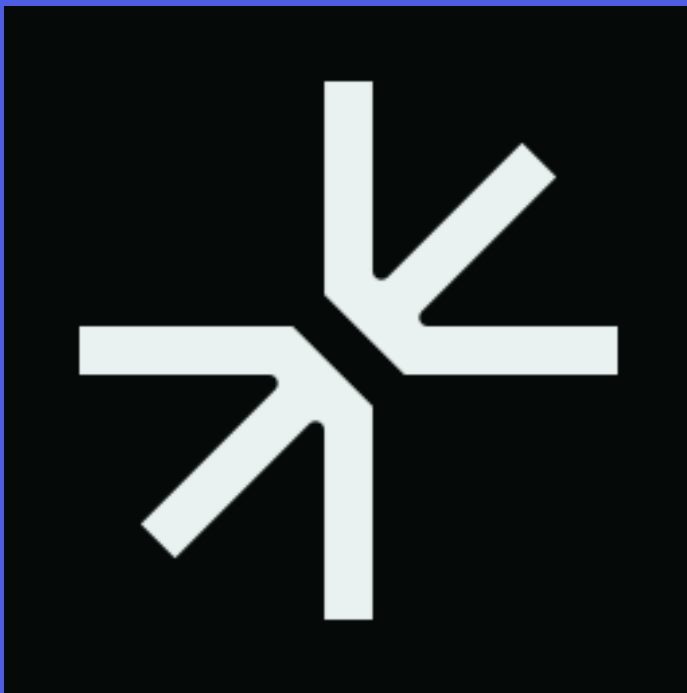


Axis Coordinate V2 Contracts Audit



July 15, 2026

This document has been prepared solely for the named client and it may not be relied upon by any third parties. The content in this document is provided for informational purposes only and does not constitute professional advice of any kind. Security assessments are limited by factors such as scope, time, and techniques used and therefore this document does not constitute a certification, guarantee, or warranty regarding the proper functioning of any system or the absence of security vulnerabilities.

Copyright © 2026 Zeppelin Group Ltd.

Table of Contents

Table of Contents	2
Summary	4
Scope	5
System Overview	6
Security Model and Trust Assumptions	7
Privileged Roles	8
Trust Assumptions	9
High Severity	12
H-01 _claimIndex Can Skip PENDING Requests, Permanently Stranding Redemptions	12
H-02 pauseCapacity Disables Cap Enforcement Instead of Halting Settlement Flow	13
Medium Severity	14
M-01 Reward Configuration Changes Do Not Affect the Active Vesting Schedule	14
M-02 Repeated Reward Funding Re-Amortizes Unvested Rewards And Delays Vesting Indefinitely	15
M-03 Pending And Claimable Redemption Value Of Fully-Restricted Accounts Cannot Be Redistributed	16
M-04 Batch Redemption Servicing Can Stall on an Ineligible Request and Delay Later Eligible Redemptions	17
M-05 Unbounded Request Lists Can DoS withdraw and redeem	18
M-06 Unauthenticated accountId Undermines ACCOUNT-scope Caps and Nonce Namespaces	19
M-07 Sparse Code Comments and Incomplete Branch Coverage	20
Low Severity	21
L-01 Redundant whenNotPaused Modifier in USDx	21
L-02 approve and permit Remain Functional While USDx Is Paused	22
L-03 Deposits and Mints Lack a Pause Control	22
L-04 previewWithdraw and previewRedeem Do Not Revert	23
L-05 ERC-4626 max* Views Do Not Return Zero When Actions Are Disabled	23
L-06 Blacklisted Spenders Can Still Move Funds Through Existing Allowances	24
L-07 Soft-Restricted Owners Can Redeem Through an Unrestricted Controller	25
L-08 Restricted Accounts Can Deposit And Mint Shares To Unrestricted Receivers	26
L-09 withdraw Can Revert for Asset Amounts Within maxWithdraw Due to Ceil Rounding	27
L-10 Incorrect ERC-165 Integration in StakedUSDx	28
L-11 Rewards Funded While No Shares Exist Are Captured by the First Depositor	29
L-12 USDx Blacklist Does Not Freeze Market Activity	29
L-13 Documented Per-User Block Limits Are Unimplemented Stubs	31
Notes & Additional Information	32
N-01 Inconsistent Naming in the StakedUSDx Asynchronous Redemption Flow	32

N-02 Per-Block Caps Key on Block Number, Whose Wall-Clock Meaning Differs Across Deployment Chains	33
N-03 Reason Codes Use Inconsistent Encodings Across the Freeze Surface Taxonomy	34
Conclusion	35
Appendix	36
Issue Classification	36

Summary

Type	DeFi	Total Issues	25 (24 resolved, 1 partially resolved)
Timeline	From 2026-07-03 To 2026-07-07	Critical Severity Issues	0 (0 resolved)
Languages	Solidity	High Severity Issues	2 (2 resolved)
		Medium Severity Issues	7 (6 resolved, 1 partially resolved)
		Low Severity Issues	13 (13 resolved)
		Notes & Additional Information	3 (3 resolved)
		Client Reported Issues	0 (0 resolved)

Scope

OpenZeppelin performed an audit of the [Axis-Coordinate/contracts](#) repository at commit [6db59ec](#).

In scope were the following files:

```
src
├── v2
│   ├── market
│   │   ├── MarketConfig.sol
│   │   └── USDxMarket.sol
│   ├── token
│   │   └── USDx.sol
│   ├── types
│   │   └── AxisV2Ids.sol
│   └── StakedUSDx.sol
```

System Overview

Axis Protocol V2 is a stablecoin issuance and staking system targeted at Ethereum mainnet. It is built around [USDx](#), a collateral-backed ERC-20 stablecoin, and [sUSDx](#), a staking vault that lets USDx holders earn protocol rewards. The scope under review covers the token itself, the primary market that mints and redeems USDx against approved collateral, the market configuration store that holds routing and risk parameters, the staking vault, and a shared library of protocol identifiers. All four contracts are upgradeable and use role-based access control from the OpenZeppelin upgradeable libraries, with storage gaps reserved for future variables.

Primary market. Minting and redemption run through [USDxMarket](#) together with its configuration contract [MarketConfig](#). Users do not transact directly. Instead, an order originator signs an EIP-712 order off chain, and a privileged operator submits it on chain through the settle, mint, or redeem entry points. On a mint, the market pulls the signed input collateral from the signer and forwards it to the custodian destination configured for the order's channel, then mints USDx to the receiver. On a redeem, the market burns USDx from the signer and transfers the output collateral to the receiver from the market's own balance. The amounts moved are fixed by the signed order, and the output amount is taken directly from the order's minimum output field, so there is no on-chain price oracle. [MarketConfig](#) stores the routes (asset pairs and side), the channels (custodian and destination), the signer whitelist, the supported assets, per-block mint and redeem limits, and a hierarchy of capacity caps at the global, per-asset, per-route, per-account, and per-channel levels evaluated over configurable windows. Capacity accounting is mutated only during settlement and is restricted to the settlement manager.

Order validation and delegated signing. Before settlement, the market checks the order signature, a per-account nonce tracked in a bitmap, the order deadline, the signer whitelist, the channel and route configuration, and both block-level and windowed capacity. Signatures are validated as EIP-712 and support contract signers through ERC-1271, as well as a two-step delegated signer mechanism in which an account first nominates a signer and the signer then confirms the relationship. Settlement records a proof identifier and sets a settled flag that guards against double settlement, and it emits the settlement receipt as an event rather than persisting the full receipt in storage.

Staking vault and asynchronous redemption. `StakedUSDx` is an ERC-4626 vault whose asset is USDx and whose shares are sUSDx. Deposits mint shares immediately, but redemptions follow an asynchronous request lifecycle modeled on ERC-7540. A holder first requests a redemption, which burns the shares and opens a request subject to a per-policy cooldown. Once the cooldown elapses, an account holding the redemption servicer role makes requests claimable, after which the controller or an approved operator claims the assets through the withdraw or redeem entry points. A request can be cancelled while it is still pending, which restores the burned shares. Rewards are funded by a reward manager and vest linearly over a configurable duration, and only vested rewards count toward reported total assets. Assets reserved for pending and claimable redemptions are tracked as liabilities so the vault can compare its asset balance against its obligations. The vault also supports soft and full account restrictions, where a full restriction can freeze a balance and allow an administrator to redistribute it. The `AxisV2Ids` library defines a shared taxonomy of surface, action, reason, and role identifiers used across the protocol, and confers no authority by itself.

