OpenZeppelin | security

Symbiotic Core v2 Adapters Audit

SYMBIOTIC

June 29, 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

Additional Considerations 8

Medium Severity 10

M-01 Token-to-Redeem Balances Below the Redemption Vault Minimum Freeze sync and convert 10

M-02 In-Flight Swap Balances in LiquidLaneAdapter Can Be Stolen by Arbitrary Callers 11

M-03 BTC and ETH Midas mTokens Undervalued in Account.totalAssets 12

M-04 BTC and ETH mTokens Cannot Be Redeemed, Bricking Swaps and Stranding Inventory 13

M-05 Predictable Jumps in totalAssets Allow Timed Value Extraction From Passive Holders 14

M-06 Deallocation Reverts on a Frozen Account, Bricking Vault-Wide Operations 15

M-07 Single Token Oracle Failure Can Brick the Vault With Self-Locked Recovery 16

Low Severity 17

L-01 Mixed Namespaced and Sequential Storage Risks State Corruption on Migration 17

L-02 Directly Deployed CoWSwapConverter Is Permanently Ownerless 18

L-03 Dust Donations Block Token-to-Redeem Removal 19

L-04 Token-Scoped Swap Nonces Allow Cross-Signer Collisions and Cancellation 19

L-05 Missing Zero-Address Validation on Address Parameters 20

L-06 Asymmetric State Handling Between Adding and Removing a Token-to-Redeem 21

L-07 Acquire-Funded Swaps Revert When the Account Exceeds Its Limit 23

L-08 Per-Token Exposure Cap Is Overshot by the Swap Discount Margin 23

L-09 MidasNonCompAccount Pending-Request Valuation Ignores the Recorded tokenOutRate 24

Notes & Additional Information 25

N-01 Typographical Error in Identifier Names 25

N-02 Floating and Inconsistent Pragma Directives 26

N-03 Possible Duplicate Event Emission 26

N-04 Inconsistent Use of Returns in Functions 27

N-05 Missing Error Message in Revert Statement 28

N-06 Functions Updating State Without Event Emissions 28

N-07 Incorrect or Misleading Documentation 29

N-08 Unused Declarations 30

N-09 Inconsistent Naming Between Interface and Implementation 30

N-10 Redundant _hashDiscount Recomputation in the Discount-Swap Path 31

N-11 Linear Scan of the converters Array Increases Gas Cost of Conversions 31
N-12 Redundant nonReentrant Modifier on the migrate Function 32

Conclusion 33

Appendix 34
Issue Classification 34

Summary

Type	DeFi
Timeline	From 2026-06-08 To 2026-06-17
Languages	Solidity

Total Issues	28 (11 resolved, 2 partially resolved)
Critical Severity Issues	0 (0 resolved)
High Severity Issues	0 (0 resolved)
Medium Severity Issues	7 (1 resolved, 2 partially resolved)
Low Severity Issues	9 (4 resolved)
Notes & Additional Information	12 (6 resolved)
Client Reported Issues	0 (0 resolved)

Scope

OpenZeppelin performed an audit of the `symbioticfi/core-mirror` repository at the `d9c5fe0` commit.

In scope were the following files:

```
src/contracts
├── adapters
│   ├── Adapter.sol
│   ├── LiquidLaneAdapter.sol
│   ├── common
│   │   └── CoWSwapConverter.sol
│   └── ll-adapter
│       ├── AccountRegistry.sol
│       ├── MidasAccount.sol
│       ├── common
│       │   ├── Account.sol
│       │   └── CooldownAccount.sol
│       └── oracles
│           └── MidasOracle.sol
└── common
    └── MigratableEntity.sol
```

System Overview

The in-scope contracts implement the `LiquidLaneAdapter`, a single-vault adapter for the Symbiotic Core v2 vault system that provides issuer-facing instant redemptions. A vault allocates its underlying asset to the adapter, and the adapter lets authorized parties exchange a configured redemption token, such as a Midas mToken representing a real-world asset, for the vault asset at an oracle-derived, discounted rate. Each adapter is bound to a single vault and can support several token-to-redeem types at once, each backed by its own redemption account.

Adapter and redemption accounts: The `LiquidLaneAdapter` holds no long-term inventory of its own. For each supported token-to-redeem it deploys a dedicated redemption account through a registry-resolved factory (`AccountRegistry` resolving to a `MidasAccount` variant). An account custodies the redeemed inventory, prices it through an oracle, and over time converts it back into the vault asset. The adapter sums each account's valuation to report its net asset value, which the vault and its delegator consume for share pricing and allocation decisions. `MigratableEntity` is the shared upgradeable base for the adapter and the accounts, with creation and migration gated by their respective factories.

Swap flow: A swap transfers the vault asset out of the adapter to a recipient and routes the incoming token-to-redeem either to pre-funded acquire balances, which represent cash that the curator or a market maker has pre-deposited, or to the redemption account, which submits it for redemption. Three swap entry points exist. The direct path (`swap(Swap)`) requires the caller to be an authorized account. The delegated path (`swap(SignedSwap)`) carries a signed authorization that binds the caller, recipient, and amounts. The discount path (`swap(DiscountSwap)`) executes a reusable discount policy signed by an authorized party and wrapped in a short-lived protocol cosignature, and is callable by any address. A configurable per-token minimum discount and exposure limit bound every swap.

Off-chain and external actors: Market makers and fillers are authorized swap accounts that quote and execute redemptions, may pre-fund acquire balances, and sign swap or discount instruments. Converters are addresses permitted to liquidate an account's non-asset balances, such as reward tokens or fallback redemption proceeds, into the vault asset through CoW Protocol pre-signed orders, with a delayed and permissionless conversion path also available to non-converters. The shipped accounts integrate the Midas RedemptionVault: a `MidasAccount` submits held mTokens for asynchronous redemption on a cooldown, finalizes

processed requests, and prices outstanding requests at either the current oracle rate (compounding variant) or each request's locked rate (non-compounding variant). `MidasOracle` adapts a Midas data feed into an 18-decimal price.

Security Model and Trust Assumptions

The system places substantial trust in a small set of privileged, semi-trusted roles, but that trust is bounded. The per-token minimum discount and exposure limit enforce that a swap never pays out more vault asset than the oracle value of the token-in less the configured discount, so the vault always receives inventory worth at least the cash it funds. As a result, no privileged role can reach vault depositor principal through the swap path. The residual blast radius of a malicious or compromised authorized role is limited to grabbing lower-trust parties' transient token-in floats, over-discounting within the configured floor, denying service through pausing or configuration, and, for the curator alone, selling an account's redemption proceeds through a converter at an unfavorable price. That last capability is the only lever that escapes the rate floor, because the conversion path carries no on-chain price floor.

Curator (adapter and account owner): Appoints the market maker, pauser, and unpauser, adds and removes tokens-to-redeem, sets each token's minimum discount and exposure limit, and owns every redemption account it creates, which grants control over each account's converter set and migrations. The minimum discount is the only on-chain bound on how much vault asset a swap pays per unit of token-in, so the curator is trusted to set it soundly. A zero or near-zero floor removes the vault's spread and its margin against oracle error. The curator is further trusted to set sensible limits, to add only legitimate tokens that are value-equivalent to the vault asset and never the vault asset itself or a token with transfer callbacks or transfer fees, to appoint honest delegates, and not to grief redemptions through configuration. The curator's ability to register a converter that immediately sells an account's redemption proceeds at any price is the single most powerful trusted capability.

