Gravity: GRVT L1 Treasury Vault Contracts

Security Review

Cantina Managed review by:
R0bert, Lead Security Researcher
Slowfi, Lead Security Researcher

April 2, 2026

Contents

1 Introduction

1.1 About Cantina

Cantina is a security services marketplace that connects top security researchers and solutions with clients. Learn more at cantina.xyz

1.2 Disclaimer

Cantina Managed provides a detailed evaluation of the security posture of the code at a particular moment based on the information available at the time of the review. While Cantina Managed endeavors to identify and disclose all potential security issues, it cannot guarantee that every vulnerability will be detected or that the code will be entirely secure against all possible attacks. The assessment is conducted based on the specific commit and version of the code provided. Any subsequent modifications to the code may introduce new vulnerabilities that were absent during the initial review. Therefore, any changes made to the code require a new security review to ensure that the code remains secure. Please be advised that the Cantina Managed security review is not a replacement for continuous security measures such as penetration testing, vulnerability scanning, and regular code reviews.

1.3 Risk assessment

Severity level	Impact: High	Impact: Medium	Impact: Low
Likelihood: high	Critical	High	Medium
Likelihood: medium	High	Medium	Low
Likelihood: low	Medium	Low	Low

1.3.1 Severity Classification

The severity of security issues found during the security review is categorized based on the above table. Critical findings have a high likelihood of being exploited and must be addressed immediately. High findings are almost certain to occur, easy to perform, or not easy but highly incentivized thus must be fixed as soon as possible.

Medium findings are conditionally possible or incentivized but are still relatively likely to occur and should be addressed. Low findings are a rare combination of circumstances to exploit, or offer little to no incentive to exploit but are recommended to be addressed.

Lastly, some findings might represent objective improvements that should be addressed but do not impact the project's overall security (Gas and Informational findings).

2 Security Review Summary

From Mar 17th to Mar 20th the Cantina team conducted a review of defivault-contract, dev-exchange-contract and dev-exchange-contract on commit hashes 15dd3d4f, e45ebb8e and bfb1d847 respectively. The team identified a total of **19** issues:

Issues Found

Severity	Count	Fixed	Acknowledged
Critical Risk	0	0	0
High Risk	3	3	0
Medium Risk	3	3	0
Low Risk	10	10	0
Gas Optimizations	0	0	0
Informational	3	2	1
Total	**19**	**18**	**1**

2.1 Scope

The security review had the following components in scope for defivault-contract, dev-exchange-contract and dev-exchange-contract on commit hashes 15dd3d4f, e45ebb8e and bfb1d847 respectively:

```
├── defivault-contract
│   └── contracts
│       ├── external
│       │   ├── IAaveV3AToken.sol
│       │   ├── IAaveV3Pool.sol
│       │   ├── IGRVTBridgeProxyFeeToken.sol
│       │   ├── IL1ZkSyncBridgeHub.sol
│       │   ├── IProxyAdmin.sol
│       │   └── IWrappedNative.sol
│       ├── gateways
│       │   ├── NativeBridgeGateway.sol
│       │   └── NativeVaultGateway.sol
│       ├── strategies
│       │   └── AaveV3Strategy.sol
│       └── vault
│           ├── GRVTL1TreasuryVault.sol
│           ├── VaultBridgeLib.sol
│           └── VaultStrategyOpsLib.sol
├── dev-exchange-contract-pr-59
│   └── contracts
│       └── exchange
│           ├── api
│           │   ├── BaseContract.sol
│           │   ├── OracleContract.sol
│           │   └── TransferContract.sol
│           ├── facets
│           │   └── GetterFacet.sol
│           └── types
│               └── DataStructure.sol
└── dev-exchange-contract-pr-72
    └── contracts
        └── exchange
            └── api
                └── TransferContract.sol
```

3 Findings

3.1 High Risk

3.1.1 Native recovery rejects token vault sender

Severity: High Risk

Context: NativeBridgeGateway.sol#L190-L194

Description: `NativeBridgeGateway.receive()` only accepts ETH from `wrappedNativeToken` and `bridgeHub.sharedBridge()`. That is too strict for the live zkSync and GRVT recovery flow. When a native bridge deposit fails and is reclaimed, the final ETH transfer back to the original `depositSender` comes from the L1NativeTokenVault, not from `sharedBridge()`.

A realistic failure looks like this. The vault rebalances native liquidity to L2 through `NativeBridgeGateway`. The L2 leg later fails. An operator, or any third party helping with recovery, submits a valid failed-deposit claim through the bridge stack. The native token vault then tries to send the refunded ETH back to `NativeBridgeGateway`, because the gateway was the original L1 `depositSender`. That transfer hits `UnexpectedNativeSender` and reverts. The claim cannot complete, the ETH is not wrapped back into the vault and the failed native deposit stays stuck until the gateway is upgraded or replaced.

This turns a normal bridge failure into a permanent recovery failure under the current wiring.

Proof of Concept: See gist cc1c45eb.

Recommendation: Consider allowing the actual recovery sender used by the bridge stack, not just `bridgeHub.sharedBridge()`. At minimum, the gateway should accept ETH from the configured native token vault as well as from `wrappedNativeToken`.

The safer fix is to bind the allowed recovery sender to the bridge configuration used for each native bridge record instead of relying on a fresh `sharedBridge()` lookup at recovery time. Add an end-to-end fork test that executes failed native recovery and proves that ETH sent from the native token vault is accepted and returned to the vault as wrapped native.

Gravity: Fixed in PR 58.

Cantina Managed: PR 58 fixes this issue for the fresh-deployment path used by `ignition/modules/NativeGateways.ts`, because that module deploys a new `NativeBridgeGateway` proxy and calls `initialize()` immediately and the new logic sets `nativeTokenVault` during initialization. However, if the intended resolution path is to upgrade an already-deployed `NativeBridgeGateway` proxy instead of deploying a fresh one, the fix is incomplete. The patch introduces a new `nativeTokenVault` storage variable and changes `receive()` to trust that address, but `nativeTokenVault` is only set in `initialize()`. Existing proxies cannot call `initialize()` again and the repository's `NativeBridgeGatewayUpgrade` flow still defaults to `upgradeCallData: "0x"`, so upgrading an already-initialized proxy would leave `nativeTokenVault == address(0)`. In that state, refunds sent by the zkSync native token vault would still revert with `UnexpectedNativeSender`. There is no committed non-zero live deployment address in the repo, so based on the checked-in deployment module this resolution appears correct if the plan is to deploy a fresh `NativeBridgeGateway`. It only remains incomplete for an in-place upgrade flow, which would need an explicit migration path such as a `reinitializer` called via `upgradeAndCall`.