Security Model and Trust Assumptions

Axis V2 spans the primary market (`USDxMarket`, `MarketConfig`), the `USDx` stablecoin, and the `StakedUSDx` vault. On-chain the contracts enforce order validity, replay protection, routing, whitelisting, and capacity, while pricing, counterparty policy, custody, and reserve funding are trusted off-chain responsibilities of governance, operators, and custodians. Security therefore rests on the honesty and key security of a small set of privileged roles and on the correctness of the off-chain layers.

Every mint or redeem is an EIP-712 signed `Order` authenticated to `order.signer` (an EOA, an ERC-1271 contract, or a source-accepted delegate) and can only be submitted by an operator role. Replay is blocked by a per-account nonce bitmap and a per-order settled/cancelled record, and the domain separator binds the chain id and contract address. `MarketConfig` gates each order on an enabled route and channel (matching side and assets), an asset and custodian allowlist, and a whitelist that both signer and receiver must satisfy; throughput is bounded by a multi-scope capacity engine (GLOBAL, ASSET, ROUTE, ACCOUNT, and CHANNEL caps over lifetime, per-block, or fixed windows) and a per-block mint/redeem limiter. All bookkeeping is finalized before any token movement, so a failed transfer, a paused token, or a blacklisted party reverts the whole settlement atomically.

USDx is an ERC-20 stablecoin with pause, blacklist, and bridge controls whose minting is authorized only through the market. **StakedUSDx** is a vault with a cooldown, a request/service/claim redemption flow, and linearly vesting rewards. Pricing (**minOutputAmount**, with no on-chain oracle) and signer-to-account policy are set off-chain, minted collateral leaves to custodians, and redemptions are paid from the market's own balance, so USDx solvency depends on off-chain custody. All in-scope contracts are upgradeable behind proxies and initialize through initializers rather than constructors.

Privileged Roles

- **DEFAULT_ADMIN_ROLE** (governance). Manages every other role and all security-critical configuration: market routes, channels, custodians, whitelist, capacity caps, per-block limits, and mint/redeem enable flags; and the vault's cooldown, redemption policy, reward configuration, maximum reward rate, and asset rescue. Trusted not to configure routes, channels, or policies that misdirect funds and not to abuse role management (the standard grant/revoke model lets it assign any role, including to itself). The deployment collapses this authority and the proxy-upgrade authority into one non-timelocked key that defaults to the deployer EOA.
- **MINT_OPERATOR_ROLE** / **REDEEM_OPERATOR_ROLE**. The sole submitters of signed orders, paying gas for users. They cannot alter an order's terms (settlement is signature-bound) or exceed the signed amounts and configured block and capacity limits, so they price orders soundly off-chain and are trusted for liveness and neutrality, since a withholding or reordering operator can censor, delay, or influence ordering relative to capacity.
- **SETTLEMENT_MANAGER_ROLE**. Cancels unsettled orders on **USDxMarket** and consumes capacity on **MarketConfig**; the market contract itself must hold it on **MarketConfig** for capacity accounting. Trusted not to grief signers, since it can cancel any order before it settles.
- **EMERGENCY_ROLE**. On **MarketConfig**, pauses individual capacity caps and disables minting and redemption, but cannot re-enable them (an administrator action). Trusted to act only in genuine emergencies, with powers limited to reducing availability rather than moving funds.
- **REDEMPTION_SERVICER_ROLE**. Advances eligible vault redemption requests to a claimable state; batch servicing scans a bounded number per call, so a large backlog may need several calls. Users depend on it for liveness, since assets cannot be claimed until serviced.

- **REWARD_MANAGER_ROLE**. Funds vault rewards, which vest linearly; funding is bounded by the reward duration and the administrator-set maximum reward rate, limiting how fast value enters the exchange rate.
- **RESTRICTION_ROLE**. Maintains the **USDx** blacklist and the vault's soft and full account restrictions, and can redistribute a fully restricted vault account's balance to another account. This freezing and confiscation power over user balances is a significant trust point; it is trusted to act honestly and only to reduce risk.
- **PAUSER_ROLE**. Halts **USDx** transfers and independently pauses the vault's redemption-request, claim, servicing, and cancellation phases and reward funding; trusted to act only to reduce risk.
- **BRIDGE_ROLE (USDx)**. Mints and burns **USDx** for cross-chain movement, so it can create **USDx** not backed by primary-market collateral; a high-trust role that must be operated so mints and burns mirror consistently across chains.
- **MARKET_ROLE (USDx)**. Authorizes **USDx** minting and is trusted to be held only by the market proxy; a grant to any other account would allow minting outside every cap, whitelist, and route check.
- **Order signers and delegates**. Whitelisted signers are trusted to sign only intended orders (a signer controls every signed field). A whitelisted source that accepts a delegated signer is trusted to accept only intended delegates; an accepted delegate need not be whitelisted, and its key compromise allows signing that source's orders.
- **Upgrade authority (ProxyAdmin owner)**. Can replace any reviewed logic through the governance-owned **ProxyAdmin**; a critical trust boundary assumed to be a timelock or multisig distinct from the deployer, with upgrade power separated from operational admin.

Trust Assumptions

Off-Chain Processes

- Order pricing and RFQ/quote generation happen off-chain, and the operator layer is trusted to enforce signer and account policy and encode sound terms before relaying (there is no on-chain oracle or valuation).
- Minted collateral leaves to off-chain custodians and redemptions are paid from the market's own balance, so **USDx** solvency depends on custodians honoring the backing and replenishing the market before redemptions settle; there is no on-chain reserve or solvency accounting (caps bound only **USDx** notional), and balance monitoring runs off-chain.

- Terms acceptance is off-chain; `termsHash` is an evidentiary commitment that is never checked on-chain.
- Post-deploy role wiring is an off-chain governance step (the deploy script grants only `DEFAULT_ADMIN_ROLE`): the market must receive `MARKET_ROLE` on `USDx` and `SETTLEMENT_MANAGER_ROLE` on `MarketConfig`, and a misconfiguration can break minting or leave capacity unenforced.

External Data Integrity

- Collateral tokens are standard ERC-20, with no fee-on-transfer, rebasing, or ERC-777 hooks, since neither transfer leg checks a post-transfer balance delta.
- The operator-signed `minOutputAmount` is the single largest data dependency, with no on-chain oracle, slippage, or ratio check, and collateral decimals are assumed correctly encoded in that price.
- EIP-712 signatures are the sole authentication of intent; validity requires the signer's key, an ERC-1271 approval, or a source-accepted delegate.

Secure Runtime

- Private keys for every privileged role are assumed secure; compromise of the governance or upgrade key is total loss.
- Governance is a timelock or multisig distinct from the deployer EOA, set explicitly at deployment, with post-deploy assertions confirming role holders and `ProxyAdmin` ownership, and upgrades respect the 50-slot `__gap` storage budget.
- Governance configures caps and routes soundly: each route's `USDx` side must map to the deployed `USDx` token (not enforced on-chain, since `_executeMovement` always mints or burns `USDx`), `channel.destination` must point to the intended custodian, and a `WindowType.FIXED` cap must not be enabled with `windowSize == 0` (which would revert every capacity check until reverted).

Scope and Design Notes

- Custodian solvency and physical custody are out of scope; the custodian is the reserve guarantor and the market the execution and evidence layer.
- Off-chain operator policy, pricing, and order/account assignment correctness are out of scope, as are terms legality and on-chain terms enforcement (`ITermsAcceptance` is unbuilt).
- Reentrancy from non-standard tokens is out of scope (standard ERC-20 assumed, with no reentrancy guard).
- `SettlementMode.ADAPTER` / `CROSS_CHAIN` and the `adapter` / `destinationDomain` fields are emit-only scaffolding, never branched on and not

relied upon for routing; for a REDEEM, `OrderSettled.destination` is neither payer nor recipient, so indexers reconstruct the flow from the `Redeem` event.

