THREE SIGMA

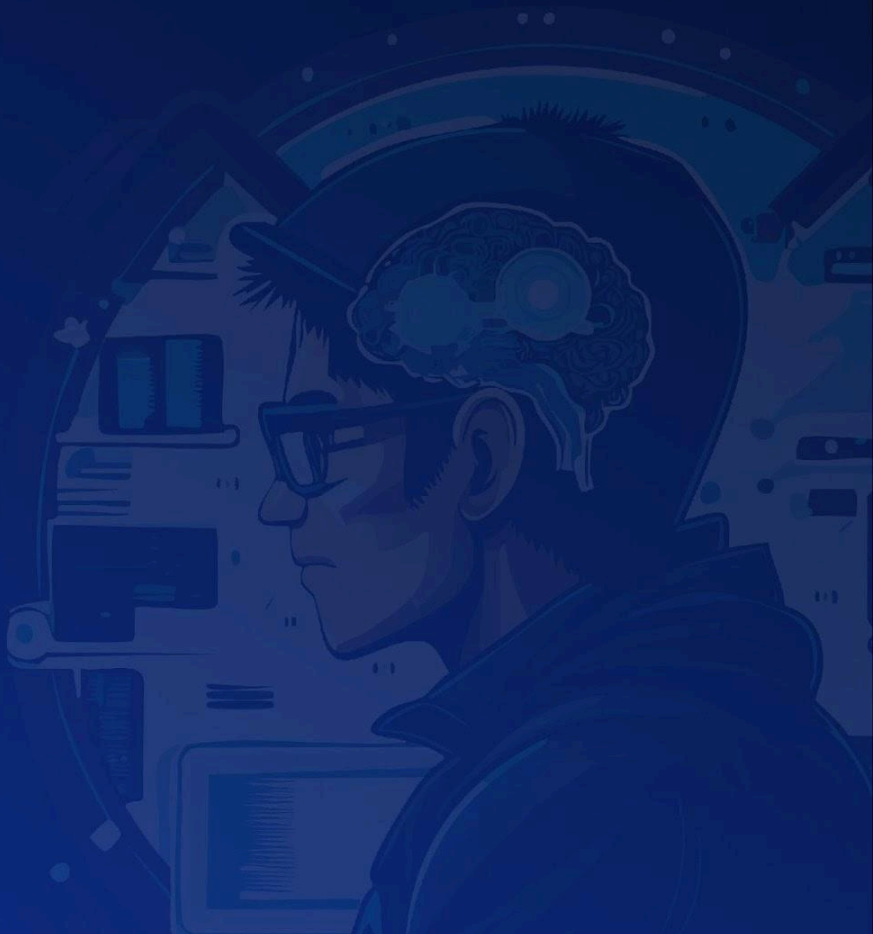
RWA [un]wind

Security Review

Disclaimer

Security Review

RWA [un]wind



Disclaimer

The ensuing audit offers no assertions or assurances about the code's security. It cannot be deemed an adequate judgment of the contract's correctness on its own. The authors of this audit present it solely as an informational exercise, reporting the thorough research involved in the secure development of the intended contracts, and make no material claims or guarantees regarding the contract's post-deployment operation. The authors of this report disclaim all liability for all kinds of potential consequences of the contract's deployment or use. Due to the possibility of human error occurring during the code's manual review process, we advise the client team to commission several independent audits in addition to a public bug bounty program.