3.1.2 Native bridge uses zero-address encoding

Severity: High Risk

Context: NativeBridgeGateway.sol#L142-L154

Description: `NativeBridgeGateway.bridgeNativeToL2` still builds the native bridge payload as `abi.encode(address(0), amount, l2Recipient)`. That no longer matches the live Matter Labs asset-router stack. In the current bridge code, native ETH is identified as `address(1)`, and the router treats a payload whose first byte is `0x00` as a different encoding mode entirely.

On a live GRVT mainnet fork, this means the native bridge submission fails before the deposit is even processed. The gateway unwraps WETH, approves the fee token, and calls `requestL2TransactionTwoBridges`, but the downstream asset router interprets the calldata as the special set-asset-handler format and immediately reverts with `NonEmptyMsgValue()` because a real native

bridge carries non-zero `msg.value`. The result is that the native bridge path is not merely misconfigured at recovery time. It is incompatible with the live router at submission time.

A realistic failure case is a routine or emergency L1 to L2 native rebalance. The vault has idle WETH, the rebalancer calls `rebalanceNativeToL2` or `emergencyNativeToL2`, and the entire operation reverts even though enough liquidity is available. No native liquidity reaches L2, and the protocol loses its intended ability to top up the exchange through the native path.

The impact is a hard liveness failure for native bridge-outs on the current bridge stack. Both normal and emergency native rebalances can be unavailable until the gateway is upgraded.

Recommendation: Update the native bridge payload to use the live native token identifier expected by the asset router instead of `address(0)`. More generally, avoid hardcoding bridge encoding details inside `NativeBridgeGateway` when they come from an external protocol that can evolve. The safer fix is to centralize the native token sentinel and calldata format in one shared integration constant or adapter, then cover the native path with a fork test that proves a real live submission gets past the asset-router boundary.

Gravity: Fixed in PR 57.

Cantina Managed: PR 57 updates `NativeBridgeGateway.bridgeNativeToL2` so the native bridge leg is no longer encoded as `abi.encode(address(0), amount, l2Recipient)`. Instead, it uses a shared helper, `ZkSyncAssetRouterEncoding.encodeLegacyNativeDeposit(amount, l2Recipient)`, which encodes the live zkSync native ETH sentinel `address(1)`. The native-token sentinel is also centralized in `ZkSyncAssetRouterEncoding`, so the gateway, mocks and tests all use the same asset-router encoding and do not drift from the live Matter Labs bridge semantics.

3.1.3 Native recovery can use ETH from an unrelated failed deposit

Severity: High Risk

Context: NativeBridgeGateway.sol#L172-L185

Description: `recoverClaimedNativeDeposit` trusts the caller to name the correct `bridgeTxHash` for the ETH currently sitting in `NativeBridgeGateway`. The function only checks that a record exists, that it has not already been recovered, and that the gateway's ETH balance is at least `record.amount`. It does not prove that the balance came from that record, or even that the named bridge ever failed.

That matters because the recovery workflow is intentionally split. The failed deposit is claimed on the shared bridge first, and the ETH is then sent to `NativeBridgeGateway` as raw balance. Only after that does someone call `recoverClaimedNativeDeposit`. Since the gateway records every native bridge submission in `nativeBridgeRecords`, any caller can wait for one failed deposit to be claimed, pick a different unrecovered record with a smaller `amount` and mark that smaller record as recovered first. The gateway will wrap that smaller amount and send it to the vault, leaving the rest of the ETH behind without any record-level association.

For example, imagine that the team has an old native bridge record for `10 ETH` that was never meant to be recovered because the original bridge succeeded, so its entry still exists with `recovered == false`. Later, a genuine bridge failure happens for `250 ETH` and operations submit the shared-bridge claim that sends `250 ETH` back to `NativeBridgeGateway`. Before they complete the matching recovery step, a watcher sees that the gateway now holds ETH and calls `recoverClaimedNativeDeposit` with the stale `10 ETH` hash. The call succeeds, the vault receives only `10 WETH` and the gateway is left holding `240 ETH` that no longer matches any recoverable record. When operations try to recover the real `250 ETH` failure, the call reverts because the gateway balance is now below `record.amount`.

Once this happens, the original failed deposit can become unrecoverable. If record `A` is for `100 ETH` and record `B` is for `5 ETH`, then claiming `A` and recovering `B` first leaves the gateway with `95 ETH`. Record `A` still expects an exact `100 ETH` recovery, so `recoverClaimedNativeDeposit(A)` now reverts on the balance check. The leftover ETH is stranded in the gateway and the contract has no generic native or token rescue path to repair the mismatch. This is not just an operator mistake. The function is permissionless, so any account can trigger the mismatch once a failed deposit has been claimed to the gateway.

Recommendation: Bind the failed-deposit claim to the recovery record in the same transaction. The safest fix is to move the shared-bridge `claimFailedDeposit` call behind a gateway-controlled function that accepts the exact claim parameters, performs the claim, and immediately marks and recovers the matching `bridgeTxHash`. That removes the window where the gateway holds unassociated ETH.

If the split workflow must remain, `recoverClaimedNativeDeposit` should at least be restricted to a trusted operator and backed by explicit claim-side accounting that ties the received ETH to one specific bridge record before `recovered` is set. A rescue path for unexpected ETH, wrapped native and fee-token balances should also be added so the system can be repaired after a mismatch instead of leaving funds stranded in the gateway.

Gravity: Fixed in PR 68.

Cantina Managed: PR 68 mitigates the issue in two ways. The merged `NativeBridgeGateway` adds `claimAndRecoverFailedNativeDeposit`, which atomically claims and recovers the exact stored record, and it makes `recoverClaimedNativeDeposit` admin-only with an exact-balance check instead of the previous `balance >= amount` behavior.

3.2 Medium Risk

3.2.1 Unbounded queue drain can become non executable

Severity: Medium Risk

Context: TransferContract.sol#L121-L138

Description: `processWithdrawalQueue()` processes queued withdrawals in an unbounded loop. In `contracts/exchange/api/TransferContract.sol`, the function keeps draining items until the queue is empty or the contract runs out of available liquidity for the asset being withdrawn. There is no `maxCount`, no caller-supplied bound, and no built-in stop condition based on remaining gas.