- The ASSET-scope cap bounds USDx notional in one direction only (not physical collateral flow), and `WindowType.FIXED` caps are fixed-bucket rather than rolling (up to twice the cap across a boundary).
- The vault's rescue function withdraws only a surplus of the underlying asset and cannot pull the share token, but there is no equivalent surplus limit on unrelated tokens.

High Severity

H-01 `_claimIndex` Can Skip `PENDING` Requests, Permanently Stranding Redemptions

`StakedUSDx` implements an asynchronous redemption flow in which `requestRedeem` burns the owner's shares, appends the new request identifier to `_controllerRequests[controller]`, and records a `RedemptionState` with `status` set to `PENDING`. A servicer later transitions individual requests to `CLAIMABLE` through `_servicePendingRedeemRequest`, and controllers claim the proceeds through `withdraw` and `redeem`, which delegate to `_claimRedeemByAssets` and `_claimRedeemByShares`. These claim loops scan the controller's request list starting from the persisted cursor `_claimIndex[controller]`.

Both claim loops increment the cursor for any request that is not currently `CLAIMABLE`, including requests that are still `PENDING`, and then persist the advanced cursor to `_claimIndex[controller]`. Because claimability is not required to follow request order, this cursor can be advanced past a request that has not yet been serviced. The servicer can make an arbitrary eligible request `CLAIMABLE` in isolation through `_serviceRedeemRequest`, so a later request can become claimable while an earlier request for the same controller remains `PENDING`. When the controller claims the later request, the loop steps over the earlier `PENDING` request to reach it and stores the advanced cursor, leaving the earlier request positioned below `_claimIndex[controller]`.

Once the skipped request is subsequently serviced to `CLAIMABLE`, it sits at an index below `_claimIndex[controller]` and is never revisited by either claim loop. The shares for that request were already burned at request time and its assets remain reserved in the claimable aggregates, yet there is no claim-by-`requestId` path to reach them and `cancelRedeemRequest` only operates while a request is `PENDING`. As a result, the reserved assets become permanently unclaimable, and the controller's `claimableAssets` aggregate is inflated by an amount that can never be withdrawn.

Consider enforcing strict per-controller FIFO servicing so that a later request cannot become `CLAIMABLE` while an earlier request remains `PENDING`. Consider additionally providing a claim-by-`requestId` recovery path, with appropriate authorization, so that individual claimable requests remain reachable even when the cursor has advanced beyond them.

Update: Resolved in [pull request #88](#) at commit [308435d](#). The team stated:

The proposed change restores `_claimIndex` to a request when that request is serviced after the controller cursor has already advanced past it, keeping the request reachable by both claim paths.

H-02 `pauseCapacity` Disables Cap Enforcement Instead of Halting Settlement Flow

The capacity engine in `MarketConfig` treats a cap whose `enabled` flag is false as "no cap configured" rather than as a hard stop. In `_applyCap` a disabled cap returns the OK code and contributes no restriction, and when no scope is enabled `_checkCapacity` leaves the effective available amount at `type(uint256).max`. Consumption mirrors this:

`_consumeCap` accrues nothing while a cap is disabled. As a result, `enabled == true` means the cap is enforced and `enabled == false` means unlimited flow for that scope.

The `pauseCapacity` function is callable by both `DEFAULT_ADMIN_ROLE` and `EMERGENCY_ROLE`, and it does exactly one thing: it sets `config.enabled` to false. Pausing an enabled protective cap therefore raises the effective available capacity for that scope to unlimited (bounded only by any other still-enabled cap and by the separate per-block limiter, which can itself default to the maximum value). The control that operators would reach for during an incident does the opposite of halting flow. The correct way to stop a scope is the inverse: keep the cap enabled and set its `cap` to zero, which makes `_applyCap` compute an available amount of zero and reject any positive requested amount. The asymmetry compounds the problem, because `unpauseCapacity` is restricted to `DEFAULT_ADMIN_ROLE`. An emergency actor who "paused" (removed) a cap cannot restore it, which contradicts the documented invariant that the emergency role cannot increase risk.

A pause also hides usage accrued within a window, which amplifies the impact. While a cap is paused, `_consumeCap` records nothing, so re-enabling it restores the stale pre-pause counter. A pause, mint, unpause sequence can therefore push a windowed cap past its configured limit even after the cap is restored, because the usage accumulated during the paused interval is never reflected in the bucket.

Consider making a pause a blocking state rather than a disable. One option is to add a distinct paused flag to the cap configuration that the check and consume paths treat as an available amount of zero while leaving `enabled` and `cap` intact, so an unpause restores the original limit. An alternative is to have `pauseCapacity` set `cap` to zero while keeping the cap enabled and restore the prior value on unpause. Either way, a paused scope should yield the

cap-exceeded code for any positive requested amount, and a test should assert that settlement reverts with that reason after `pauseCapacity`.

Update: Resolved in [pull request #89](#) at commit [189be52](#). The change adds a distinct paused flag for each cap key, preserves the configured cap, and reports zero available capacity while paused. Notably, a zero-amount request against a paused cap [returns](#) `CHECK_OK` rather than the exceeded code.

Medium Severity

M-01 Reward Configuration Changes Do Not Affect the Active Vesting Schedule

`StakedUSDx` accrues rewards by vesting `_unvestedRewards` into `totalAssets` at `_rewardRate` until `_periodFinish`, both of which are set exclusively in `fundRewards`. The `_rewardPaused` flag is read only by the `fundRewards` guard, so `pauseRewardFunding` and `setRewardConfig` with `paused` set to `true` block new funding but do not stop already-funded rewards from continuing to vest into the exchange rate. Similarly, `setRewardConfig` updates `_rewardDuration` and `_maxRewardRate` without recomputing `_rewardRate` or `_periodFinish`, so a mid-period change to the duration or maximum rate does not re-scale the active schedule and takes effect only on the next `fundRewards` call. Neither behavior causes loss or insolvency, since `_vestedRewards` remains capped at `_unvestedRewards` and the corresponding assets are already held by the contract, but both may diverge from an operator's expectation that pausing or reconfiguring rewards halts or reshapes ongoing accrual.

Consider documenting that these controls govern only future funding cycles, or, if halting or re-scaling the active schedule is intended, updating `_rewardRate` and `_periodFinish` after checkpointing in `pauseRewardFunding` and `setRewardConfig` (for example, zeroing the rate to freeze accrual, or recomputing it as `_unvestedRewards / duration` with `_periodFinish` reset to reshape it).

Update: Resolved in [pull request #90](#) at commit [e7cec0e](#). The team stated:

The proposed change checkpoints the current reward state and reschedules active rewards whenever funding is paused or reward configuration changes. Pausing freezes

remaining rewards, while unpaused duration or rate changes recompute the active schedule.

M-02 Repeated Reward Funding Re-Amortizes Unvested Rewards And Delays Vesting Indefinitely

`StakedUSDx` distributes funded rewards to share holders by vesting them linearly over time. Each call to `fundRewards` sets a `_rewardRate` and a `_periodFinish`, and `_vestedRewards` accrues rewards at that rate until `_periodFinish` is reached. As time passes, `_checkpointRewards` moves the vested portion from `_unvestedRewards` into `_accountedAssets`, which increases `totalAssets` and therefore the exchange rate at which holders redeem. This mechanism is intended to release a funded reward amount to holders gradually over a single `_rewardDuration`.

The `fundRewards` function may be called again before the current schedule completes. When it is, it first checkpoints the already vested portion, then recomputes `nextRewardRate` as $(_unvestedRewards + assets) / _rewardDuration$ and unconditionally sets `_periodFinish` to `block.timestamp + _rewardDuration`, without regard for the time remaining in the previous schedule. As a result, every top-up re-amortizes all currently unvested rewards, including those funded earlier, over a fresh full duration and pushes the completion time forward. A reward manager can defer distribution indefinitely by repeatedly funding small, non-zero amounts.

Consider exposing distinct operations for extending the duration and for adding rewards to the current window, so that the resulting timeline is explicit and cannot be pushed forward indefinitely through repeated dust-sized top-ups.