Market maker: An authorized swap account that quotes and executes swaps, signs delegated-swap and discount instruments, appoints fillers, and, when enabled, pre-funds acquire balances. A signed discount is a reusable authorization that does not bind the recipient or amount, so it grants bearer-like exposure until revoked or expired. The market maker is trusted to revoke it before misuse and not to redirect co-parties' in-flight token-in floats. Because acquire-funded payouts are sent to each party's configured receiver, the market maker, like the

curator for its own receiver, is further trusted to set a receiver that neither reverts nor is blocked from receiving the asset, since a reverting receiver on the market-maker tranche reverts every swap that spends it.

Fillers: Authorized swap accounts appointed by the market maker that execute and sign instruments under the same honest-execution and no-float-grab expectations. Their trust is derived entirely from the market maker's appointment.

Pauser and unpauser: Pause and unpause all swaps. A malicious pauser can deny service to the entire redemption desk.

Converters: Bypass the permissionless conversion delay and can pre-sign a sale of an account's convertible balance at any minimum output through CoW Protocol. They are therefore fully trusted with each account's convertible balance, and the curator is trusted to keep the converter set correct and bounded. Non-converters can also drive conversions through a delayed, permissionless path that has no on-chain price floor. The design relies on converters monitoring that path and vetoing a malicious request within the one-day execution delay, or cancelling a pre-signed order within its short validity window. A monitoring lapse longer than the delay would allow convertible balances, including redemption proceeds, to be sold at near-zero price through the open solver network.

Authorized swap set (cross-cutting): Because a swap spends whatever token-in the adapter already holds and pays a caller-chosen recipient, every authorized swapper can consume another authorized party's in-flight or pre-positioned token-in and redirect the proceeds. The authorized set is therefore assumed to be mutually trusting, and appointing a filler effectively grants it float-grab capability over every other party's swaps.

Factory owner: Whitelists the implementation versions that the adapter and accounts can be created or migrated to. Whitelisting validates only that an implementation reports the expected factory, not its behavior, and blacklisting is advisory rather than enforced on-chain. The factory owner is trusted to whitelist only correct, audited implementations and to preserve immutable compatibility across versions, since each version bakes in its own immutables while migration retains prior storage.

Additional Considerations

Oracle: Valuation of the token-to-redeem, and of pending requests in the compounding account, depends on the account's oracle. The oracle is assumed to be accurate, non-manipulable, and operating normally. A wrong price misstates net asset value.

Asset and redemption-token pricing: The oracle prices the token-to-redeem against a shared quote (for example, US dollars), and the code treats that quote as asset-denominated, so the vault asset is assumed to be worth exactly one quote unit (for example, a US-dollar stablecoin at par). When an account's fallback redemption token differs from the vault asset, it is valued one-to-one after a decimals adjustment with no price applied. A de-peg of the asset, or a value divergence between the redemption token and the asset, misstates net asset value for as long as the position is held.

Midas integration: The integrated Midas redemption vaults are assumed to keep the non-instant redemption fee at zero, since net asset value does not subtract a redemption fee from held or pending inventory. The redemption-request lifecycle is assumed to behave as modeled. In particular, the accounting assumes a rejected request returns the mToken, which may not match on-chain behavior and is worth confirming. Conversions run a synchronization step first, so a paused or reverting redemption vault blocks them, which is a liveness coupling rather than a loss vector.

Unbounded request set: Each account stores its outstanding redemption requests in an array that is iterated in full by valuation, finalization, and synchronization, and aggregated across all accounts by the adapter. These iterations are assumed never to revert from out-of-gas, which holds only while the count of simultaneously pending requests stays low. That in turn depends on the cooldown rate-limiting new requests and on the Midas vault resolving requests at least as fast as they are created.

Token behavior: Tokens-to-redeem and tokens accumulated for conversion are assumed to be standard ERC-20 tokens without transfer callbacks, reentrancy hooks, or fee-on-transfer and rebasing behavior, none of which the swap path or CoW settlement handle. The accounting also does not guard the case where a token-to-redeem is configured to equal the vault asset, although the symmetric redemption-token case is guarded, so the curator is assumed not to configure such a pairing.

Signature validation: Signers that are smart contracts, or externally owned accounts using EIP-7702 delegation, must be compatible with the signature-validation method the adapter uses for delegated and discount instruments.

Deployment: The deployer must configure the genuine CoW Protocol settlement contract, since the vault relayer that receives a maximum token approval is derived from it.

Medium Severity

M-01 Token-to-Redeem Balances Below the Redemption Vault Minimum Freeze `sync` and `convert`

The `_sync` function of the `CooldownAccount` contract finalizes existing redemption requests and then, once the time gate is satisfied and the held token-to-redeem balance is non-zero, submits the entire balance to the Midas redemption vault through `_requestRedeem`. Midas enforces a per-token `minAmount` and reverts any request below it, unless the account is registered in the vault's permissioned `isFreeFromMinAmount` mapping, yet the gate only checks that the balance is non-zero rather than that it reaches `minAmount`.

A sub-`minAmount` balance therefore reverts the request, and because finalization and submission share the same call, the entire call reverts. This bricks the permissionless `sync`, `convert` (which calls `sync` before creating an order), and any swap whose post-transfer balance remains below the minimum, since `_swap` calls `sync` unconditionally. While the balance stays below the minimum, there is no in-contract recovery, as `convert` rejects the token-to-redeem and no sweep exists; the state clears only once a later swap or transfer lifts the balance above the minimum. The owner is the most affected party, because `msg.sender == owner()` satisfies the time gate unconditionally, so every owner-initiated `sync` or `convert` retries the reverting request. The condition arises during normal operation when a small amount reaches the account, and when the token is freely transferable an attacker can donate a negligible amount to freeze it deliberately.

Consider gating the submission on a viable amount rather than on a non-zero balance. Because the Midas per-token `minAmount` is not exposed by the in-scope redemption-vault interface, wrap the `_requestRedeem` call in a `try/catch` so that a sub-minimum or otherwise rejected request is skipped instead of reverting `_sync`, and therefore `sync`, `convert`, and swaps. On a skipped request, leave the inventory held so that it stays counted in the net asset value, and do not advance `lastRequestTimestamp`, so the account retries the submission automatically the next time the balance crosses the minimum.

Update: *Partially Resolved at commits 3df5573 and 5c0b610. The team stated:*

> *We are not making sync() to not revert for such minAmount cases, but have: (1) Added "isRequestCreated" to CooldownAccount to have such possibility for other Accounts (for Midas - seems safer to revert than assume that the revert was caused by exactly minAmount case and not by some worst case scenario; while it doesn't break any, e.g., claim logic) (2) Made convert() to not revert because of this.*

M-02 In-Flight Swap Balances in `LiquidLaneAdapter` Can Be Stolen by Arbitrary Callers

The `LiquidLaneAdapter` contract settles instant redemptions through three `swap` overloads, all routed into the internal `_swap` function. Rather than pulling the input from the caller, `_swap` spends whatever `tokenIn` balance the adapter already holds and pays `amountOut` of the vault asset to a caller-supplied `recipient`, assuming the redeemer or filler has already moved `tokenIn` in. Only the `swap(SignedSwap, …)` path binds the execution to its funder, requiring `caller == msg.sender` and signing `recipient` and `amountIn`; the `swap(DiscountSwap, …)` overload validates only `discount.signer` and takes `recipient` and `amountIn` as raw, unsigned arguments.

