A spoke at its launch cap is the one an integrator meets in normal operation.

Integrators building error handling from the table decode the wrong error on hub exits and have no entry for a deposit rejected by the launch cap.

I-04 The integrator error table names the wrong error for a hub exit above the caller's balance and omits the deposit errors **CONTINUED**

— RECOMMENDATION

Either check the caller's DLRS balance in the hub exit and revert

`InsufficientReceiptShares(receiptAmount, balance)` before the burn, so the table holds as written, or change the row to say a hub exit reverts with the DLRS token's `ERC20InsufficientBalance`. Add rows for `AssetNotListed`, `WrongPool`, `FeeOnTransferNotSupported` and `LaunchCapExceeded(poolId, attempted, cap)`.

— RESOLUTION

Status: Reported during the fix review period.

§ 05

Disclaimer.

This report is not, nor should be considered, an "endorsement" or "disapproval" of any particular project or team. This report is not, nor should be considered, an indication of the economics or value of any "product" or "asset" created by any team or project that contracts Guardian to perform a security assessment. This report does not provide any warranty or guarantee regarding the absolute bug-free nature of the technology analyzed, nor do they provide any indication of the technologies proprietors, business, business model or legal compliance.

This report should not be used in any way to make decisions around investment or involvement with any particular project. This report in no way provides investment advice, nor should be leveraged as investment advice of any sort. This report represents an extensive assessing process intending to help our customers increase the quality of their code while reducing the high level of risk presented by cryptographic tokens and blockchain technology.

Blockchain technology and cryptographic assets present a high level of ongoing risk. Guardian's position is that each company and individual are responsible for their own due diligence and continuous security. Guardian's goal is to help reduce the attack vectors and the high level of variance associated with utilizing new and consistently changing technologies, and in no way claims any guarantee of security or functionality of the technology we agree to analyze.

The assessment services provided by Guardian is subject to dependencies and under continuing development. You agree that your access and/or use, including but not limited to any services, reports, and materials, will be at your sole risk on an as-is, where-is, and as-available basis.

Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty. The assessment reports could include false positives, false negatives, and other unpredictable results. The services may access, and depend upon, multiple layers of third-parties.

Notice that smart contracts deployed on the blockchain are not resistant from internal/external exploit. Notice that active smart contract owner privileges constitute an elevated impact to any smart contract's safety and security. Therefore, Guardian does not guarantee the explicit security of the audited smart contract, regardless of the verdict.

§ 06

About Guardian.

Guardian is a smart contract security firm. Every engagement is worked by two independent teams competing on the same scope, each with its own tooling and methods, so no single reviewer's blind spot survives.

\$45B+DIGITAL ASSETS UNDER
REVIEW**200+**

ENGAGEMENTS COMPLETED

500+CRITICAL & HIGH FINDINGS
REPORTED

Reach us at audits@guardianaudits.com.