Update: Resolved in [pull request #91](#) at commit [9bee3dd](#). The team stated:

The proposed change makes reward top-ups use the remaining active vesting duration and retain the existing period finish, using the full configured duration only for a new schedule.

M-03 Pending And Claimable Redemption Value Of Fully-Restricted Accounts Cannot Be Redistributed

The vault exposes `redistributeLockedAmount` so that the `RESTRICTION_ROLE` can move the holdings of a `FULL_RESTRICTION` account to a recovery address. This function operates only on the account's live share balance: it reads `balanceOf(from)` and reassigns those shares through `super._update`. Independently, the asynchronous redemption flow removes shares from the live balance the moment a request is created. `requestRedeem` burns the owner's shares and moves the corresponding value into the pending redemption accounting, after which servicing migrates it into the claimable accounting.

As a consequence, any value an account has already committed to a redemption request is no longer part of `balanceOf` and is therefore not reachable by `redistributeLockedAmount`. Once such an account is placed under `FULL_RESTRICTION`, this pending or claimable value cannot be moved or recovered through any available path:

- `redistributeLockedAmount` transfers only `balanceOf(from)`, so it recovers nothing from the redemption pipeline.
- The owner cannot claim the value, because `withdraw` and `redeem` call `_requireClaimAllowed`, which reverts when `_fullRestriction[owner]` is set.
- The owner cannot cancel the request, because `cancelRedeemRequest` reverts when `_fullRestriction[controller]` or `_fullRestriction[state.owner]` is set, and only `PENDING` requests are cancellable.
- An administrator cannot rescue the backing assets, because `rescueTokens` limits the underlying asset to `_surplusAssets`, and `_requiredAssets` includes `_pendingRedeemAssets` and `_claimableRedeemAssets`, so the reserved backing is never counted as surplus.

The pending and claimable portion of a fully-restricted account is therefore permanently frozen. It can be neither seized by the protocol nor withdrawn by the account, and the reserved assets remain locked in the vault liabilities indefinitely. This is asymmetric with the intended enforcement model, in which `FULL_RESTRICTION` is expected to allow the holdings of a restricted account to be redistributed to a recovery address. An account can also reach this state deliberately by submitting a redemption request before a restriction is applied, placing that value beyond the reach of `redistributeLockedAmount`.

Consider extending the redistribution mechanism in a new function so that the `RESTRICTION_ROLE` can also reassign the pending and claimable redemption positions of a fully-restricted account to a recovery address.

Update: Resolved in [pull request #114](#) at commit [7b54d8d](#). The team stated:

The proposed change adds a restriction-role `redistributeRedeemRequest` path that reassigns a pending or claimable request to an unrestricted recovery controller while moving the corresponding aggregate accounting and retaining reserved assets as vault liabilities.

M-04 Batch Redemption Servicing Can Stall on an Ineligible Request and Delay Later Eligible Redemptions

The `StakedUSDx` redemption flow supports batched servicing through `serviceRedemptions`, which scans request IDs starting at `_nextServiceRequestId` and marks eligible `PENDING` requests as `CLAIMABLE`. Each request records an `eligibleAt` timestamp in `requestRedeem` computed as `block.timestamp + policy.cooldown`, where the redemption policy is controller-specific and its cooldown can differ across controllers up to a `MAX_COOLDOWN_DURATION` of 90 days.

During a scan, `serviceRedemptions` caches the first `PENDING` request it encounters in `nextPendingRequestId` and, when that request is not yet eligible, skips it while retaining it as the value later `persisted as the next cursor`. The loop is bounded to a `MAX_SERVICE_SCAN_REQUESTS` of 256 requests scanned per call, and this budget is consumed by every request in the window, including already-serviced `CLAIMABLE` requests that are skipped. Consequently, the cursor is reset to the same ineligible request on every call, and each subsequent invocation re-examines the same window of at most 256 request IDs without advancing beyond it.

Consider an early request created under a controller with a long cooldown, followed by a large number of later requests created under a controller with a short or zero cooldown. The first servicing call marks the eligible requests inside the initial window as `CLAIMABLE`, but the cursor remains fixed at the early ineligible request. Every later call rescans that same window and never reaches the request IDs beyond it, so the later eligible redemptions are not serviced through `serviceRedemptions` until the blocking request becomes eligible or is cancelled, which can take up to 90 days. This degrades redemption liveness for the later requests and forces operational workarounds such as servicing individual requests through `serviceRedeemRequest`.

Consider updating the cursor logic so that each batch advances past `PENDING` requests that are not yet eligible, allowing the scan to reach later request IDs while the skipped requests are

serviced afterward through `serviceRedeemRequest`. Alternatively, consider allowing `serviceRedemptions` to accept an explicit list of request IDs to service, so that liveness for eligible requests does not depend on the position of ineligible requests within the request ID space.

Update: Resolved in [pull request #93](#) at commit [14f2b84](#). The team stated:

The proposed change advances the batch-service cursor to the end of a bounded scan when more request IDs remain, so an ineligible pending request cannot trap every later batch in the same window; skipped requests are retried after the scan reaches the current stream end.

M-05 Unbounded Request Lists Can DoS `withdraw` and `redeem`

`StakedUSDx` records redemption requests per controller by appending each new request identifier to the `_controllerRequests` array. When a controller later claims through the ERC-4626 entry points `withdraw` and `redeem`, the resulting calls to `_claimRedeemByShares` and `_claimRedeemByAssets` linearly scan `_controllerRequests[controller]` starting at `_claimIndex[controller]`, advancing the index past every entry whose status is not `CLAIMABLE`. The gas consumed by a claim is therefore proportional to the number of entries traversed, and the updated `_claimIndex` is persisted only when the enclosing transaction succeeds.

The `requestRedeem` function authorizes the caller solely against the `owner` whose shares are burned, through `_isControllerOrOperator(owner, msg.sender)`, and performs no check binding the caller to the `controller` under which the request is stored. Consequently, any account holding a negligible amount of shares can append request identifiers to the list of an unrelated `controller`, because each call pushes onto `_controllerRequests[controller]` for the caller-chosen `controller`. These injected requests remain `PENDING`, are never pruned, and are not compacted by `cancelRedeemRequest`, which leaves cancelled entries in place. Each injected request need only carry a minimal share amount, so the attacker commits negligible value while still consuming a storage slot in the target list.

By pre-filling a victim controller's list with a large number of requests before that controller acquires claimable requests, an attacker forces every subsequent claim to traverse the injected entries before reaching any `CLAIMABLE` request. Because skipping a non-`CLAIMABLE` entry advances the index without reducing the amount still to be claimed, the loop cannot stop early,

so once enough entries are injected the traversal exceeds the block gas limit and the claim reverts without persisting progress, blocking the controller's `withdraw` and `redeem` claims until the injected entries are serviced and cleared through a large number of minimal claims.

Consider restricting insertion into a controller's request list to callers authorized by that controller, for example by requiring `_isControllerOrOperator(controller, msg.sender)` in `requestRedeem`. Alternatively, consider allowing a claim to reference specific request identifiers so that unrelated entries cannot block a controller's claims.

Update: Resolved in [pull request #94](#) at commit [1767353](#). The team stated:

The proposed change requires a requestRedeem caller to be authorized by both the share owner and the selected controller, preventing insertion into an unrelated controller's request list while retaining approved split owner/controller flows.

M-06 Unauthenticated `accountId` Undermines ACCOUNT-scope Caps and Nonce Namespaces

Order settlement authenticates the order originator solely through its signer. Both `settle` and the preflight `checkSettlement` route through `_checkSettlement`, whose only account-level authorization is `isWhitelisted(order.signer)`, backed by a flat address-to-bool `whitelist`. The order is EIP-712 signed over a struct that includes `accountId`, yet `accountId` is validated only for being `non-zero`. Nothing binds `accountId` to the signer: the `_delegatedSigners` mapping is an unrelated signer-delegation feature, and `_checkChannel` never reads the `accountId` it is passed. Because `accountId` is a signed field chosen by the signer, a whitelisted signer can set it to any non-zero value and the signature still validates.