Because `_swap` never associates the spent `tokenIn` with the caller, any `tokenIn` the adapter holds is consumed by whoever calls `swap` next. Whenever `tokenIn` is moved in without an atomic same-transaction swap (a non-atomic deposit, a pre-positioned filler balance, a mistaken transfer, or a swap waiting in the mempool), it becomes a grabbable float: an attacker calls `swap` with `recipient` set to itself and `amountIn` set to the float, routing the victim's `tokenIn` into the account and collecting the discounted `amountOut`. Through `DiscountSwap` this needs no privilege, since any holder of a still-valid signer- and protocol-cosigned discount can call it. The swap rate floor keeps the vault neutral-to-positive, so no depositor principal is at risk and the entire loss falls on the float owner. The reusable discount is not itself the flaw, but its replayability lets a single observed discount drain successive floats until it expires or is revoked.

Consider binding both the funding source and the recipient to the executor at call time on every overload, so that a swap can only ever spend the caller's own funds and pay the caller's own recipient. For the direct and discount paths, pull `tokenIn` from `msg.sender` through `transferFrom`; for the signed path, pull from the signer or from an explicit signed funder field, consistent with its delegated-`caller` design. To stop a residual float from being grabbed by the next swap, have the swap consume only the freshly pulled amount rather than any pre-existing adapter balance, for example by measuring the balance delta. Note that this

shifts the integration flow from transfer-then-swap to approve-then-swap and requires updating integrators accordingly.

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

> *We assume the swap() is called atomically.*

M-03 BTC and ETH Midas mTokens Undervalued in `Account.totalAssets`

The Midas accounts value their held and pending token-to-redeem inventory through the `MidasOracle` contract, whose `getPrice` function forwards the mToken data feed's `getDataInBase18` unmodified. The `Account` contract then treats that value as the token-to-redeem price denominated in the vault asset, which is USDC for every shipped account, and this underpins `totalAssets` and, for compounding accounts, the live valuation of pending requests.

The BTC- and ETH-based mTokens, however, do not feed a USD price: their data feeds return the mToken's value in its own underlying asset (the mToken-to-BTC or mToken-to-ETH rate, near `1e18`), which `Account` misreads as a USDC price. One mBTC, worth roughly 106,000 USD, is therefore valued at about 1.03 USDC, as are mHyperBTC, mevBTC, mRe7BTC, and mHyperETH, so `totalAssets` underreports the position by roughly five orders of magnitude for the BTC tokens and three for the ETH token whenever such inventory or a pending request is held. This understates the share price, dilutes incumbent depositors, and opens an entry-and-exit basis against them, though the exposure is bounded because such inventory accrues through curator funding or a direct transfer rather than a profitable attack (a loading swap credits the mToken in and pays out the same undervalued asset amount). The behavior was confirmed on Ethereum mainnet, where the control token mTBILL returns a USD-consistent value while the BTC and ETH feeds return a near-unity ratio. The test suite does not surface the divergence, since its mock data feed returns a USD-consistent value for every mToken.

Consider resolving the price to the vault-asset numeraire inside the in-scope `MidasOracle` contract, by composing the mToken-to-underlying data feed with a BTC/USD or ETH/USD feed so that `getPrice` returns a USD-denominated value. The added feed should carry the same staleness and zero-price handling as the existing path so it does not introduce a further valuation revert. In addition, add a magnitude sanity check at onboarding rather than only asserting that the price is non-zero, so a wrong-numeraire feed fails fast at deployment, and update the mock data feed in the test suite to return underlying-denominated values for the BTC and ETH feeds so the divergence is exercised.

Update: *Resolved at commits 2a74be9, 6de7ece and 1b0816e.*

M-04 BTC and ETH mTokens Cannot Be Redeemed, Bricking Swaps and Stranding Inventory

When a swap forwards a token-to-redeem to its account, the `_swap` function unconditionally calls `sync`, which for Midas accounts submits the held balance through `_requestRedeem`. That function selects an output token: the vault asset when the Midas vault configures it, and otherwise the account's `REDEMPTION_TOKEN`, which is hardcoded to USDC in every shipped account.

On the Midas vaults for the BTC- and ETH-based mTokens, USDC is not a configured payment token, so `redeemRequest` reverts with `"MV: token not exists"`. Because `_requestRedeem` always passes USDC for these tokens, every request reverts, and since `_swap` calls `sync` unconditionally, the first swap forwarding such an mToken reverts in full, as do permissionless `sync` and `convert` for that token. Instant redemption is therefore permanently non-functional for these tokens. The test suite does not catch this, because its mock Midas redemption vault is configured to accept USDC, so requests succeed under test while the real mainnet vaults, which do not configure USDC, would revert.

Consider pairing these mTokens only with a vault whose asset the corresponding Midas vault configures, for example a tBTC or WETH vault, which also resolves the related undervaluation finding; in that case the hardcoded `REDEMPTION_TOKEN` should be revised, since it would otherwise remain an incorrect fallback. Do not instead repoint `REDEMPTION_TOKEN` to a configured non-asset token such as tBTC or WETH while keeping a USD-denominated vault asset: `_redemptionTokenToAssets` values the held redemption-token balance one-to-one with the vault asset and applies no price, which would trade the revert for a silent and severe net-asset-value misstatement. To catch a misconfiguration at deployment, place the non-zero data-feed assertion in `_initialize`, where the vault asset is known, and check the token that `_requestRedeem` will actually submit, namely that `tokensConfig(_asset).dataFeed` or `tokensConfig(REDEMPTION_TOKEN).dataFeed` is non-zero, rather than in the account constructor, which cannot read the vault asset and would over-constrain `REDEMPTION_TOKEN`. Finally, guard the unconditional `sync` in `_swap`, for example with a `try/catch` around the submission, so that other Midas-side preconditions cannot brick the swap path.

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

> *The deployment configs are not final/validated in general, but, thank you, it's really helpful.*

M-05 Predictable Jumps in `totalAssets` Allow Timed Value Extraction From Passive Holders

Each liquidity-lane account reports its value through `totalAssets`, which the adapter aggregates as its net asset value. This live figure mixes components valued on different bases, so as inventory moves through the redemption lifecycle the basis changes and produces predictable, stepwise jumps:

- Held token-to-redeem is valued at the gross oracle rate. In the non-compounding account a pending request is then valued at the amount and rate locked at creation, so submitting through `_requestRedeem` steps the valuation; in the compounding account it is valued at the current oracle rate, so submission is approximately continuous and the step instead arises later from rate drift.
- A pending request is valued at its marked rate until it is processed, when it is replaced by the realized payout; any gap between the mark and the realized amount steps the valuation at processing.
- A rejected or canceled request drops out of the pending valuation, stepping it down to the extent the token-to-redeem is not returned.
- Once a Midas redemption fee is configured, held token-to-redeem is overvalued relative to the net realized amount, adding a downward correction at processing.
- A convertible balance outside the components above (a reward or airdrop token, or residual `tokenIn` left by a partially fillable CoW order) is omitted from the valuation until a `convert` order is filled by a CoW solver, which steps `totalAssets` up by the realized amount; the on-chain `convert` only pre-signs the order, so the step occurs at the later solver fill.