Table of Contents

Security Review

RWA [un]wind

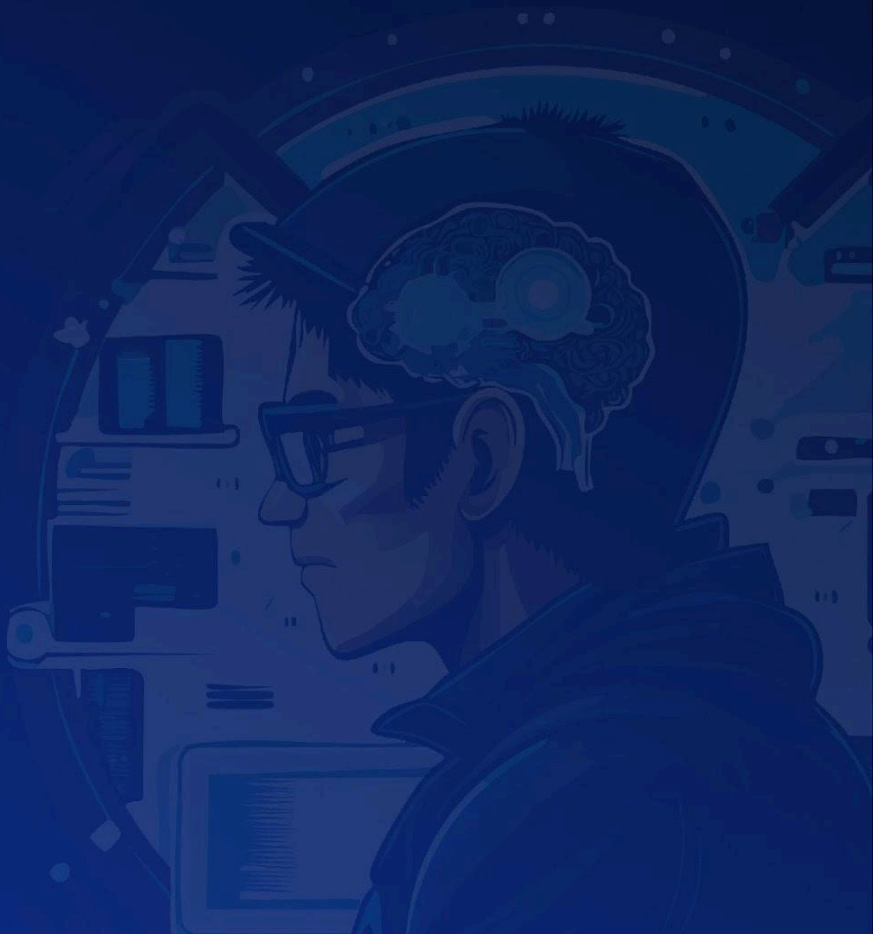
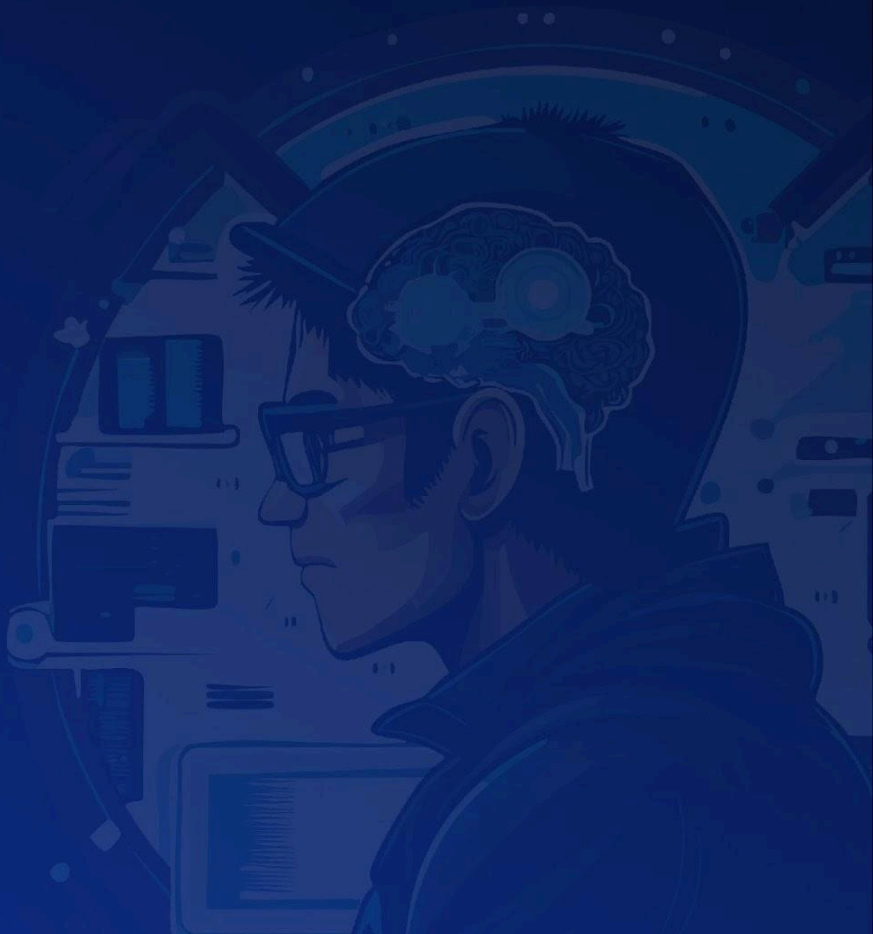