That design is dangerous because the backlog itself is also unbounded. Once any withdrawal is pending, later withdrawals are appended to the same queue rather than being executed immediately. Over time, the queue can therefore accumulate an arbitrary number of entries. If liquidity is later restored and is sufficient to cover many queued withdrawals, the recovery call will attempt to process all of them in a single transaction.

At that point, recovery can fail entirely due to gas limits. If the loop grows beyond what fits in one transaction, the call reverts and none of the queued withdrawals are processed. Because the transaction reverts as a whole, the queue head remains unchanged and the backlog cannot make forward progress through that recovery path. The result is a liveness failure in the queue-drain mechanism itself.

In the current deployment model, where the exchange operates on a private L2, this is better understood as an operational and protocol-liveness risk than as a public permissionless attack. But the issue remains real. A sufficiently large backlog can make the documented recovery action non-executable under the chain's gas limits, leaving withdrawals stuck even after liquidity has been replenished.

Impact: if the withdrawal backlog grows large enough, `processWithdrawalQueue()` can exceed transaction gas limits and revert, making the queue non-drainable through its intended recovery path and prolonging withdrawal disruption.

Recommendation: Queue processing should be explicitly bounded so recovery can make partial progress in predictable chunks. The simplest fix is to let the caller process at most `N` items per transaction, advancing the queue head incrementally even when the full backlog is large.

One straightforward pattern is:

```
function processWithdrawalQueue(uint256 maxCount) external
↪   onlyTxOriginRole(LIQUIDITY_ORCHESTRATOR_ROLE) {
    uint256 processed;

    while (processed < maxCount && state.pendingWithdrawalQueue.head <
    ↪   state.pendingWithdrawalQueue.tail) {
        // process one queued withdrawal
        processed++;
    }
}
```

This ensures that recovery remains executable regardless of backlog size, as operators can drain the queue across multiple transactions. As defense in depth, the implementation can also stop early when remaining gas drops below a safety threshold, and the test suite should include a regression case showing that a large backlog can still be drained over multiple bounded calls without reverting the entire operation.

Gravity: Fixed in PR 80.

Cantina Managed: PR 80 changes `processWithdrawalQueue()` into `processWithdrawalQueue(uint256 maxCount)`. The drain loop is bounded by a processed-item counter, so operators can drain the queue in multiple predictable transactions instead of attempting to process the entire backlog in one call. A large withdrawal backlog no longer has to be recovered through a single all-or-nothing transaction that can exceed the gas limit and revert without making progress.

3.2.2 Emergency unwind can miss fully recoverable Aave liquidity

Severity: Medium Risk

Context: VaultBridgeLib.sol#L257-L283

Description: `unwindStrategiesForEmergency` assumes that `strategy.deallocate(token, min(remainingNeeded, exposure))` is always the right way to drain emergency liquidity from a strategy. That is not true for the shipped `AaveV3Strategy`. Its `strategyExposure` includes both the `aToken`-backed position and any residual underlying held directly by the strategy, but `deallocate` forwards the requested amount into `aavePool.withdraw` before sweeping residual underlying. The result is that the unwind loop can ask Aave for more than the strategy's actual aToken-backed balance.

This creates a failure mode on the exact path operators rely on during stress. A strategy may have `100` of aToken position and `1` unit of residual underlying dust, so `strategyExposure` reports `101`. If the emergency bridge needs all `101`, the loop requests `101` through `deallocate`. The Aave adapter then calls `withdraw(101)`, which reverts because only `100` is withdrawable from Aave. The library catches that revert, marks the strategy as skipped, and continues. The residual underlying is never swept because the revert happens before the sweep step. A later `deallocateAll()` would recover the entire `101`, but the emergency path never tries it.

The funds are recoverable, but the vault can still conclude that the requested emergency bridge amount is unavailable and revert the top-up. That is a meaningful operational issue, especially because emergency bridge flows are designed to work while paused and after token support has been removed.

Recommendation: When the emergency unwind is trying to drain a strategy's full exposure, do not rely only on bounded `deallocate(request)`. The smallest safe change is to call `deallocateAll()` whenever `request >= exposure`, or to retry with `deallocateAll()` when `deallocate(request)` fails but the strategy still reports non-zero exposure.

That fallback matches the adapter's own semantics. `AaveV3Strategy.deallocateAll` already uses `type(uint256).max`, which is the correct full-exit path for Aave and still sweeps residual underlying back to the vault after the withdraw. If more strategies are added later, the interface should make full-exit behavior explicit so emergency unwinds do not depend on adapter-specific assumptions.

Gravity: Fixed in PR 67. The emergency path now uses `deallocateAll` when draining a strategy completely, while partial withdrawals continue to use the bounded call. A test covering the Aave dust case was added to prove the full amount is recovered.

Cantina Managed: The key fix is in `VaultBridgeLib`. The emergency unwind loop now sets `useAll = request == exposure`, and `tryEmergencyDeallocate` calls `deallocateAll(token)` instead of bounded `deallocate(token, request)` in that case.

3.2.3 Aave partial deallocate breaks on residual dust

Severity: Medium Risk

Context: AaveV3Strategy.sol#L86

Description: `AaveV3Strategy` is using two different meanings for the same amount. `strategyExposure` reports total assets as `aToken` balance plus any loose underlying sitting in the strategy. Partial `deallocate(amount)` does something different: it asks Aave for `amount`, then sends any loose underlying on top.

```
function strategyExposure(address token) external view override returns (uint256
↪  exposure) {
    if (token != underlying) return 0;
    return IERC20(aToken).balanceOf(address(this)) +
    ↪  IERC20(underlying).balanceOf(address(this));
```

```
}

function _deallocateAndSweep(uint256 requested) internal returns (uint256 received) {
    if (IERC20(aToken).balanceOf(address(this)) != 0) {
        received = aavePool.withdraw(underlying, requested, vault);
    }
    uint256 swept = _sweepUninvestedTokenToVault();
    if (swept != 0) received += swept;
}
```

Those two views only match while the direct underlying balance is zero. That is not a safe assumption here. Anyone can transfer the underlying token straight into the strategy and `allocate` does not reinvest an existing residual balance before supplying fresh funds to Aave. Once that happens, the strategy starts reporting exposure it cannot satisfy through the bounded Aave withdraw path alone.

The bad behavior is easy to see with a simple example. If the strategy has `100` in Aave and someone sends it `3` underlying, `strategyExposure` becomes `103`. A partial `deallocate(40)` does not return `40`. It withdraws `40` from Aave, then sweeps the extra `3`, so the vault gets `43`.