Each step is deterministic and observable, since it is driven by an on-chain transaction, namely the permissionless request submission, the Midas processing or rejection, or a `convert` settlement. Wherever `totalAssets` prices share entry and exit, a participant can deposit before an upward step or withdraw before a downward step, capturing the discontinuity at the expense of the remaining holders.

Consider accounting for the Midas redemption fee in `totalAssets`, by subtracting the per-token fee reported by `tokensConfig` from the held and pending token-to-redeem valuation, so that the fee no longer produces a downward step when a request is processed. The convertible-balance step, where a reward or airdrop token or residual `tokenIn` only enters

`totalAssets` once a CoW conversion settles it, is not addressed: valuing such a balance before settlement would require a reliable oracle for every convertible asset, which is not practical, so it is left as an accepted limitation.

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

> *We'll be implementing related softening mitigations on the vault's side in future.*

M-06 Deallocation Reverts on a Frozen Account, Bricking Vault-Wide Operations

The `deallocate` function of the `LiquidLaneAdapter` contract iterates over every account and, for each one holding a non-zero vault-asset balance, performs a `safeTransferFrom` from the account to the adapter. The transfer has no `try/catch` and no per-account skip, so a revert on any single account reverts the whole call, even though the `IAdapter` interface documents `deallocate` as a function that must not revert.

The vault asset is expected to be a stablecoin with blacklisting and pause controls. If the issuer freezes or blacklists a single account, or pauses transfers globally, that account's `safeTransferFrom` reverts and reverts the entire `deallocate`. Because the delegator's `sweepPending` calls `deallocate` for every adapter on deposits, withdrawals, and every swap (each swap allocates through the delegator and triggers a sweep), a freeze on one account reverts deposits, withdrawals, and swaps across the whole vault until it clears. No funds are lost, since the balances stay escrowed, but the curator cannot even retire the affected token, because `removeTokenToRedeem` requires the account's `totalAssets` to be zero. Every other adapter wraps its external allocation and deallocation calls in a `try/catch` and returns zero on failure; the `LiquidLaneAdapter` is the only one whose `deallocate` can revert.

Consider wrapping each per-account `safeTransferFrom` in a `try/catch` and skipping any account whose transfer reverts, so that a single frozen account cannot block deallocation of the others. This honors the documented must-not-revert behavior and matches the pattern the other adapters already follow.

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

> *Currently the policy of reverting during any undefined scenarios seems safer.*

M-07 Single Token Oracle Failure Can Brick the Vault With Self-Locked Recovery

Each liquidity-lane account values its inventory through an oracle: the `totalAssets` function reads it whenever the account holds token-to-redeem inventory, and `MidasCompAccount` also reads it to value pending requests. The `_tokenToRedeemToAssets` function reverts with `InvalidOracle` on a zero price, and the `getPrice` function of `MidasOracle` forwards `getDataInBase18` without a `try/catch`, so a stale, unhealthy, or deprecated feed reverts. The adapter then aggregates each account's value as a bare sum, with no per-account isolation.

This valuation feeds the vault's accounting unconditionally: `VaultV2.getAccrueInterest` sums the free assets and `UniversalDelegator.totalAssets` (itself a bare sum over every adapter), and `accrueInterest` runs at the start of every deposit, mint, withdraw, and redeem. A single token whose oracle returns zero or reverts therefore reverts the account's `totalAssets`, then the adapter's, then the delegator's, freezing share-price computation, fee accrual, deposits, and withdrawals across the whole vault and every sibling adapter on the delegator. No privilege or attacker is required, since any oracle staleness, zeroing, or feed deprecation triggers it, and an attacker can time a known staleness window. Recovery is self-locked: `removeTokenToRedeem` reads the offending account's `totalAssets`, and the delegator's `removeAdapter` and `forceDeallocate` read the adapter's, so all three escape hatches revert on the same fault, making the freeze effectively permanent for a delisted feed. Every other adapter values its position from token balances alone, reads no price oracle, and wraps its external calls in a `try/catch`, and the `IAdapter` interface documents the deallocation path as a function that must not revert; the `LiquidLaneAdapter` is the only adapter that introduces a reverting accounting read.

The reverting behavior fails closed, protecting value at the cost of availability, so the remediation should preserve that property rather than paper over the missing price. Substituting a fabricated value, such as zero or the last-known contribution, to keep the vault live would be unsafe: it would step the account's reported value down for the full position while the feed is faulting and back up on recovery, letting a depositor enter at the depressed valuation and redeem at the restored one, which is the timed value extraction described in a separate finding and a strictly worse outcome.

Consider skipping the oracle read entirely when the account holds no inventory or pending value to price, so an idle token can never brick valuation; decoupling `removeTokenToRedeem` from a reverting `totalAssets`, so a faulty token can always be retired rather than self-locked; and hardening `MidasOracle` against transient feed faults. While a held token's oracle is genuinely down, pausing entry and exit, the current outcome, is

preferable to minting or burning shares against an unknown valuation, so the priority is guaranteeing an unbricked recovery path rather than restoring liveness by inventing a price.

Update: *Partially Resolved at commits 1b0816e and 2681202.*

Low Severity

L-01 Mixed Namespaced and Sequential Storage Risks State Corruption on Migration

The adapter and account contracts sit behind ERC-1967 proxies, and the account contracts are migratable through `MigratableEntity.migrate`. They mix ERC-7201 namespaced storage, inherited from `OwnableUpgradeable`, `Initializable`, and `PausableUpgradeable`, with plain sequential storage from non-upgradeable OpenZeppelin bases and from their own sequentially declared variables; the `__gap` arrays exist only because that own state is not namespaced.

No live issue exists today, since keccak-derived namespaced slots never overlap the sequential slots counted from zero. The risk is structural and surfaces on migration, the operation this hierarchy exists for: with sequential storage only appends to the most-derived contract are layout-safe, so inserting or reordering any base-layer variable shifts every slot below it, and after `upgradeToAndCall` the new implementation reads and writes existing values under the wrong names, corrupting state such as `requestIds`, `acquireBalance`, or `vault`. The `__gap`s mitigate this only partially and only for the layer that owns them. The account chain is the most exposed, since only `MigratableEntity.__gap[10]` separates it from the bases while `Account`, `CooldownAccount`, and `MidasAccount` add no gaps of their own.

The non-upgradeable bases that consume sequential slots are:

- `Nonces` in `CoWSwapConverter.sol` (`_nonces`, slot 10).
- `EIP712` in `LiquidLaneAdapter.sol` (`_nameFallback` and `_versionFallback`, slots 0 to 1).

The own state variables declared sequentially rather than in an ERC-7201 struct are:

- `vault` in `Adapter.sol` (slot 12).
- `adapter`, `vault`, `_asset`, and `_unit` in `Account.sol` (slots 11 to 14).

- `lastRequestTimestamp` in `CooldownAccount.sol` (slot 15).
- `requestIds` in `MidasAccount.sol` (slot 16).
- `pauser` through `_isTokenToRedeem` in `LiquidLaneAdapter.sol` (slots 63 to 74).