Table of Contents

Summary

Security Review

RWA [un]wind



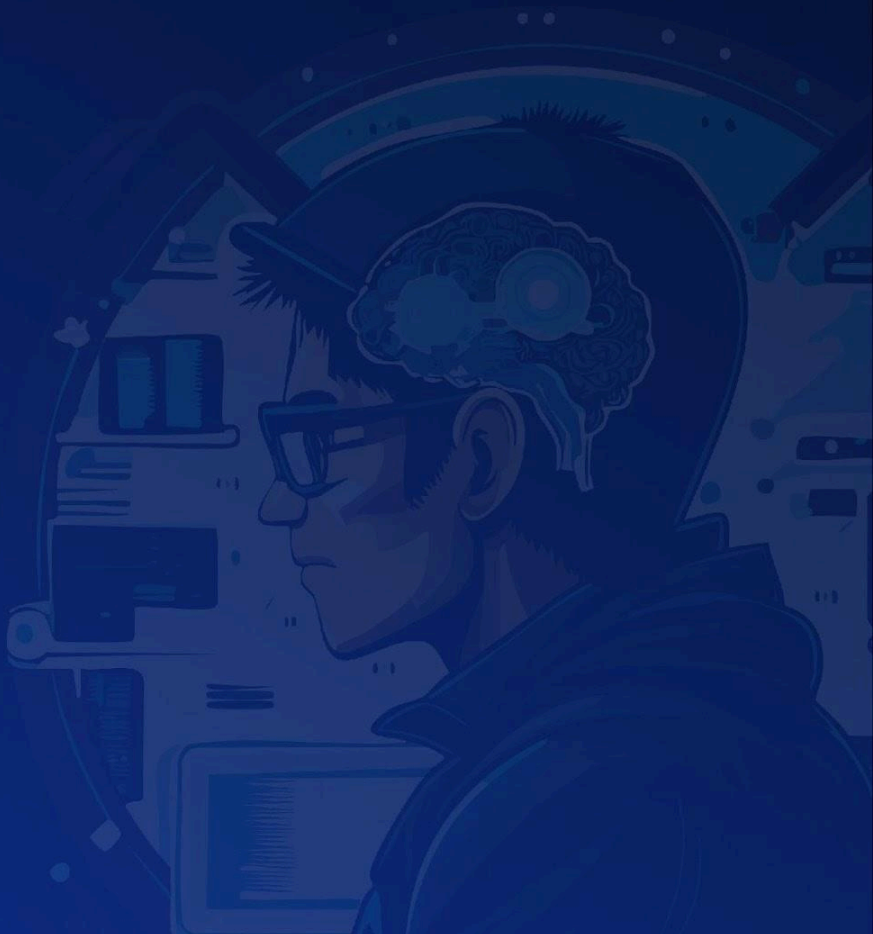
Summary

Three Sigma audited RWA [un]wind in a 8 person week engagement. The audit was conducted from 25/02/2026 to 27/03/2026.

Protocol Description

RWA Unwind is a protocol that manages the full lifecycle of leveraged positions on tokenized Real-World Assets (RWAs) within Euler V2 lending markets. The system consists of two core modules, Wind and Unwind, that enable users to atomically enter and exit leveraged RWA positions despite the asynchronous settlement nature of the underlying assets. During a Wind, the protocol deploys per-request escrow contracts that mint a temporary interim collateral token (PSBA), borrow from Euler, and queue deposits into Pareto credit vaults (Idle Finance); once the RWA shares are claimable, an operator settles the queue and finalizes each position by swapping the interim collateral for real RWA shares and transferring the CDP to the user. During an Unwind, the reverse process locks the user's CDP, replaces RWA collateral with PSBA, queues a redemption through Pareto, and upon settlement returns the underlying proceeds minus fees. The protocol integrates with the Ethereum Vault Connector (EVC) for batched vault operations, Pyth Network for price feed validation, and Keyring Network for KYC gating, and relies on role-gated operators to coordinate queue settlement and liquidation of unhealthy positions.