That creates two real problems for the vault. The first one is harvest griefing. `harvestYieldFromStrategy` checks the strategy's reported yield before the call, then reverts if more than that amount comes back. A tiny donation is enough to make a normal harvest revert because partial `deallocate` returns the requested Aave withdrawal plus the donated dust.

```
uint256 maxYield = harvestableYield(token, strategy);
if (maxYield == 0 || amount > maxYield) revert YieldNotAvailable();
//...
if (withdrawnToVault > maxYield) revert YieldNotAvailable();
```

The second problem is emergency unwind liveness. The emergency path reads `strategyExposure`, chooses a partial request when `request < exposure` and then calls bounded `deallocate(request)`. If residual underlying has pushed `exposure` above the actual `aToken` balance, a request can be smaller than `exposure` and still larger than what Aave can withdraw. In that case the emergency step reverts and gets skipped even though the strategy has enough total assets to cover the request.

```
uint256 request = remainingNeeded < exposure ? remainingNeeded : exposure;
bool useAll = request == exposure;
```

Recommendation: Pick one meaning for partial `deallocate(amount)` and enforce it everywhere. The cleanest fix is to make `amount` mean "return at most this much in total." In practice that means using any residual underlying first, then withdrawing only the shortfall from Aave, and stopping once the request has been met. That keeps partial deallocation aligned with `strategyExposure` and avoids both over-return and false Aave reverts.

If that is more invasive than desired, the smaller safe change is to stop sweeping residual underlying during bounded `deallocate` and keep the sweep in `deallocateAll` only. That still removes the mismatch for harvest and emergency partial unwinds. Separately, `allocate` should reinvest any pre-existing residual underlying so donated dust does not linger between calls and keep reopening the same edge case.

Gravity: Fixed in PR 72. Bounded `deallocate` now uses loose underlying first and withdraws only the shortfall from Aave, returning at most the requested total. `deallocateAll` remains the full-exit path and still sweeps all residual dust.

Cantina Managed: Fix verified.

3.3 Low Risk

3.3.1 Unsupported ERC20s can use bridge path

Severity: Low Risk

Context: VaultBridgeLib.sol#L135-L162

Description: The vault treats token support and bridge compatibility as if they were the same thing, but they are not. Once governance marks an ERC20 as supported, `rebalanceErc20ToL2` and `emergencyErc20ToL2` allow that token to use the generic shared-bridge path as long as it is not the

wrapped native token. `VaultBridgeLib` then approves the requested amount and submits the bridge call without any token-specific compatibility check.

That is only safe for plain ERC20 behavior. Tokens with transfer fees, rebasing logic, unusual transfer hooks, or other non-standard semantics can be safe enough to custody in the vault while still being unsafe for the generic bridge path. The current design has no separate allowlist or adapter layer for that distinction. Once a token is marked as vault-supported, operators can route it into a bridge flow that implicitly assumes standard transfer semantics.

Because of this, a token can be safe to hold in the vault but still unsafe to send through the generic bridge flow. In that case, a rebalance can revert, deliver less than expected, or leave funds stuck until someone intervenes manually. The system does not enforce a simple rule that every vault-supported ERC20 is also safe to bridge.

Recommendation: Separate bridge eligibility from general vault-token support. Maintain an explicit allowlist of tokens that are known to be compatible with the shared bridge and reject all other ERC20s from the generic rebalance paths. If non-standard tokens must be bridged, route them through dedicated adapters with token-specific accounting and validation instead of the generic shared ERC20 flow. The runbook and the documentation should also state clearly that vault support alone does not imply bridge compatibility.

Gravity: Fixed in PR 62.

Cantina Managed: PR 62 adds a separate `bridgeable` allowlist and enforces it on both ERC20 bridge paths, resolving this issue for fresh deployments. The only caveat is upgrade safety: if an already-deployed pre-PR 62 vault were upgraded in place, the added storage variable would require separate layout handling.

3.3.2 Native harvest can under-measure forwarded ETH

Severity: Low Risk

Context: VaultStrategyOpsLib.sol#L81-L88

Description: `VaultStrategyOpsLib.payoutHarvestProceeds` decides how much native yield was received by measuring the recipient's ending ETH balance delta. That is only accurate if the recipient keeps the ETH. It is not accurate for treasury contracts that accept ETH and then immediately forward it, split it, or sweep it elsewhere during the same call.

In that case the transfer itself succeeds, but the recipient may finish with little or no retained ETH. The helper then reports a much smaller `received` amount than what actually left the vault and reached the recipient's downstream logic. The payout succeeded, but the accounting says otherwise.

For example, imagine that the protocol upgrades `yieldRecipient` from an EOA to a treasury router that immediately forwards incoming ETH to a multisig, splitter, or accounting module. Suppose the vault harvests `10 ETH` of wrapped-native yield with `minReceived = 9 ETH`. The vault unwraps and sends the full `10 ETH` and the router immediately forwards all `10 ETH` to its downstream treasury destination in the same transaction. From the protocol's perspective the payout worked, but the router's ending ETH balance is still `0`, so `payoutHarvestProceeds` reports `received = 0`. If `minReceived` is enforced, the harvest reverts even though the treasury system actually received the funds. If `minReceived = 0`, the harvest succeeds but telemetry still claims that nothing was received.

Recommendation: Do not use the recipient's terminal ETH balance as the authoritative receipt measure for native harvest payouts. If the native transfer succeeds, either treat the transferred `amount` as received or route native harvest payout through an explicit treasury interface that can acknowledge receipt directly.

Gravity: Fixed in PR 61.

Cantina Managed: PR 61 changes the native branch of `VaultStrategyOpsLib.payoutHarvestProceeds` to treat a successful native payout as the full transferred `amount` instead of measuring the recipient's ending ETH balance delta. If the configured `yieldRecipient` is a treasury router or forwarding contract that immediately redistributes ETH during the same call, the vault no longer under-measures `received` as `0` or some smaller retained amount. As long as `_sendNative` succeeds, the harvest now accounts for the full native payout that left the vault.

3.3.3 Aave dust-only residual cannot be swept

Severity: Low Risk

Context: AaveV3Strategy.sol#L349-L354

Description: `AaveV3Strategy` always calls `aavePool.withdraw(...)` before it tries to sweep leftover `underlying` held directly by the strategy. That ordering is fine while the strategy still owns some `aToken` balance, but it breaks once the Aave position has already been fully exited and the strategy only holds residual `underlying`.