Consider moving the own state into ERC-7201 namespaced structs, after which the `__gap`s become unnecessary, and replacing the non-upgradeable bases that consume sequential slots, namely `Nonces` with `NoncesUpgradeable` and `EIP712` with `EIP712Upgradeable`. Because this re-layout is itself breaking, apply it only to proxies deployed for the first time, never to an already-deployed instance, which it would orphan. Switching to `EIP712Upgradeable` is also worthwhile as upgrade-safety hygiene and to drop the redundant immutable domain-separator cache, though the in-flight-signature risk does not currently arise: `EIP712` is used only by the non-migratable `LiquidLaneAdapter`, and its cache already misses under the proxy and rebuilds the separator correctly. Consider as well moving `ReentrancyGuard` to `ReentrancyGuardTransient` for gas, as part of the same re-layout rather than piecemeal.

Update: *Resolved at commits 18c66c5, 9410d11 and fa4d485.*

L-02 Directly Deployed `CoWSwapConverter` Is Permanently Ownerless

The `CoWSwapConverter` contract is declared concrete rather than `abstract`, so it is independently deployable, yet it has no owner-initialization path of its own: its `__CoWSwapConverter_init` function sets the converters but never calls `__Ownable_init`, and the owner is otherwise set only by `MigratableEntity.initialize`, which a standalone instance does not inherit. A directly deployed instance is therefore never initialized, so `owner()` stays `address(0)` and the converter set stays empty, leaving `setConverters` (owner-gated) and `invalidateCovert` (converter-gated) permanently uncallable. There is no protocol impact, since the contract is always used through `Account`, but the concrete declaration is a latent hazard.

Consider marking `CoWSwapConverter` as `abstract`. Every consumer inherits it and no code deploys it directly, so the change is fully compatible and removes the standalone-deployment footgun.

Update: *Resolved at commit af3586f.*

L-03 Dust Donations Block Token-to-Redeem Removal

The `removeTokenToRedeem` function of the `LiquidLaneAdapter` contract reverts with `AccountHasAssets` whenever `account.totalAssets() > 0`, a value that includes the account's freely transferable vault-asset balance. Anyone can therefore send one wei of the asset, the token-to-redeem, or the redemption token to keep `totalAssets()` above zero and block removal. A donated asset balance is only a transient block, since `sweepPending` calls `deallocate` on every deposit, withdrawal, and swap, sweeping the account's full asset balance out; the durable vector is a one-wei token-to-redeem or redemption-token donation, which `deallocate` does not move and which clears only through `sync` or `convert`. Because only the owner can add tokens, an attacker cannot force the `MAX_TOKENS_TO_REDEEM = 50` cap to fill, and the owner can already neutralize a token without removing it by setting its `limit` to zero.

Consider gating the `AccountHasAssets` check only on non-recoverable value, namely token-to-redeem inventory and pending requests, and draining the recoverable asset balance to the vault before removal rather than reverting on it. Do not simply drop the check: removing a token whose account still holds an asset balance, inventory, or a pending request would orphan that backing, since `totalAssets`, `freeAssets`, and `deallocate` all iterate only the active token set, so the value would leave the net asset value and become unrecoverable. If a dust threshold is used instead, keep it negligibly small.

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

> *We will keep it as it is for now.*

L-04 Token-Scoped Swap Nonces Allow Cross-Signer Collisions and Cancellation

The `LiquidLaneAdapter` contract tracks swap-signature nonces in `isUsedNonce[tokenToRedeem][nonce]`, keyed by token only and shared across the `SignedSwap` and `DiscountSwap` paths and across every signer (owner, market maker, and all fillers), unlike per-signer designs such as Permit2 and Seaport. This token-only scoping has several weaknesses:

- Two signers may pick the same nonce for the same token; the first execution to consume it silently invalidates the other's signature, which the signers cannot coordinate away off-chain.

- The `invalidateNonce` function lets any current filler mark any `(token, nonce)` pair as used, so a rogue filler can cancel another signer's pending quote or pre-burn a nonce another signer intends to use. Per-signer scoping is what normally prevents this.
- The namespace is shared between the two schemes, so a `SignedSwap` nonce and a `Discount` nonce can collide, and the `DiscountSwap` path checks the nonce but never sets it (intentionally, since the discount is reusable), so the two schemes share one space under inconsistent consumption rules.
- Each nonce occupies a full slot through a `mapping(uint256 => bool)`, costing a cold `SSTORE` per quote with no reclamation, and revocation is per-nonce only, with no mass-cancel primitive.

Consider scoping the nonce by signer as well as token, for example `isUsedNonce[signer][tokenToRedeem][nonce]`, and updating `invalidateNonce` in the same change to take the signer dimension and restrict each caller to its own `(signer, token, nonce)` keys; otherwise a filler could still cancel another signer's nonce and the core issue would remain. The per-signer key is safe for the reusable `DiscountSwap` path, which never writes the nonce. Consider also moving to incremental nonces to reduce storage cost and to support mass invalidation through a per-signer nonce floor.

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

> *We assume that different signing addresses coordinate as a single entity.*

L-05 Missing Zero-Address Validation on Address Parameters

Across the adapter family, several constructors and the entity initializer assign critical addresses without checking them against the zero address. The most consequential is the entity owner: `MigratableEntity.initialize` sets an owner only when `owner_` is non-zero (since `__Ownable_init` reverts on zero), and the permissionless `create` function forwards `owner_` unvalidated. An entity created with a zero owner has no owner, so every `onlyOwner` function, the entire `LiquidLaneAdapter` administrative surface, is permanently uncallable; it also cannot be migrated, since `migrate` requires the caller to be the unreachable zero owner, nor recovered, since `transferOwnership` is owner-gated. The intended path is unaffected, as `LiquidLaneAdapter` creates its accounts with its own non-zero owner, but any integration calling `create` with the zero address produces a frozen, unrecoverable entity.

The remaining genuine cases are set-once addresses whose zero value is not caught at deployment and instead breaks the contract on first use:

- `VAULT_FACTORY` in the `Adapter` constructor, used to validate vaults during initialization.
- `ORACLE` in the `Account` constructor, without which all token-to-redeem valuation reverts.
- `REDEMPTION_TOKEN` and `REDEMPTION_VAULT` in the `MidasAccount` constructor, without which redemption requests and finalization revert.
- `DATA_FEED` in the `MidasOracle` constructor, without which `getPrice` reverts.
- `FACTORY` in the `MigratableEntity` constructor, without which the entity can never be migrated.

The other flagged parameters do not warrant a check: a zero `cowSwapSettlement` or `tokenToRedeem` already reverts at construction, because the constructors immediately call `vaultRelayer` and `decimals` on them, and the conversion functions' `tokenIn` and `tokenOut` are transient inputs already rejected by the existing token validation or by the subsequent balance and transfer calls.

Consider reverting on a zero value at the point of assignment for each of the set-once addresses listed above. For the owner, enforce a non-zero `owner_` in the deployment path that calls `create`, rather than in the intentional zero-owner branch of `initialize`, so the guard prevents a frozen entity without interfering with the framework's deliberate handling.