Despite being unauthenticated, `accountId` is the sole key for two pieces of settlement state. The replay nonce bitmap is written by `_markNonceUsed` into an `accountId`-indexed slot, and the ACCOUNT-scope cap is derived from `_accountKey(request.accountId)` in both the `capacity check` and the `capacity consumption`. This lets a signer subject to a restrictive per-account cap escape it, while also enabling targeted griefing of another account's namespaces.

Exploitability is bounded by the trusted-operator model. A `MINT_OPERATOR` or `REDEEM_OPERATOR` must relay every order for `settle` to proceed, and the documented request-for-quote flow has the operator assign `accountId`, so an approved counterparty does not ordinarily get to choose it. Global, asset, route, channel, and per-block caps still bound aggregate flow, so this is not an unbounded mint or a global supply breach. The gap is

that the ACCOUNT-scope cap and the `accountId`-keyed nonce namespace are not self-enforcing at the contract layer and instead silently depend on the operator never relaying an order carrying a signer-chosen `accountId`.

Consider binding `accountId` to the signer on-chain before trusting it for accounting, for example an admin-registered signer-to-account mapping asserted in `_checkSettlement`, or keying the nonce bitmap and the ACCOUNT-scope cap on the signer (or on the pair of `accountId` and signer) rather than on `accountId` alone. Until `accountId` is authenticated, the ACCOUNT-scope cap and the `accountId`-keyed nonce namespace cannot be relied upon as per-counterparty controls.

Update: Resolved in [pull request #95](#) at commit [5ad57e6](#). An admin-only `_signerAccounts` mapping now records each signer's permitted account, and `_checkSettlement` now rejects any order whose `marketConfig.signerAccount(order.signer)` does not equal `order.accountId`. An admin may bind two or more signers to the same `accountId`, in which case they share the nonce namespace and ACCOUNT cap.

M-07 Sparse Code Comments and Incomplete Branch Coverage

The four implementation contracts in scope total roughly 2,300 Solidity lines but contain fewer than 15 comment lines combined and no function-level `NatSpec` describing purpose, parameters, or preconditions. Security-critical logic is left unexplained, and without comments, reviewers cannot separate deliberate invariants from oversights and maintainers cannot change the code safely.

Test coverage, measured with `forge coverage --ir-minimum`, shows high line and function coverage but incomplete branch coverage. Branches are the conditional guards (whitelist, cap, pause, restriction, and validation checks) that carry the system's security properties, so the reverting side of many guards is never exercised.

Contract	Lines	Branches	Functions
<code>StakedUSDx.sol</code>	83.16%	52.63%	86.49%
<code>MarketConfig.sol</code>	84.58%	47.83%	85.11%
<code>USDxMarket.sol</code>	78.40%	56.14%	67.86%
<code>USDx.sol</code>	94.34%	50.00%	100.00%

Contract	Lines	Branches	Functions
Total	88.04%	66.02%	88.12%

Consider adding function-level NatSpec to the four contracts, prioritizing the settlement, capacity, restriction, and redemption paths, and documenting the status and reason constants and each validation bound. Consider raising branch coverage toward parity with line coverage (around 90%), for example by adding negative-path tests that assert every guard rejects with its specific reason code.

Update: Partially Resolved in [pull request #96](#) at commit [aa5e9a8](#). `@param` and `@return` tags remain mostly missing from functions, and `MarketConfig` branch coverage sits at 79.69% as of [b2b449cc](#) (integration PR #114) because branches added during aggregate integration were not accompanied by tests. The team stated:

Added focused branch-guard tests across StakedUSDx, MarketConfig, USDxMarket, and USDx. Final branch coverage is StakedUSDx 93.88% (92/98), MarketConfig 91.84% (45/49), USDxMarket 96.77% (60/62), and USDx 100% (11/11).

Low Severity

L-01 Redundant `whenNotPaused` Modifier in `USDx`

In the `USDx` contract, the `whenNotPaused` modifier is applied to the `mint`, `burn`, `burnFrom`, `bridgeMint`, and `bridgeBurn` functions. All of these functions modify balances exclusively through `_mint` or `_burn`, which route every balance change through the overridden `_update` function. Since `_update` already reverts with `OperationPaused` when the contract is paused, the modifier duplicates a check that is enforced on every code path, adding redundant code and an additional storage read without affecting behavior.

Consider removing the `whenNotPaused` modifier from these functions and relying on the pause check in `_update` as the single point of pause enforcement.

Update: Resolved in [pull request #97](#) at commit [98ea836](#). The team stated:

The proposed change removes `whenNotPaused` from `USDx` `mint`, `burn`, `burn-from`, and `bridge` balance-changing entry points and retains `_update` as the single pause-enforcement point.

L-02 approve and permit Remain Functional While USDx Is Paused

The `USDx` contract implements a pause mechanism through the `whenNotPaused` modifier on its mint and burn functions and a corresponding check in the `_update` override, which blocks all transfers, mints, and burns while the contract is paused. However, the inherited `approve` and `permit` functions update allowances through `_approve`, which does not invoke `_update` and is not otherwise restricted. As a result, allowances can still be granted, modified, or revoked while `USDx` is paused, and any allowance created during a pause becomes spendable as soon as the contract is unpaused. This may be unexpected if the pause is intended to freeze all state-changing token operations, for example while an incident is being investigated.

Consider overriding `approve` and `permit` to apply the `whenNotPaused` modifier so that the pause covers all state-changing ERC-20 operations. Alternatively, if allowance updates are intended to remain available during a pause, consider documenting this behavior explicitly.

Update: Resolved in [pull request #98](#) at commit [1a3544d](#). The team stated:

The proposed change overrides USDx approve and permit to enforce the token pause before allowance or permit-nonce state can change.

L-03 Deposits and Mints Lack a Pause Control

The vault provides granular pause controls for its redemption lifecycle and reward funding: `pauseRedeemRequests`, `pauseRedeemClaims`, `pauseRedemptionServicing`, and `pauseRedeemCancellations` gate the redemption paths, and `pauseRewardFunding` halts reward inflows. The share-issuing path has no equivalent switch. The `_deposit` function, which backs both `deposit` and `mint`, checks only recipient restrictions via `_requireCanReceiveShares` and never consults a pause flag. As a result, new capital can continue to enter the vault even while every redemption path is frozen. In an emergency such as a suspected accounting fault, operators can stop assets from leaving but cannot stop them from arriving.

Consider adding a deposit and mint pause gated by `PAUSER_ROLE`, for symmetry with the existing redemption and reward controls, so that inflows can be halted alongside the redemption paths during incident response.

Update: Resolved in [pull request #114](#) at commit [7b54d8d](#). The team stated:

The proposed change adds a dedicated deposit-pause state with emergency pause and admin pause or unpause controls, and gates the shared ERC-4626 deposit path so both deposit and mint stop during an incident.

L-04 `previewWithdraw` and `previewRedeem` Do Not Revert

The `StakedUSDx` vault implements the ERC-7540 asynchronous redemption flow through `requestRedeem`, `pendingRedeemRequest`, `claimableRedeemRequest`, and the operator model. For asynchronous redeem vaults, ERC-7540 mandates that `previewWithdraw` and `previewRedeem` revert for all callers and inputs. However, `previewWithdraw` and `previewRedeem` return amounts converted at the current exchange rate. In addition to deviating from the standard, the returned values do not correspond to the amounts an actual claim would transfer, as `withdraw` and `redeem` settle at the per-request rates fixed when redemptions are serviced. Integrations that rely on standard-conformant behavior may treat these values as meaningful claim previews.

Consider having `previewWithdraw` and `previewRedeem` revert unconditionally, in accordance with ERC-7540.

Update: Resolved in [pull request #100](#) at commit [b0f9ebb](#). The team stated:

The proposed change makes `previewWithdraw` and `previewRedeem` revert unconditionally with `OperationNotAllowed`, matching the asynchronous ERC-7540 redeem model.

L-05 ERC-4626 `max*` Views Do Not Return Zero When Actions Are Disabled

The ERC-4626 standard requires each `max*` view to return the maximum amount for which the corresponding action would not revert, factoring in both global and user-specific limits, and to return zero when the action is disabled. In `StakedUSDx`, several of these views ignore the pause and restriction conditions that gate the operations they describe, so they report non-zero limits for actions that are guaranteed to revert.