In that state, the adapter asks Aave to withdraw from an empty position first. On live Aave V3, an empty `withdraw(type(uint256).max)` path resolves to a zero withdrawal and reverts before the strategy reaches its local sweep step. The same problem affects the bounded `deallocate` path for the same reason. As a result, the contract can end up holding a small direct `underlying` balance that the normal exit path can no longer clear.

That state is reachable without any privileged action. After the strategy has been fully unwound, anyone can transfer a small amount of the configured `underlying` token directly to the strategy. The strategy then reports non-zero exposure, but both deallocation entry points revert before the direct-transfer dust is returned to the vault.

This is a liveness and cleanup issue rather than a theft vector. A trivial donation can keep the pair looking non-empty, which can pin it in `WithdrawOnly` and interfere with a clean removal until governance reconfigures or upgrades the adapter.

Recommendation: Handle the zero-`aToken` case before calling Aave. If the strategy has no `aToken` balance, skip the pool withdrawal and sweep any locally held `underlying` directly to the vault. More generally, the local dust sweep should not depend on a successful external withdrawal when the only remaining assets are already in the strategy itself.

Gravity: Fixed in PR 60.

Cantina Managed: PR 60 updates `AaveV3Strategy._deallocateAndSweep()` so the strategy only calls `aavePool.withdraw(...)` when it still holds a non-zero `aToken` balance. If the Aave position has already been fully exited and only direct `underlying` dust remains on the strategy, the adapter now skips the external Aave withdraw and sweeps the locally held `underlying` directly back to the vault. Both `deallocateAll()` and bounded `deallocate()` can now clear dust-only residual balances, return strategy exposure to zero and avoid leaving the strategy pair stuck in `WithdrawOnly` solely because of donated residual `underlying`.

3.3.4 Queue recovery runbook uses the wrong signer

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

Description: The queue recovery runbook documents a signer that is not authorized to call the recovery function it instructs operators to use. The contract `TransferContract.sol`, `processWithdrawalQueue` function is gated by `onlyTxOriginRole(LIQUIDITY_ORCHESTRATOR_ROLE)`, so successful execution requires the transaction origin to hold the liquidity orchestrator role.

The operational documentation does not match that requirement. In `docs/transfer-facet-cast-calls.md`, the command shown for draining the withdrawal queue is signed with `EXCHANGE_ADMIN_PRIVATE_KEY` even though the surrounding text describes the function as liquidity-orchestrator-only. Unless the exchange admin key has separately been granted `LIQUIDITY_ORCHESTRATOR_ROLE`, the documented command will revert at runtime.

Under the current deployment model this is an internal operational issue rather than a public exploit. The exchange runs on a private L2 and the runbook is intended for operators, not arbitrary external users. That does not make it harmless. The queue-drain path exists specifically for recovery once liquidity has been restored and pending withdrawals need to be processed. If the documented command fails during that flow, operators can lose time diagnosing an avoidable permissions error while the queue remains stuck.

Impact: the published recovery procedure can fail in a time-sensitive queue-drain scenario because it uses a signer that is not authorized by the contract. This creates avoidable operational delay and increases the risk of extended withdrawal disruption during incident response.

Recommendation: The runbook should be updated to use a key that is actually authorized to call `processWithdrawalQueue()`, or it should explicitly state that the configured admin key must also hold

`LIQUIDITY_ORCHESTRATOR_ROLE`.

A straightforward fix is to replace `EXCHANGE_ADMIN_PRIVATE_KEY` in the documented command with the orchestrator signer used for liquidity operations, for example:

```
cast send "$EXCHANGE_PROXY_ADDR" \
  "processWithdrawalQueue()" \
  --private-key "$LIQUIDITY_ORCHESTRATOR_PRIVATE_KEY" \
  --rpc-url "$RPC_URL"
```

As defense in depth, the operational checklist should also include a preflight role-verification step so the team confirms the chosen signer has `LIQUIDITY_ORCHESTRATOR_ROLE` before relying on the recovery runbook during an incident.

Gravity: Fixed in PR 78.

Cantina Managed: PR 78 updates `docs/transfer-facet-cast-calls.md` so the `processWithdrawalQueue()` example uses `LIQUIDITY_ORCHESTRATOR_PRIVATE_KEY` instead of `EXCHANGE_ADMIN_PRIVATE_KEY`. That change correctly aligns the runbook with the contract authorization model: `processWithdrawalQueue()` is gated by `onlyTxOriginRole(LIQUIDITY_ORCHESTRATOR_ROLE)` in `TransferContract.sol`, so the transaction origin must hold `LIQUIDITY_ORCHESTRATOR_ROLE`.

3.3.5 Storage gap accounting is off by one slot

Severity: Low Risk

Context: DataStructure.sol#L163-L170

Description: The upgradeable storage gap in `contracts/exchange/types/DataStructure.sol` is reduced by the wrong amount for the storage added in this change. The PR introduces a new `WithdrawalQueue` field together with two new address fields, but the reserved `__gap` array is only shrunk by three slots.

That accounting is incorrect because `WithdrawalQueue` is not a one-slot addition in this layout. Its `requests` mapping consumes one anchor slot, and its `head` and `tail` indices consume another packed slot. Combined with the two newly added address fields, this change consumes four slots in total. Reducing `__gap` by only three leaves the contract's reserved-gap bookkeeping inconsistent with the actual storage layout.

I do not see evidence of immediate storage corruption in the current inheritance tree, so this is not an active exploit or an instant upgrade brick on this specific deployment. The danger is forward-looking: upgrade-safe contracts rely on accurate gap accounting so future versions can add storage without colliding with inherited layout. Once the accounting is wrong, later upgrades can be designed under a false assumption about how many reserved slots remain.

This issue is independent of chain access or operator permissions. Whether the exchange runs on a public chain or a private L2 does not change the underlying upgrade-safety risk.

Impact: the contract's reserved-gap bookkeeping no longer matches the real storage layout, increasing the risk of storage-collision mistakes or unsafe assumptions in future upgrades.

Recommendation: The `__gap` size should be reduced to reflect the actual number of newly consumed slots. Since this change adds four slots of storage, the reserved gap should be decreased by four rather than three.

A straightforward correction is:

```
uint256[43] __gap;
```

After adjusting the gap, the team should regenerate and review the storage layout to confirm that the reserved region starts at the expected slot and that future upgrades can rely on accurate remaining-gap accounting. As a general practice, any PR that adds state to upgradeable contracts should include a storage-layout check so slot consumption and gap updates are validated together.