Scope
Security Review
RWA [un]wind



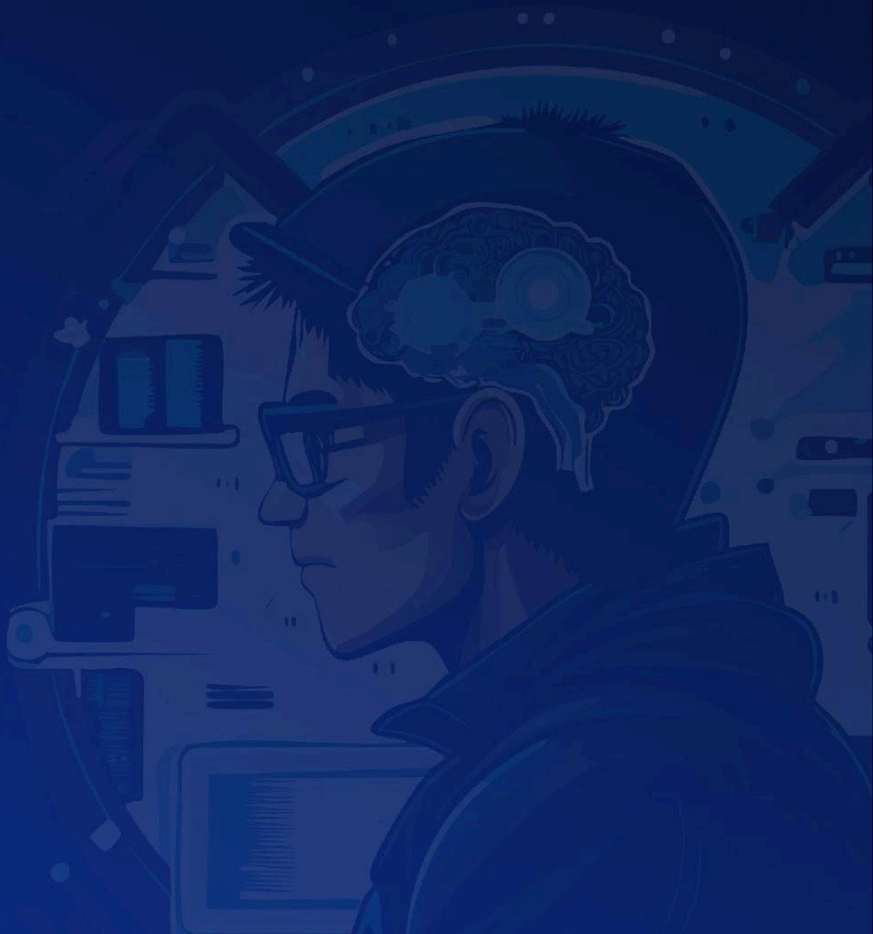
Scope

Filepath	nSLOC
contracts/src/unwind/euler/core/UnwindManagerExtension.sol	585
contracts/src/wind/euler/core/WindManager.sol	487
contracts/src/unwind/euler/core/UnwindManager.sol	318
contracts/src/unwind/euler/core/UnwindEscrow.sol	230
contracts/src/unwind/euler/adapters/ParetoAdapter.sol	214
contracts/src/wind/euler/core/WindEscrow.sol	190
contracts/src/wind/euler/adapters/ParetoAdapter.sol	177
contracts/src/unwind/euler/core/UnwindManagerFactory.sol	135
contracts/src/wind/euler/core/WindManagerFactory.sol	92
contracts/src/unwind/euler/core/UnwindLiquidator.sol	83
contracts/src/unwind/euler/core/UnwindManagerStorage.sol	70
contracts/src/utils/euler/ParetoPriceOracleAdapter.sol	58
contracts/src/utils/euler/PassThroughPriceOracleAdapter.sol	55
contracts/src/libraries/EulerHookLib.sol	40
contracts/src/libraries/QueueLib.sol	40
contracts/src/unwind/euler/core/UnwindManagerDeployer.sol	31
contracts/src/common/PendingSettlementBaseAsset.sol	30
contracts/src/common/BaseKeyringTermsOfService.sol	28
contracts/src/wind/euler/core/WindManagerDeployer.sol	28
contracts/src/libraries/BatchCall.sol	27
contracts/src/wind/common/FeeCollector.sol	27
contracts/src/common/BaseKeyringFeeManager.sol	24
contracts/src/common/BaseKeyringPythUpdate.sol	16
Total	**2985**

Methodology

Security Review

RWA [un]wind



To begin, we reasoned meticulously about the contract's business logic, checking security-critical features to ensure that there were no gaps in the business logic and/or inconsistencies between the aforementioned logic and the implementation. Second, we thoroughly examined the code for known security flaws and attack vectors. Finally, we discussed the most catastrophic situations with the team and reasoned backwards to ensure they are not reachable in any unintentional form.