`maxWithdraw` and `maxRedeem` return the owner's `claimableAssets` and `claimableShares` unconditionally. However, `withdraw` and `redeem` first call `_requireClaimOpen`, which reverts while `_claimPaused` is set, and

`_requireClaimAllowed`, which reverts when the owner is fully restricted. As a result, both views report a non-zero amount for an owner whose claims are globally paused or who is fully restricted, even though every corresponding `withdraw` or `redeem` call reverts. Likewise, `maxDeposit` and `maxMint` are not overridden and inherit the default value of `type(uint256).max`, while `_deposit` calls `_requireCanReceiveShares` and reverts for a restricted receiver. Integrations that rely on these views as a non-reverting upper bound therefore receive incorrect values.

Consider overriding `maxWithdraw` and `maxRedeem` to return zero when `_claimPaused` is set or the owner is fully restricted, and overriding `maxDeposit` and `maxMint` to return zero for a restricted receiver, so that each view reflects the conditions under which the corresponding action reverts.

Update: Resolved in [pull request #114](#) at commit [c38b44b](#). The team stated:

The integrated change makes every max view match the action it describes: `maxWithdraw` and `maxRedeem` return zero during claim pause or full restriction, while `maxDeposit` and `maxMint` return zero for restricted receivers and during the L-03 deposit pause.

L-06 Blacklisted Spenders Can Still Move Funds Through Existing Allowances

`USDx` enforces its blacklist inside the ERC-20 transfer hook `_update`, which reverts whenever the `from` or `to` party of a transfer is restricted. Addresses are added to and removed from the restriction set by the `RESTRICTION_ROLE` through `blacklist` and `unBlacklist`.

The `_update` hook validates only the `from` and `to` addresses of a transfer and never considers `msg.sender`, the account spending an allowance. Allowance-based flows such as `transferFrom` and `burnFrom` reach `_update` with `from` set to the token holder rather than to the operator, so a blacklisted spender is never checked. As a result, an address that has been blacklisted can continue to exercise allowances granted before the restriction was applied: it can call `transferFrom` to move tokens between two non-blacklisted addresses, or `burnFrom` to burn tokens from a non-blacklisted holder. Blacklisting is commonly relied upon as an incident-response control to neutralize a compromised router or spender without pausing the entire token, and this gap defeats that expectation for any allowance already in place.

Consider explicitly restricting allowance-based actions performed by blacklisted spenders, for example by overriding `transferFrom` and `burnFrom` to revert when `msg.sender` is blacklisted while retaining the existing `from` and `to` checks in `_update`. Consider also documenting whether the blacklist is intended to be sender-based, holder-based, or both, as this determines whether allowances should remain usable once an address is blacklisted.

Update: Resolved in [pull request #102](#) at commit [826bbf3](#). The team stated:

The proposed change checks `msg.sender` in both `transferFrom` and `burnFrom` and rejects a blacklisted allowance spender before allowance or balance state is modified, while preserving existing source and destination checks in `_update`.

L-07 Soft-Restricted Owners Can Redeem Through an Unrestricted Controller

The claim-time check `_requireClaimAllowed` enforces that a soft-restricted account may only claim redemption proceeds to itself, reverting when `receiver != owner`. However, `requestRedeem` permits a soft-restricted owner to record a redemption under an unrestricted third-party `controller`, since it checks only `_fullRestriction` for the `owner` and `controller` at [L250-L252](#). The original share owner recorded in `RedemptionState.owner` is never consulted again during servicing or claiming, so the redemption is evaluated against the `controller`, and the `controller` receives the proceeds.

In the common configuration this grants no additional capability, because a soft-restricted owner may already redeem to itself and then transfer the freely transferable underlying `USDx` onward. Soft restriction governs share movement rather than the flow of the underlying asset, and self-redemption to the owner is explicitly permitted.

The gap has practical effect only when an account is simultaneously soft-restricted on `StakedUSDx` and blacklisted on the underlying `USDx`. In that case, a direct self-redemption reverts on the `USDx` transfer to the owner, while a redemption routed through an unrestricted controller completes. This leaves the atomic `receiver != owner` restriction circumventable through the controller mechanism, even though full restriction remains the intended control for accounts that must not exit at all.

Consider requiring `controller == owner` in `requestRedeem` when the owner is soft-restricted, so that the claim mechanism cannot be used to bypass the intended asset restrictions.

Update: Resolved in [pull request #103](#) at commit [7c78a24](#). The team stated:

The proposed change requires a soft-restricted share owner to select itself as redemption controller, preventing routing through an unrestricted third party.

L-08 Restricted Accounts Can Deposit And Mint Shares To Unrestricted Receivers

`StakedUSDx` maintains an internal restriction system, tracked by `_restricted` and `_fullRestriction`, that is configured through `addToBlacklist` and surfaced as `blockedActions` in `getAccountState`. A soft-restricted account is reported as `BLOCKED_NON_EXIT` and a fully restricted account as `BLOCKED_ALL`, and these guarantees are enforced on share transfers and burns in `_update` and on claims in `_requireClaimAllowed`.

The ERC-4626 entry path routes through the overridden `_deposit`, which validates only the `receiver` through `_requireCanReceiveShares` and never inspects the `caller` whose assets are pulled by `_assetToken().safeTransferFrom(caller, address(this), assets)`. As a result, a restricted or fully restricted account can still initiate `deposit` or `mint`, funding a new or increased `sUSDx` position for any unrestricted third party even though its `blockedActions` indicate that such entry actions are disallowed. For example, an account marked restricted through `addToBlacklist` can approve `StakedUSDx`, call `deposit(assets, receiver)` with an unrestricted `receiver`, and convert its `USDx` into `sUSDx` held by that receiver. This contradicts the restriction model's blocked-actions guarantees and may undermine off-chain compliance assumptions, as it allows restricted accounts to move value into the vault on behalf of others. The effect is bounded by any independent restrictions enforced by the underlying `USDx` token, which could block the asset transfer originating from a restricted caller.

Consider enforcing restriction checks on the `caller` within `_deposit`, and across any other vault entry points, so that restricted and fully restricted accounts cannot initiate deposits or mints regardless of the designated receiver.

Update: Resolved in [pull request #104](#) at commit [18e53fa](#). The team stated:

The proposed change validates the asset-supplying caller inside the shared deposit path, so both soft- and fully restricted accounts are blocked from initiating deposits or mints even when the receiver is unrestricted.

L-09 `withdraw` Can Revert for Asset Amounts Within `maxWithdraw` Due to Ceil Rounding

`StakedUSDx` records asynchronous redemptions as per-request pairs of `shares` and `assetsReserved`, where `assetsReserved` is fixed at request time in `requestRedeem` as `_convertToAssets(shares, Math.Rounding.Floor)`. Once a request becomes claimable, `withdraw` validates the requested amount only against the aggregate controller limit reported by `maxWithdraw`, which returns the sum of claimable assets, and then settles that amount across individual requests in `_claimRedeemByAssets`.

For a partial claim against a request, where `assetsToClaim` is less than `assetsReserved`, `_claimRedeemByAssets` computes the shares to burn as `assetsToClaim.mulDiv(state.shares, state.assetsReserved, Math.Rounding.Ceil)` and `reverts with InsufficientClaimable` whenever this ceil rounding yields a value greater than or equal to the request's remaining shares while not all of the reserved assets are consumed. As a result, certain intermediate asset amounts are not claimable even though `maxWithdraw` reports them as available. For example, a claimable request with `state.shares` equal to 2 and `state.assetsReserved` equal to 3 reports a `maxWithdraw` of 3, but a call to `withdraw` for 2 assets computes `sharesToClaim` as `ceil(2 * 2 / 3)`, which equals 2. Because this consumes all remaining shares while leaving one reserved asset unclaimed, the guard reverts and the withdrawal fails even though the requested amount does not exceed `maxWithdraw`.

This is primarily a liveness and composability concern rather than a loss of funds, and the unclaimable band for any single request is narrow, on the order of `assetsReserved / shares`. However, under the ERC-4626 interface `maxWithdraw` is expected to be a safe upper bound for non-reverting `withdraw` calls. Integrations that rely on exact-asset withdrawals may be forced into trial-and-error amounts, or into claiming full per-request balances, degrading composability for controllers holding affected requests.