Gravity: Fixed in PR 77.

Cantina Managed: PR 77 reduces the storage gap from `uint256[44] __gap` to `uint256[43] __gap`, which is the correct post-PR 59 value. The slot math is: pre-PR 59 gap of `47`, new `WithdrawalQueue` consuming `2` slots total (`requests` mapping anchor plus packed `head` and `tail`), new `l1DefiVaultAddress` consuming `1` slot, and new `nativeVaultGatewayAddress` consuming `1` slot. The change therefore consumes `4` slots in total, so the correct remaining gap is `47 - 4 = 43`. Because the resolution updates the gap to `43`, it brings the reserved-gap bookkeeping back in line with the actual storage layout.

3.3.6 Default ERC20 config still counts as ETH

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

Description: PR72 attempts to fix the bridge recipient selection logic so that `nativeVaultGatewayAddress` is used only when ETH is actually configured, while all other assets continue routing to `l1DefiVaultAddress`. The intended safety property is clear: the protocol should only use the native ETH withdrawal destination when the contract can positively identify the bridged asset as ETH. The implementation does not fully achieve that property because it relies on a config accessor that silently falls back to the default slot of the two-dimensional config map.

The relevant helper fetches the ETH token address by calling `_getAddressConfig2D` for `ConfigID.ERC20_ADDRESSES` and the ETH subkey. That helper does not perform an exact lookup. If the requested subkey is unset, it transparently falls back to `DEFAULT_CONFIG_ENTRY` and returns that value together with its `isSet` flag. In other words, the code path that is trying to answer the question "is ETH explicitly configured?" is actually answering the weaker question "does either the ETH slot or the default slot contain a value?".

That distinction matters because the configuration layer does not forbid writing subkey zero for two-dimensional configs. As a result, `ERC20_ADDRESSES[DEFAULT_CONFIG_ENTRY]` can legally exist. If operations populate that default entry for convenience, migration support, or any other reason while leaving the ETH-specific slot unset, `_getL1BridgeRecipient` will still observe `isEthConfigured == true`. At that point, any token whose L2 address matches the default entry will be treated as if it were ETH and will be routed to `nativeVaultGatewayAddress` instead of `l1DefiVaultAddress`.

This means the PR72 fix is incomplete. The routing decision still depends on ambient fallback state rather than on an explicit ETH configuration. The protocol can therefore misclassify a non-ETH asset as ETH without any malicious token behavior and without violating the current config validation rules. If the two recipient addresses have different operational expectations, accounting assumptions, or custody flows, the misrouted asset can end up in the wrong destination and require manual recovery. In the worst case, funds could be bridged into a path that is only safe for native ETH handling, creating loss or prolonged unavailability for that asset.

Impact: an operator configuration that appears to leave ETH disabled can still activate the ETH routing branch through the default entry. This undermines the main correctness guarantee added by PR72 and can send non-ETH assets to the wrong L1 recipient.

Recommendation: Do not use a fallback-based config accessor when the logic requires an exact ETH-specific presence check. The recipient helper should read the ETH slot directly and treat ETH as configured only if that exact entry is set. The default entry should not participate in this decision.

One straightforward fix is to bypass `_getAddressConfig2D` and inspect the ETH subkey explicitly:

```
ConfigValue storage ethConfig =
    state.config2DValues[ConfigID.ERC20_ADDRESSES][_currencyToConfig(Currency.ETH)];

bool isEthConfigured = ethConfig.isSet;
address ethL2Token = _configToAddress(ethConfig.val);
```

Then branch to `nativeVaultGatewayAddress` only when `isEthConfigured` is true and `l2Token == ethL2Token`. For defense in depth, consider forbidding `DEFAULT_CONFIG_ENTRY` for `ConfigID.ERC20_ADDRESSES` entirely unless a real fallback is intentionally supported across all call sites. The test suite should also add a regression case where the default ERC20 entry is populated, the ETH slot is unset, and the recipient helper must still return `l1DefiVaultAddress` for non-ETH assets.

Gravity: Fixed in PR 77.

Cantina Managed: PR 77 removes the fallback-based ETH presence check from `_getL1BridgeRecipient()`. Instead of calling `_getAddressConfig2D(ConfigID.ERC20_ADDRESSES, _currencyToConfig(Currency.ETH))`, which can fall back to `DEFAULT_CONFIG_ENTRY`, the patched code reads the ETH config slot directly from `state.config2DValues[ConfigID.ERC20_ADDRESSES][_currencyToConfig(Currency.ETH)]`. That means `isEthConfigured` is now derived only from the exact ETH entry's `isSet` flag. A populated default ERC20 entry can no longer make an unset ETH slot appear configured, so a non-ETH token that matches the default entry will no longer be misclassified as ETH and routed to `nativeVaultGatewayAddress`.

3.3.7 Vault bridge spends queued withdrawal liquidity

Severity: Low Risk

Context: TransferContract.sol#L167-L193

Description: PR59 changes withdrawal settlement so queued requests reduce user liabilities immediately, but the corresponding ERC20 tokens can remain on the exchange until `processWithdrawalQueue()` later bridges them. In the same contract, `bridgeToL1DefiVault()` is allowed to bridge arbitrary exchange-held token balances without checking how much of that balance is already owed to queued withdrawals.

This means queued-withdrawal liquidity is economically committed but not segregated. The queue path removes `amountToSend` from `state.totalSpotBalances` before any bridge side effect occurs, while the vault-bridge path still treats the raw on-contract ERC20 balance as fully spendable. If the liquidity orchestrator uses `bridgeToL1DefiVault()` on the same token, it can consume balances that queued user withdrawals still need in order to settle.

Under the current trust model this is not a public theft bug because the conflicting action requires `LIQUIDITY_ORCHESTRATOR_ROLE`. The problem is that the contract does not enforce any separation between user-withdrawal liquidity and vault-bridge liquidity. A mistaken or overly aggressive bridge operation can therefore prolong queued withdrawals and make the overdue-queue halt more likely.

Recommendation: The contract should reserve queued withdrawal amounts per token and exclude those reserves from vault bridging. The smallest safe fix is to track outstanding queued obligations by token or currency and require `bridgeToL1DefiVault()` to leave enough balance behind to satisfy them.

A straightforward pattern is to compute a spendable balance as:

```
spendable = IERC20(l2Token).balanceOf(address(this)) -
↪  reservedForQueuedWithdrawals[l2Token];
require(amount <= spendable, "insufficient unreserved liquidity");
```

If the team does not want to maintain a separate reserve accounting path, the vault bridge should at minimum be blocked whenever there are pending queued withdrawals for the same token.