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

> *This is acknowledged.*

L-06 Asymmetric State Handling Between Adding and Removing a Token-to-Redeem

The `LiquidLaneAdapter` contract tracks each token-to-redeem through an active set (`tokensToRedeem`, `_isTokenToRedeem`, `limit`) and persisted per-token mappings (`accounts`, `minDiscount`, `acquireBalance`, `isUsedNonce`). `addTokenToRedeem` populates both and creates the account, but `removeTokenToRedeem` clears only the active set, leaving the mappings intact. Functions that iterate the active array gate on membership; those that read the mappings do not, producing several inconsistencies:

- `getMaxAssets`, `getMaxRate`, and `getAmountOut` do not gate on `_isTokenToRedeem`, so they return live quotes (including a stale `acquireBalance`)

for removed tokens that `swap` would reject with `InvalidTokenToRedeem`, describing capacity that cannot be realized.

- `setMinDiscount` and `depositToAcquire` lack the membership gate that `setLimit` has, so a discount floor and acquire balances can be set on inactive tokens and survive removal. A persisted `acquireBalance` is benign, since an inactive token cannot be swapped and the balance stays withdrawable through `withdrawToAcquire`.
- `removeTokenToRedeem` reads `accounts[token].totalAssets()` before the membership check, so a never-added token reverts with a low-level call failure rather than `InvalidTokenToRedeem`.
- Since the account is created only when `accounts[token]` is zero and removal never clears it, re-adding a token reuses the original account and its persisted `minDiscount`; only `limit` is reset, so the token resumes at that floor. Because `minDiscount` is a minimum, the persisted value is more protective than a fresh account's zero floor, so this is a configuration surprise to re-verify on re-add rather than a loss of protection.
- Assets that reach a removed account are excluded from `totalAssets`, `freeAssets`, and `deallocate`, so they are stranded until the token is re-added.

The persistence of `accounts[token]` also fixes account ownership: each account's owner is set to `owner()` at creation and never re-synchronized, so a single `transferOwnership` on the adapter orphans control of every account at once. The former owner retains each account's owner-gated actions, the CoW converter allowlist via `setConverters` and migration via the factory, and could, for example, register a converter to dump redemption proceeds at an unfavorable rate or withhold a migration; the converter allowlist compounds this, being seeded with the creation-time owner and not coupled to `owner()`.

Consider making the lifecycle symmetric: gate the token-parameterized view functions on `_isTokenToRedeem` so they do not quote inactive tokens, re-verify `minDiscount` on re-add, and evaluate the membership check before the `totalAssets()` read in `removeTokenToRedeem` (also validating that the `AccountRegistry` returns a configured factory) so invalid inputs revert with meaningful errors. The account-ownership and reuse model should be decided and documented explicitly; the cleanest option is to own the accounts from the adapter itself, after which the adapter can mediate the converter allowlist and migrations and re-seed them on an ownership change. Note that rotating each account's owner on `transferOwnership` is not possible while the adapter is not the account owner, and that clearing `accounts[token]` on removal to force a fresh account is safe only together with adapter-owned accounts, since otherwise any balance later sent to the old account address becomes permanently unreachable.

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

> *We gate/zero only a minimal set of functionality to have faster params recovery on re-adding the asset.*

L-07 Acquire-Funded Swaps Revert When the Account Exceeds Its Limit

A swap in the `LiquidLaneAdapter` contract is funded from two sources: prefunded acquisition balances from the curator or market maker, and a fresh vault allocation. The `_swap` function splits the output into an acquisition-funded part and a vault-funded `tokenOutToAllocate`, then enforces the per-token limit with `account.totalAssets() + tokenOutToAllocate > limit[tokenIn]`. The limit is meant to bound vault-funded exposure, which grows only along the allocation path, yet the check runs unconditionally, including when `tokenOutToAllocate` is zero.

A fully acquisition-funded swap allocates nothing from the vault and leaves the account's `totalAssets()` unchanged, so the limit is irrelevant to it. Even so, if the account's value already exceeds the limit, reachable through appreciation of held tokens, the discount margin accumulated by prior allocation-funded swaps, or a `setLimit` reduction, the check evaluates `totalAssets() + 0 > limit[tokenIn]` as true and reverts with `LimitExceeded`. Acquisition liquidity, meant to be independent of the limit, is therefore unusable precisely when the account is at or beyond it, which is when it is most needed. The behavior also diverges from `getMaxAssets`, which reports the acquisition balance as available output when the account is over its limit, so the on-chain quote misleads integrators into attempting swaps that revert. No funds are at risk.

Consider evaluating the limit check only when `tokenOutToAllocate` is greater than zero, so it applies solely to swaps that increase vault-funded exposure. A swap with `tokenOutToAllocate` equal to zero leaves the account untouched, so skipping the check is safe and realigns `_swap` with what `getMaxAssets` already reports.

Update: *Resolved at commit e0f3ef5.*

L-08 Per-Token Exposure Cap Is Overshot by the Swap Discount Margin

The `LiquidLaneAdapter` contract enforces a per-token concentration cap, `limit[tokenIn]`, intended to bound the vault's exposure to a single token-to-redeem. The check in `_swap` is `account.totalAssets() + tokenOutToAllocate >`

`limit[tokenIn]`, where `tokenOutToAllocate` is the vault cash funding the swap. The swap, however, routes the token-to-redeem into the account at its full oracle value, and because the rate floor requires `amountOut <= oracleValue * (1 - minDiscount)`, the booked value is `tokenOutToAllocate / (1 - d)`, strictly above the cash charged. The check therefore under-counts the exposure it bounds, letting `totalAssets()` settle above `limit` by up to `tokenOutToAllocate * d / (1 - d)`. For example, with `limit = 10,000`, an empty account, and `minDiscount = 2%`, a swap funding `10,000` of cash passes (`0 + 10,000 <= 10,000`), routes token-to-redeem worth `10,000 / 0.98 = 10,204`, and leaves `totalAssets()` at `10,204`, exceeding the cap by `204`. The overshoot is bounded, as the next swap reads the higher `totalAssets()` and reverts, and the excess is acquired at a vault-favorable price, so this is a breach of the concentration cap rather than a loss of funds. The `getMaxAssets` quote shares the same cash-versus-value basis.

Consider charging against `limit` the value that will actually be booked, computed from the post-acquire residual as `getAmountOut(tokenIn, tokenInToRedeem)`, rather than the raw vault cash. That figure already excludes the acquisition-funded portion and reflects the discount actually applied; deriving it instead from the full `amountIn`, or from a fixed `1 / (1 - minDiscount)` factor, would over-count acquisition-funded swaps or under-count swaps whose applied discount exceeds the minimum.

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

> *Limits mostly exist to limit allocations exposure, not the profit.*

L-09 `MidasNonCompAccount` Pending-Request Valuation Ignores the Recorded `tokenOutRate`

The non-compounding Midas account, `MidasNonCompAccount`, prices each pending redemption request at the rates locked at creation. Its `_pendingAssets` function reads the request's `amountMToken` and `mTokenRate` and values it through `_tokenToRedeemToAssets`, scaling `amountMToken * mTokenRate` into the vault-asset decimals. The Midas vault, however, records two rates and returns both: `mTokenRate`, the redeemed token's price, and `tokenOutRate`, the output token's price, with the eventual payout proportional to `amountMToken * mTokenRate / tokenOutRate`. Because the trailing element of the destructuring is left unnamed, `_pendingAssets` discards `tokenOutRate` and omits the division, which is equivalent to assuming `tokenOutRate == 1e18`. The assumption holds for the current deployments, whose output tokens are US-dollar stablecoins near one dollar, so the defect is dormant; it is latent, though, because any

onboarded token whose `tokenOutRate` deviates from `1e18` (a non-stable output token or a depegged stablecoin) would misstate the pending valuation, and therefore `totalAssets` and the share price, by the ratio of `tokenOutRate` to `1e18`. The request struct already carries the exact `tokenOutRate` needed.