Consider aligning `maxWithdraw` with the amounts that `_claimRedeemByAssets` can actually settle so that it remains a safe non-reverting upper bound. Alternatively, if the current behavior is intended, consider documenting this rounding constraint so integrators do not rely on `maxWithdraw` as a guaranteed non-reverting bound for `withdraw`.

Update: Resolved in [pull request #114](#) at commit [7b54d8d](#) by adding clear documentation on the behavior of the `withdraw` function.

L-10 Incorrect ERC-165 Integration in StakedUSDx

Axis documents `sUSDx` as an ERC-4626 vault share token that supports ERC-7540 asynchronous redemptions, and the codebase models the asynchronous flow through `IERC7540Redeem` and its base `IERC7540Operator`, both of which extend `IERC165`. The [ERC-165 support section of EIP-7540](#) requires a compliant contract to return `true` from `supportsInterface` for the operator identifier `0xe3bc4e65`, the asynchronous redemption identifier `0x620ee8e4`, and the ERC-7575 identifier `0x2f0a18c5`, and to expose a `share` function for share token discovery.

The `supportsInterface` implementation of `StakedUSDx` reports support only for the Axis-local composite interfaces and returns `false` for each of the standardized identifiers above. The single ERC-7540 identifier the contract attempts to advertise is also degenerate: it reports support for `type(IAsyncRedeemVault).interfaceId`, but `IAsyncRedeemVault` is an empty interface that only inherits `IERC7540Redeem`. Because Solidity computes `type(I).interfaceId` as the exclusive-or of the selectors declared directly in the interface and excludes inherited functions, an interface with no declared functions yields `0x00000000`. As a result, the contract returns `true` for `supportsInterface(0x00000000)` while returning `false` for the `0x620ee8e4` and `0xe3bc4e65` identifiers that its inherited functions actually resolve to. In addition, `StakedUSDx` does not implement a `share` function.

Integrations that follow the standard and probe these identifiers through ERC-165, or that call `share` to discover the share token, will treat the vault as non-compliant. The impact is limited to external interoperability and does not affect the vault's internal accounting.

Consider extending `supportsInterface` to return `true` for the concrete standardized identifiers `type(IERC7540Redeem).interfaceId`, `type(IERC7540Operator).interfaceId`, and the ERC-7575 identifier, rather than the empty `IAsyncRedeemVault` composite, and implementing `share` with the intended semantics. Consider also adding a regression test that asserts the exact `supportsInterface` responses and the presence of `share` expected by external routers and adapters.

Update: Resolved in [pull request #106](#) at commit [91464b0](#). The team stated:

The proposed change advertises the concrete ERC-7540 operator and async-redeem IDs plus the ERC-7575 vault ID, stops advertising the empty composite ID, and adds `share()` returning the vault and share-token address. Regression tests assert the exact

standard IDs, rejection of 0x00000000, unknown-interface behavior, share discovery, and selector freeze.

L-11 Rewards Funded While No Shares Exist Are Captured by the First Depositor

Rewards are distributed through time-based vesting. `fundRewards` records the funded amount in `_unvestedRewards` and starts a vesting schedule, and `_vestedRewards` progressively folds the vested portion into the accounted assets that back `totalAssets`. Share pricing in `_convertToShares` scales the deposited assets by `totalSupply` over the accounted assets, except when `totalSupply` is zero, in which case it returns the deposited amount unchanged so that the first deposit mints one-to-one.

The `fundRewards` function does not require any shares to exist when it is called. If rewards are funded while `totalSupply` is zero, for example before the first deposit at launch or after every share has been redeemed while an amount is still vesting, the vested rewards accrue into `totalAssets` even though no shares back them. This causes the first depositor after rewards are funded to capture these accrued rewards entirely, instead of them being distributed proportionally across a share supply.

Consider rejecting `fundRewards` when `totalSupply` is zero, or deferring the start of the vesting schedule until shares exist, so that funded rewards are only ever accrued while there is a share supply to distribute them to.

Update: Resolved in [pull request #107](#) at commit [3c427a9](#). The team stated:

The proposed change rejects fundRewards whenever totalSupply is zero, preventing rewards from vesting without an active share supply.

L-12 USDx Blacklist Does Not Freeze Market Activity

The USDx token enforces its pause and blacklist restrictions only inside the `_update` hook, which governs USDx transfers, mints, and burns. Market settlement gates actors through the `MarketConfig` whitelist rather than the token restriction state. Because a MINT and a REDEEM each move USDx on only one of their two legs, blacklisting an actor at the token layer freezes that actor on some settlement paths but leaves others open.

Considering the four combinations of blacklisted actor and settlement side:

1. Blacklisted signer on MINT: the signer-side leg is the collateral transfer (not USDx) and USDx is minted to the receiver, so settlement proceeds. A blacklisted signer can still drive a MINT to a clean whitelisted receiver.
2. Blacklisted receiver on REDEEM: the receiver is paid in collateral (not USDx), so the payout proceeds. A blacklisted receiver can still collect a REDEEM payout.
3. Blacklisted signer on REDEEM: `usdx.burnFrom` reaches `_update` with the blacklisted address as the source and reverts, so this path is correctly blocked.
4. Blacklisted receiver on MINT: `usdx.mint` reaches `_update` with the blacklisted address as the destination and reverts, so this path is also correctly blocked.

Because two of the four combinations settle normally, fully freezing an actor across both mint and redeem requires governance to remove the actor from the market whitelist (via `setWhitelist`) and/or disable the relevant side through `setMintRedeemEnabled`, not to rely on the USDx blacklist alone. Compounding the operational gap, the public pre-flight checkers `checkSettlement`, `checkMint`, and `checkRedeem` never consult the USDx restriction state. The only calls into the token anywhere in the contract are `usdx.mint` and `usdx.burnFrom`, so neither `paused()` nor `isBlacklisted()` is ever read. These checkers can therefore return an "allowed" verdict for an order that then reverts inside `USDx._update` on pause or blacklist, a false positive that off-chain integrators must not treat as authoritative. Settlement itself stays atomic: a reverted settle consumes no nonce or capacity and is resubmittable, so this is a view-versus-execution divergence and a policy-surface gap, not a funds or atomicity defect.

Consider documenting that freezing an actor requires removing them from the market whitelist (and pausing where appropriate) rather than only calling the token blacklist. If token-level restriction is intended to gate the market, consider having `_checkSettlement` and the view checkers consult the USDx pause and blacklist state for the signer and receiver addresses that will reach `_update`.

Update: Resolved in [pull request #108](#) at commit [a6526b0](#). A minor observation is that the settle-time revert for the paused case falls through to the generic `InvalidOrder()` in `_revertForReason` rather than a dedicated pause error. The team stated:

The proposed change makes market preflight and settlement consult USDx pause and blacklist state for both signer and receiver before nonce or capacity consumption or token movement, returning explicit restriction reason codes. Regression tests cover the four signer or receiver and mint or redeem blacklist combinations plus paused-token preflight behavior. The finding-specific commit is included in open aggregate PR #114.

L-13 Documented Per-User Block Limits Are Unimplemented Stubs

The market exposes two per-user view functions, `getUserLimit` and `getUserBlockCapacity`, which forward to the corresponding functions in `MarketConfig`, and the documentation describes per-user block limits as a risk control. Neither getter behaves per-user. `getUserLimit` is declared `pure`, ignores its `account` argument, and always returns zeroed limits with `custom` set to false. `getUserBlockCapacity` copies the supplied `account` into the returned struct but derives every quantitative field from the global per-block counters, so it reports the same protocol-wide figures as `getCurrentBlockCapacity`.

Settlement matches this: `_checkSettlement` enforces the whitelist, the mint and redeem flags, the channel, the global per-block cap, and the multi-scope cap engine, but never consults any per-user limit. An off-chain integrator or risk dashboard reading these getters would receive the protocol-wide block capacity while expecting a specific account's remaining allowance, over-reporting how much that account may mint or redeem.