Gravity: Fixed in PR 105.

Cantina Managed: PR 105 blocks vault bridging for all tokens while any queued withdrawal exists, not just for the queued token.

3.3.8 Queue getter has unbounded page size

Severity: Low Risk

Context: GetterFacet.sol#L190-L212

Description: `getPendingWithdrawalRequests()` accepts an arbitrary `limit` and uses it directly to size the returned memory array and to control the copy loop. There is no protocol-side cap on how many queue entries a caller may request in one call.

That makes the getter easy to misuse from onchain integrations and operational tooling. A caller that requests too many entries can force excessive memory expansion or simply run out of gas while copying the queue into memory, causing the call to revert. The revert does not corrupt protocol state because the function is read-only, but it does mean the getter is not reliably callable with attacker-controlled or unchecked pagination inputs. Any contract or workflow that depends on this view for queue introspection can therefore self-brick on large requests.

Under the current private-L2 deployment model this is better understood as a low-severity robustness issue than as a direct exploit. The main risk is broken integrations, failed monitoring, or operational friction when tooling assumes the getter will always return a large page successfully.

Recommendation: Add an explicit upper bound to `limit` and fail fast when callers request more than the supported page size. The bound should be chosen to keep the worst-case memory allocation and copy cost comfortably within the chain's execution limits.

A simple pattern is:

```
uint64 internal constant MAX_PENDING_WITHDRAWAL_PAGE_SIZE = 100;

function getPendingWithdrawalRequests(
    uint64 start,
    uint64 limit
) public view returns (PendingWithdrawalRequest[] memory requests) {
    require(limit <= MAX_PENDING_WITHDRAWAL_PAGE_SIZE, "limit too large");
//...
}
```

If the team wants a more forgiving API, it can clip `limit` down to the maximum instead of reverting. In either case, the interface and operator tooling should document the supported page size so callers paginate safely.

Gravity: Fixed in PR 104.

Cantina Managed: Fix verified.

3.3.9 Frozen `nativeTokenVault` trust can break failed-deposit refunds

Severity: Low Risk

Context: NativeBridgeGateway.sol#L30

Description: `NativeBridgeGateway` resolves `nativeTokenVault` once during initialization and stores that address permanently. Later native bridge submissions do not stay pinned to that same bridge-stack snapshot. `bridgeNativeToL2` reads the current `bridgeHub.sharedBridge()` at execution time and routes the deposit through whatever shared bridge the hub points to then, while `receive()` only accepts refund ETH from the old stored `nativeTokenVault`.

This split was introduced in commit a54f925 - fix(cantina-1): accept zkSync native token vault refunds (58)). Before that change, the gateway accepted refund ETH from the live `sharedBridge()` instead of from a frozen stored sender.

That split makes failed native-deposit recovery brittle across upstream bridge-stack migrations. If zkSync rotates the shared bridge or the native-token-vault sender used for refunds, new failed-deposit claims can start reverting when the new refund sender pushes ETH into the gateway. The issue does not let an unprivileged user steal funds, but it can block recovery for legitimate failed deposits until the gateway is upgraded or reconfigured.

Recommendation: Keep the refund sender check aligned with the bridge stack used for new deposits. The cleanest fix is to derive the expected native-token-vault sender from the current `bridgeHub.sharedBridge()` when validating incoming refund ETH, or to add a tightly controlled admin refresh function that resynchronizes `nativeTokenVault` after an upstream bridge migration. The important part is to avoid pinning the refund sender forever while bridge execution continues to follow live bridge-hub configuration.

Gravity: Fixed in PR 69.

Cantina Managed: PR 69 removes the frozen stored sender and changes `receive()` to accept ETH only if `msg.sender == _resolveNativeTokenVaultFromBridgeHub(bridgeHub)`, resolving refund authorization against the live native-token-vault for the current bridge stack.

3.3.10 Recovery is tied to live bridge stack

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

Description: `NativeBridgeGateway` does not bind failed native-deposit recovery to the bridge topology that actually originated a deposit. When `bridgeNativeToL2` submits a native transfer, it resolves the current `sharedBridge` from `bridgeHub` and also checks that the current bridge stack exposes a non-zero native token vault through `_resolveNativeTokenVaultFromBridgeHub`. However, none of that bridge metadata is persisted into `NativeBridgeRecord`. The stored record keeps only `chainId`, `amount`, and `recovered`. Later, when recovery is attempted, `claimAndRecoverFailedNativeDeposit` once again asks `bridgeHub` for the current `sharedBridge` and routes the failed-deposit claim through that live contract. The contract also accepts inbound ETH only from `wrappedNativeToken` or from the native token vault currently returned by `_resolveNativeTokenVaultFromBridgeHub`. In other words, historical recovery is authorized against whatever bridge stack happens to be current at claim time, not against the bridge stack that actually produced the original `bridgeTxHash`.

That is a fragile assumption for an incident-only path that may be exercised long after the original deposit. Native bridge claims are exactly the sort of operation that can remain pending across upstream migrations, bridge upgrades, router replacements, or changes in how the bridge stack forwards refunds. If a failed deposit belongs to an older `sharedBridge`, or if the refund for that historical deposit is emitted by an older native token vault, the gateway has no way to recognize that this is still the correct recovery path for the recorded deposit. A call to `claimAndRecoverFailedNativeDeposit` can target the wrong `sharedBridge` and fail before any ETH is returned. Even if operators or upstream tooling reach the correct historical bridge contract, the final refund can still revert when it hits `receive`, because the gateway only trusts the native token vault that is currently discoverable from the latest `bridgeHub.sharedBridge()` value.

This makes the recovery path depend on mutable third-party configuration that is external to the vault and external to the gateway's own records. The contract effectively assumes that all historical failed native deposits remain recoverable through the newest bridge stack forever, and that the newest native token vault remains the sender of record for old refunds. That may be true for the current deployment today, but the gateway does not verify or snapshot that guarantee anywhere. If the assumption stops holding after a bridge migration, valid historical failed deposits can become unrecoverable through the normal contract flow even though the protocol still has enough information to know which deposit is being recovered.

Recommendation: Persist recovery-critical bridge metadata at submission time instead of recomputing it from the latest bridge configuration during recovery. At minimum, extend `NativeBridgeRecord` to store the exact `sharedBridge` used for the deposit and the trusted refund sender associated with that bridge era, which for the current stack is the resolved native token vault. Then make `claimAndRecoverFailedNativeDeposit` call `claimFailedDeposit` on the stored `sharedBridge` rather than on a freshly resolved one.