Consider mirroring the Midas payout formula by rescaling the valuation by `1e18 / tokenOutRate` rather than dividing by the raw `tokenOutRate`, which is itself an 18-decimal value and would collapse the result to near zero if divided directly; the division should also guard against a zero `tokenOutRate` to avoid introducing a valuation revert. Alternatively, assert that `tokenOutRate` equals `1e18` during valuation or token onboarding, so the assumption is enforced rather than left implicit.

Update: *Resolved at commit 15d9f30.*

Notes & Additional Information

N-01 Typographical Error in Identifier Names

Consistent and correctly spelled identifiers improve the readability and maintainability of the codebase. The function and event used to invalidate a CoW Protocol conversion order are misspelled as `Covert` instead of `Convert`:

- The `invalidateCovert` function in `CoWSwapConverter.sol`.
- The `InvalidateCovert` event emission in `CoWSwapConverter.sol`.

Both identifiers are declared in the `ICoWSwapConverter` interface, as the `InvalidateCovert` event and the `invalidateCovert` function.

Consider renaming these identifiers to `invalidateConvert` and `InvalidateConvert` respectively, updating all references accordingly.

Update: *Resolved at commit 6de7ece.*

N-02 Floating and Inconsistent Pragma Directives

Pragma directives should be fixed and consistent across the codebase to clearly identify the Solidity version with which the contracts will be compiled. Locking the pragma helps ensure that contracts are not accidentally deployed using an outdated or unintended compiler version that could introduce bugs or affect the system negatively.

Throughout the codebase, multiple instances of floating pragma directives were identified:

- The `solidity ^0.8.28` directive in `Adapter.sol`.
- The `solidity ^0.8.28` directive in `LiquidLaneAdapter.sol`.
- The `solidity ^0.8.28` directive in `CoWSwapConverter.sol`.
- The `solidity ^0.8.28` directive in `AccountRegistry.sol`.
- The `solidity ^0.8.28` directive in `MidasAccount.sol`.
- The `solidity ^0.8.28` directive in `Account.sol`.
- The `solidity ^0.8.28` directive in `CooldownAccount.sol`.
- The `solidity ^0.8.28` directive in `MidasOracle.sol`.
- The `solidity ^0.8.25` directive in `MigratableEntity.sol`.

Furthermore, these directives are not consistent across the codebase. The `MigratableEntity.sol` contract declares a `pragma solidity ^0.8.25;` directive, whereas the remaining in-scope contracts declare a `pragma solidity ^0.8.28;` directive. This inconsistency is also present across imports. The `Adapter.sol` contract imports the `MigratableEntity.sol` file, and the `Account.sol` contract imports it as well, in both cases pulling in a file with a different pragma directive than the importing contract.

Consider using the same fixed pragma version across all files to clearly identify the Solidity version intended for compilation and deployment.

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

> *This is acknowledged as legacy support.*

N-03 Possible Duplicate Event Emission

When a setter function does not check whether the new value differs from the existing one, it is possible to set the same value repeatedly. This creates a potential for event spamming, and repeated emission of identical events can confuse off-chain clients that rely on those events to track state.

Throughout the codebase, several setter functions update state and emit an event without first verifying that the new value differs from the current one:

- The `setReceiver` function emits `SetReceiver`.
- The `setFiller` function emits `SetFiller`.
- The `setMarketMaker` function emits `SetMarketMaker`.
- The `setLimit` function emits `SetLimit`.
- The `setMinDiscount` function emits `SetMinDiscount`.
- The `setPauser` function emits `SetPauser`.
- The `setUnpauser` function emits `SetUnpauser`.
- The `setConverters` function emits `SetConverters`.

Consider skipping the state write and event emission with an early return when the new value equals the current one, rather than reverting, so that idempotent re-affirmation calls common in deployment scripts and batched transactions are not broken. The comparison should use the meaningful key in each case: `setFiller` should compare the `(marketMaker, filler, status)` tuple rather than `status` alone, since the emitted `marketMaker` can differ between calls; `setConverters` compares a dynamic array, which requires a full or hashed comparison; and `setReceiver` is permissionless, so any added control should avoid creating a new revert surface in batched flows.

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

> *This is acknowledged.*

N-04 Inconsistent Use of Returns in Functions

To improve readability, functions should use a consistent return style: either declare a named return variable and assign to it, or declare an unnamed return type and use explicit `return` statements, but not mix both approaches.

Throughout the codebase, the following functions use returns inconsistently:

- The `executableAt` function declares a named return variable `timestamp` but returns the value through an explicit `return` statement, leaving the named variable unused.
- The `getMaxAssets` function declares a named return variable `assets`, assigns to it in some branches, and also returns values through explicit `return` statements in others, combining both styles within a single function.

Consider using a consistent return style within these functions, either by relying on named return variables or by using explicit `return` statements exclusively.

Update: *Resolved at commit dd434d7.*

N-05 Missing Error Message in Revert Statement

Providing informative error messages in `revert` statements improves code clarity and assists with troubleshooting when a requirement is not satisfied.

Within `Adapter.sol`, the `_migrate` function reverts unconditionally to signal that migration is unsupported. However, the `revert` statement does not include an error message, so callers receive no indication of why the call failed.

Consider adding a named custom error such as `MigrationNotSupported` to the `revert` statement to improve the clarity and maintainability of the code.

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

> *This is acknowledged.*

N-06 Functions Updating State Without Event Emissions

Emitting events on state changes allows off-chain services to track the contract state reliably and aids monitoring, indexing, and troubleshooting. Throughout the codebase, several functions update the contract state without emitting a corresponding event:

- The `_initialize` function in `Account.sol` sets the `vault`, `adapter`, `_asset`, and `_unit` state variables without emitting an event. The analogous `_initialize` function in `Adapter.sol` emits a `SetVault` event when setting its vault.
- The `_sync` function in `CooldownAccount.sol` updates the `lastRequestTimestamp` state variable without emitting an event.
- The `_requestRedeem` function in `MidasAccount.sol` appends a new entry to the `requestIds` array without emitting an event.
- The `_finalizeRequests` function in `MidasAccount.sol` removes finalized entries from the `requestIds` array without emitting an event.

Consider emitting events whenever there are relevant state changes to improve the clarity of the codebase and make it less error-prone for off-chain consumers that track these values.

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

> *This is acknowledged.*

N-07 Incorrect or Misleading Documentation

Clear and accurate documentation helps readers understand the intended behavior of the code and reduces the likelihood of errors during maintenance. Throughout the codebase, several comments and NatSpec annotations were identified that do not match the code they describe:

- The `_asset` state variable in `Account.sol`, which holds the vault asset address, is documented as `Vault asset unit`. This is the description that applies to the `_unit` variable declared immediately below it.
- The `_allocate` function in `LiquidLaneAdapter.sol` is documented as triggering adapter allocation through the current delegator. However, the function is declared `pure` and simply returns the `amount` argument. The actual allocation through the delegator is performed in the `_swap` function.
- The `_hash` function in `CoWSwapConverter.sol` documents its return value as `The 32 byte EIP-712 struct hash`. The function actually returns the domain-separated EIP-712 signing digest, which is consistent with the `signing hash` wording in the preceding `@dev` line but not with the `@return` description.
- The `_hashDiscount` function in `LiquidLaneAdapter.sol` documents a named return variable `digest`, but the function declares an unnamed return value.
- A comment in the `deallocate` function of `LiquidLaneAdapter.sol` states that the account's realized asset balance is swept to the vault, whereas the subsequent transfer moves the balance to the adapter itself. The vault subsequently pulls these proceeds through a previously granted approval.
- The `InvalidTokenIn` error in `ICoWSwapConverter.sol` is documented as being raised only when the input token is the vault asset. However, it is also raised when the input token equals the output token in the base `convert` function, and when the input token is the token-to-redeem in the `convert` override of `Account.sol`. The documentation therefore describes only one of the conditions under which the error is raised.