A per-account throttle does exist, through the ACCOUNT-scope cap keyed by `_accountKey`, which is enforced by the capacity engine and consumed on settlement. The residual issue is therefore a naming, ABI, and documentation gap: two functions carry per-user names but return global values, and the documented per-user block limit is not the mechanism actually implemented. These are ungated views with no effect on settlement, so the only consequence is that an integrator may misread the data.

Consider either implementing and enforcing genuine per-user block limits (for example, with dedicated storage plus a check in `_checkSettlement`), or removing the stub getters and updating the documentation and ABI so integrators do not treat the global figures as per-user, and instead point them to the ACCOUNT-scope cap for per-account throttling.

Update: Resolved in [pull request #109](#) at commit [9c7727e](#). `getUserLimit` and `getUserBlockCapacity` were kept for backward compatibility but should not be used since they always return zero. The team stated:

The proposed change retains the legacy user-capacity selectors for ABI compatibility but makes them explicitly non-account-scoped, returns zeroed user capacity instead of echoing global counters, and directs integrations to ACCOUNT-scope capUsage keyed by accountId. Regression tests verify the legacy views never report protocol-wide usage while the actual ACCOUNT cap remains enforced. The finding-specific commit is included in open aggregate PR #114.

Notes & Additional Information

N-01 Inconsistent Naming in the `StakedUSDx` Asynchronous Redemption Flow

`StakedUSDx` implements an asynchronous redemption model in which `requestRedeem` assigns control of the resulting position to a `controller`, and every request-keyed mapping, including `_redeemRequests`, `_redemptions`, and `_requestController`, is indexed by that `controller`. The `withdraw` and `redeem` functions, together with their internal helpers, instead name this account `owner` even though they read the `controller`. Naming the same account `owner` on the claim path is inconsistent with the rest of the flow and can mislead readers into conflating it with the ERC-4626 share owner.

Separately, `_isControllerOrOperator` names its first parameter `controller`, but the function only verifies that the caller is the account itself or one of its approved operators, carrying no controller-specific meaning.

Consider renaming `_isControllerOrOperator` to `_isSelfOrOperator` to reflect that it only checks self-or-operator authorization, and renaming the `owner` parameter in `withdraw`, `redeem`, and their internal helpers to `controller` to match the terminology used throughout the redemption request flow.

Update: Resolved in [pull request #110](#) at commit [3d2688e](#). The team stated:

The proposed change renames claim-path owner parameters to controller and `isControllerOrOperator` to `_isSelfOrOperator`, aligning terminology with controller-keyed redemption state without changing selectors or runtime behavior. The request and claim documentation now distinguishes share-owner request authority from controller claim authority.

N-02 Per-Block Caps Key on Block Number, Whose Wall-Clock Meaning Differs Across Deployment Chains

`MarketConfig` enforces two block-scoped throttles. The `maxMintPerBlock` and `maxRedeemPerBlock` limiter accumulates settled amounts into per-block mappings inside `consumeBlockCapacity` and reads them back through `getCurrentBlockCapacity`, and the optional `WindowType.BLOCK` cap window buckets usage by the current block. All of these key on `block.number` and reset when it advances; `USDxMarket` applies the limiter through `__checkBlockCapacity`.

The wall-clock meaning of `block.number` is not uniform across chains. On Arbitrum it tracks the (slow) L1 block and is shared across many fast L2 transactions, while on some L2s it is the (fast) L2 block. A per-block cap tuned for Ethereum mainnet therefore changes character when the same value is deployed elsewhere. The direction of the effect is asymmetric:

1. On Arbitrum-style chains where `block.number` is shared across many blocks, the per-block cap becomes more restrictive, since those transactions compete for a single block's allowance.
2. On faster-block L2s the same numeric cap becomes more permissive per wall-clock second, because more blocks (and therefore more full per-block allowances) elapse in the same real-time interval.

This is a calibration caveat rather than an attacker-controllable defect: `block.number` is not manipulable by order submitters, and settlement remains gated on the trusted operator roles.

Consider recalibrating `maxMintPerBlock`, `maxRedeemPerBlock`, and any `WindowType.BLOCK` cap for each target chain's block cadence (both limits are governance-tunable via `setMaxMintPerBlock` and `setMaxRedeemPerBlock`), and documenting that "per block" is not necessarily a fixed wall-clock rate. Where a wall-clock throttle is intended, consider a time-based `WindowType.FIXED` cap instead.

Update: Resolved in [pull request #111](#) at commit [f0c6f95](#). The team stated:

The proposed change documents that global per-block limits and `WindowType.BLOCK` caps are keyed by `block.number`, require per-chain calibration, and should use `FIXED` timestamp windows when a wall-clock throttle is intended. A regression test verifies elapsed time alone does not reset a block window and advancing `block.number` does; no runtime cap-semantics change is proposed. The finding-specific commit is included in open aggregate PR #114.

N-03 Reason Codes Use Inconsistent Encodings Across the Freeze Surface Taxonomy

The `AxisV2Ids` library defines the canonical `REASON_*_identifiers` as `keccak256` hashes of their labels, whereas `MarketConfig` and `USDxMarket` define the equivalent `CHECK_*_codes` as `bytes32` string literals and return them as the `reasonCode` field of the `checkSettlement` result. Because `keccak256("LABEL")` does not equal `bytes32("LABEL")`, the two encodings are not comparable even when their labels are identical.

No on-chain logic equates the two representations, so the mismatch does not create an authorization or validation flaw by itself. However, off-chain policy enforcement, alerting, or automation that treats `AxisV2Ids` as the canonical namespace and compares a returned `reasonCode` against an `AxisV2Ids.REASON_*` constant would never match, silently misclassifying checks.

Consider standardizing on a single reason-code encoding across `AxisV2Ids`, `MarketConfig`, and `USDxMarket`.

Update: Resolved in [pull request #112](#) at commit [0402daf](#). The team stated:

The proposed change converts the AxisV2Ids REASON family to the same readable short-bytes32 encoding returned by MarketConfig and USDxMarket, while leaving role, action, surface, and selector identifiers hashed. Regression tests verify representative live preflight codes compare equal to the canonical reason constants. The finding-specific commit is included in open aggregate PR #114, whose current integration checks pass; no remediation PR is merged to main.

Conclusion

Axis V2 is a primary-market stablecoin system. Its in-scope contracts settle EIP-712 signed mint and redeem orders against off-chain-priced collateral: `USDxMarket` executes each settlement while `MarketConfig` enforces routing, whitelisting, and multi-scope capacity limits alongside a per-block throughput limiter and emergency controls. `USDx` is the resulting ERC-20 stablecoin, extended with pause, blacklist, and bridge controls, and `StakedUSDx` is an ERC-4626 / ERC-7540 vault that offers staking through an asynchronous, cooldown-based redemption flow and linearly vesting rewards. All four contracts are upgradeable behind proxies and depend on a set of privileged roles for configuration and day-to-day operation.

At High and Medium severity, the findings clustered in two areas. In the staking vault, weaknesses in the asynchronous redemption flow can strand, delay, or block user claims, the reward-funding logic can distort or delay vesting, and the account-restriction surface can leave some balances unreachable. On the market side, the findings concern control semantics that diverge from documented intent: the emergency control disables a capacity limit instead of enforcing it, and the unauthenticated order `accountId` leaves ACCOUNT-scope caps and the nonce namespace non-self-enforcing on-chain. The remaining findings were lower severity, concerning interface conformance, rounding, and access-control edge cases in the token and vault.

Two cross-cutting weaknesses run through the codebase. First, the contracts are sparsely documented, with almost no function-level NatSpec, and test coverage is incomplete, with branch coverage leaving security-critical logic unexercised; together these make it difficult to distinguish deliberate invariants from oversights and to modify the code safely, and both should be addressed before deployment. Second, the system rests on broad trust assumptions: core guarantees (collateral backing and redemption solvency, order pricing, the freezing of actors, and correct cross-contract role wiring and upgrades) are enforced off-chain by governance, operators, and custodians rather than by the contracts themselves, and these assumptions should be documented prominently so that users understand which guarantees the protocol does and does not provide on-chain. We thank the Axis team for being available to clarify the system's design and intended behavior over the course of the review.

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.