The refund authorization check should also be tied to stored recovery metadata rather than to the live `bridgeHub` view. A practical approach is to run claim and recovery atomically, set a transient expected refund sender from the stored record before invoking the bridge claim, and have `receive` accept ETH only from `wrappedNativeToken` or that transient sender during the claim. If the protocol expects long-lived support for multiple bridge eras, keep an explicit migration-aware compatibility policy instead of relying on dynamic lookups of the newest bridge stack.

Gravity: Fixed in PR 71. Recovery no longer depends on the live bridge stack. Each native deposit now snapshots the exact `sharedBridge` and refund sender at submission time, and `claimAndRecoverFailedNativeDeposit` uses that recorded metadata during recovery so that old failed deposits still recover correctly after bridge hub, shared bridge, or native token vault rotation. The design was also simplified by removing the old split manual recovery flow, keeping recovery atomic end-to-end, and restricting `setBridgeHub(...)` to future deposits only without affecting historical recovery.

Cantina Managed: PR 71 correctly addresses this issue.

3.4 Informational

3.4.1 Native vault `receive()` rejects stipend senders

Severity: Informational

Context: NativeVaultGateway.sol#L72-L79

Description: `NativeVaultGateway.receive()` does not fail for all native deposits. It fails specifically for callers that trigger it through Solidity's `.transfer()` or `.send()`, because those helpers only forward the 2300-gas stipend. The `receive()` path immediately calls `_depositToVault`, which wraps ETH and then transfers wrapped ETH to the vault. That work needs much more gas than the stipend allows, so the transaction runs out of gas before the deposit can complete.

The behavior therefore depends on how ETH is sent. A normal `call{value: amount}("")` with enough gas succeeds, but `.transfer()` reverts and `.send()` returns `false`. The issue is not that the gateway cannot accept ETH. The issue is that its convenience `receive()` path is only compatible with full-gas sends, not with stipend-based sends.

This creates an integration and liveness risk. An upstream contract may reasonably try to send ETH with `.transfer()` or `.send()` and assume the gateway will accept it, but that path will fail even though the same deposit would succeed through a regular `call` or the explicit deposit entrypoint.

Recommendation: Document clearly that `NativeVaultGateway.receive()` only works when the sender provides enough gas. Integrations should use either the explicit `depositToVault` entrypoint or a full-gas native `call` path, and should not rely on `.transfer()` or `.send()` to trigger the receive logic. If stipend compatibility is required, the gateway needs a different architecture because the current wrap-and-forward flow cannot fit inside the stipend gas budget.

Gravity: Fixed in PR 64.

Cantina Managed: PR 64 addresses the recommendation by adding documentation that integrations must use `depositToVault()` or a full-gas native `call` and must not rely on `.transfer()` or `.send()`, adding an explicit `NativeDepositRequiresFullGas()` revert in `receive()` when only the stipend gas budget is available, and adding tests that verify `.transfer()` reverts, `.send()` returns `false`, and full-gas calls still succeed.

3.4.2 Native vault gateway can strand dust

Severity: Informational

Context: NativeVaultGateway.sol#L8-L18

Description: `NativeVaultGateway` is designed to behave like a stateless ingress adapter, but it only processes the ETH attached to the current call. Both `depositToVault` and `receive` forward into `_depositToVault`, which wraps exactly `msg.value` and transfers exactly that wrapped amount to the vault. The contract has no mechanism to sweep balances that already exist before the call starts.

That means any unsolicited assets sent to the gateway can remain there permanently. An accidental ETH transfer, a forced ETH balance, or a direct transfer of wrapped native or another ERC20 to the gateway address does not get cleaned up by later deposits. A subsequent user deposit only wraps and forwards the new call's ETH. It does not flush whatever was already sitting in the contract.

Recommendation: Add a controlled rescue path that can sweep unexpected ETH and arbitrary ERC20 balances out of `NativeVaultGateway`. If the intended policy is to treat such balances as unrecoverable donations, document that explicitly and make sure monitoring, runbooks and integration guidance do not assume the gateway can never retain value.

Gravity: Fixed in PR 63. A dust sweep capability was added to `NativeVaultGateway` with documentation noting that it is not expected under normal contract operation.

Cantina Managed: PR 63 adds an admin-only `sweepNative(recipient, amount)` function for stranded ETH and an admin-only `sweepToken(token, recipient, amount)` function for arbitrary ERC20 balances, addressing the reported dust-lock scenario. It also documents the behavior in the architecture docs and runbook.

3.4.3 Overdue withdrawal halt depends on submitter timestamp discipline

Severity: Informational

Context: BaseContract.sol#L73-L82

Description: The new overdue-withdrawal halt is presented as a hard 2 hour deadline, but the check is only as strong as the timestamp chosen by the sequenced caller. `_setSequence()` calls

`_requireNoOverdueWithdrawalRequest(timestamp)` using the incoming `timestamp` argument, and then only enforces that the supplied value is not less than `state.timestamp`.

That means the deadline is not tied to an objective wall clock. A trusted submitter can keep `state.timestamp` flat or advance it more slowly than real time, which prevents queued withdrawals from ever becoming overdue onchain even if they have been pending far longer in practice. The queue stores `enqueuedTimestampNs` from the same sequencer-controlled clock, so both sides of the comparison are derived from submitter-managed values. The public getter `hasOverdueWithdrawalRequest()` compounds this by checking against the current stored `state.timestamp`, which can make the queue appear healthy even when a later sequenced transaction could choose a larger timestamp and immediately trip the halt.

This does not give external users new power under the current trust model, since the submitter already controls sequencing. The issue is that the code does not actually enforce the operational guarantee implied by a fixed 2 hour deadline. The halt is a policy convention for the submitter, not a contract-level time bound.

Recommendation: If the protocol wants a real elapsed-time guarantee, the overdue check should be anchored to a time source that the submitter cannot arbitrarily hold flat. If that is not possible in this architecture, the code and documentation should stop describing the mechanism as a hard deadline and instead treat it as a submitter-enforced service target.

One simple mitigation is to make the operator tooling monitor queue age against an external clock and alert independently of `state.timestamp`. If the contract must enforce the halt itself, the team should redesign the timing source or introduce an explicit operator-controlled pause that does not pretend to be an objective timeout.

Gravity: Acknowledged.

Cantina Managed: Acknowledged.