Consider revising these comments and NatSpec annotations to accurately reflect the behavior of the code.

Update: *Resolved at commit aa896e4.*

N-08 Unused Declarations

Removing declarations that are never used reduces the surface area of the codebase, prevents a misleading ABI, and avoids confusion during maintenance. Several interface members are declared but never referenced by any implementation. They appear to be remnants of a superseded order-reservation design that the shipped `CoWSwapConverter` contract, which instead keys orders by nonce and `executableAt` timestamp, no longer uses.

In the `ICoWSwapConverter` interface:

- The `InsufficientSellBalance`, `InvalidNonce`, `OrderNotExpired`, and `UnknownOrder` errors are declared but never reverted with.
- The `ReservedOrder` struct is declared but never used.
- The `ReleaseExpiredOrder` event is declared but never emitted.

In the `IAccount` interface:

- The `NotAdapter` error is declared but never reverted with. The `adapter` address is used only to grant the ERC-20 allowance, never as an access-control gate.

Consider removing these unused declarations, or wiring them into the logic where they were intended to be used.

Update: *Resolved at commits 8d5d01a and 2c5891e.*

N-09 Inconsistent Naming Between Interface and Implementation

Keeping parameter names and signatures consistent between an interface and its implementation improves readability and prevents confusion for integrators who rely on the interface. The following declarations diverge between the interface and the contract that implements them:

- The `setConverters` function in `CoWSwapConverter.sol` names its parameter `newConverters` and declares it as `calldata`, whereas the corresponding declaration in `ICoWSwapConverter.sol` names it `converters` and declares it as `memory`. The function selector is identical, so the behavior is unaffected, but the parameter name and data location disagree.

- The `setFiller` function in `LiquidLaneAdapter.sol` names its boolean parameter `status`, whereas the corresponding declaration in `ILiquidLaneAdapter.sol` and the `SetFiller` event both name it `isAuthorized`.

Consider aligning the parameter names and data locations between each interface and its implementation.

Update: *Resolved at commit b43a1f6.*

N-10 Redundant `_hashDiscount` Recomputation in the Discount-Swap Path

The discount-based `swap` function of the `LiquidLaneAdapter` contract validates two EIP-712 signatures, one from the signer and one from the protocol. To do so, it computes the discount struct hash through the `_hashDiscount` function once while verifying the signer's signature, and again while assembling the payload hashed for the protocol's signature.

Both invocations operate on the same `discount` calldata and therefore produce an identical result. Because `_hashDiscount` performs an `abi.encode` of seven fields followed by a `keccak256`, the second invocation needlessly repeats that work on every discount-based swap.

Consider computing the discount struct hash once, storing it in a local variable, and reusing it for both signature checks.

Update: *Resolved at commit 729ee68.*

N-11 Linear Scan of the `converters` Array Increases Gas Cost of Conversions

The `_isConverter` function of the `CoWSwapConverter` contract performs a linear scan over the owner-set `converters` array and is invoked on every `convert` and `invalidateConvert` call. As the array grows, the per-call gas cost increases.

Consider tracking converter membership with a `mapping(address => bool)` to make lookups constant cost and avoid storing duplicate entries. Note that the interface exposes an indexed `converters(uint256)` getter and a `SetConverters(address[])` event, which a membership mapping alone cannot back, and that `setConverters` replaces the whole set, so a naive mapping would leave stale entries marked as members on each replacement. A

complete change should either retain the array for enumeration and events while adding a parallel membership mapping that is cleared and repopulated on each `setConverters`, or replace the indexed getter with a membership query, which is an interface change. Given the converter set is small and owner-controlled, the practical gas saving is modest.

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

> *This is acknowledged.*

N-12 Redundant `nonReentrant` Modifier on the `migrate` Function

The `migrate` function of the `MigratableEntity` contract is marked `nonReentrant`. However, its internal migration routine already runs under the `reinitializer` modifier, which sets the initializing flag and reverts on any nested initialization or migration, so the entry point is already protected. The reentrancy guard's only distinct effect is to share the reentrancy lock with the contract's other `nonReentrant` functions, such as `sync`. Since the `_migrate` hook performs no external calls in any in-scope implementation, with the accounts inheriting the empty default and the adapters overriding it as a `pure` no-op, there is nothing to reenter during a migration. The modifier therefore adds gas overhead without guarding against any reachable reentrancy, and it needlessly prevents future `_migrate` implementations from calling the contract's other `nonReentrant` entry points.

Consider removing the `nonReentrant` modifier from the `migrate` function and relying on the `reinitializer` guard for protection.

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

> *We prefer not touching legacy.*

Conclusion

The `LiquidLaneAdapter` extends a Symbiotic Core v2 vault with issuer-facing instant redemptions, exchanging real-world asset tokens such as Midas mTokens for the vault asset at an oracle-derived discount and unwinding the redeemed inventory over time through asynchronous Midas redemptions and CoW Protocol conversions. OpenZeppelin audited the adapter, its redemption-account hierarchy, the CoW conversion module, the Midas oracle, and the shared migratable base.

The codebase is compact, and its swap path bounds every payout by a per-token discount floor and an exposure limit, so the vault always receives inventory worth at least the cash it pays out, which keeps depositor principal out of reach of any single privileged role through the swap path. The residual risk concentrates in two areas. The first is a heavy reliance on semi-trusted operational roles, namely the curator, the market maker, fillers, and converters, whose responsibilities are extensive and, on the conversion path, gated by off-chain monitoring rather than by on-chain limits. The second is the consistency and availability of net asset value across the redemption lifecycle, where inventory is valued through an external oracle and through Midas request state on differing bases, and where the accounting and deallocation paths can revert on an external fault.

The most consequential issues concern availability. A single token-to-redeem whose oracle returns zero or reverts can brick the vault's accounting, with recovery paths that themselves depend on the same reverting valuation, and a frozen or sub-minimum redemption account can revert deallocation or synchronization across the vault. Unlike the other adapters, this adapter does not isolate these external faults. The remaining findings cluster around redemption value being grabbable or redirected by callers other than the party that funded it, the mispricing and non-redeemability of certain Midas mToken configurations, predictable valuation transitions that allow timed extraction from passive holders, and several lower-severity lifecycle and configuration inconsistencies. Isolating the accounting and deallocation paths from external faults, binding swap funding and recipients, correcting the Midas mToken pricing and redemption configuration, and exercising the full Midas request lifecycle in the test suite would materially reduce the residual risk.

The Symbiotic team is thanked for their responsiveness and willingness to provide technical context throughout the engagement.

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.