Taxonomy

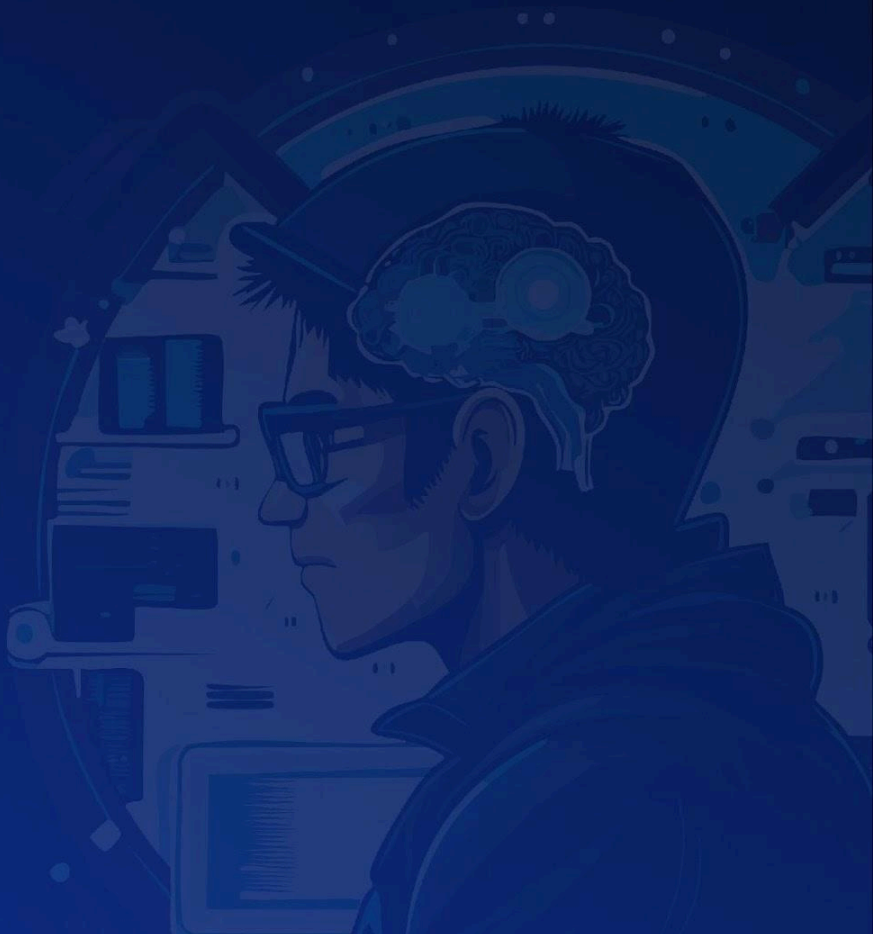
In this audit, we classify findings based on Immunefi's Vulnerability Severity Classification System (v2.3) as a guideline. The final classification considers both the potential impact of an issue, as defined in the referenced system, and its likelihood of being exploited. The following table summarizes the general expected classification according to impact and likelihood; however, each issue will be evaluated on a case-by-case basis and may not strictly follow it.

Impact / Likelihood	LOW	MEDIUM	HIGH
NONE	None		
LOW	Low		
MEDIUM	Low	Medium	Medium
HIGH	Medium	High	High
CRITICAL	High	Critical	Critical

Project Dashboard

Security Review

RWA [un]wind



Project Dashboard

Application Summary

Name	RWA [un]wind
Repository	https://github.com/Keyring-Network/rwa-unwind
Commit	0c461c4
Language	Solidity
Platform	Avalanche

Engagement Summary

Timeline	25/02/2026 to 27/03/2026
№ of Auditors	2
Review Time	8 person weeks

Vulnerability Summary

Issue Classification	Found	Addressed	Acknowledged
Critical	0	0	0
High	7	6	1
Medium	8	6	2
Low	4	2	2
None	13	12	1

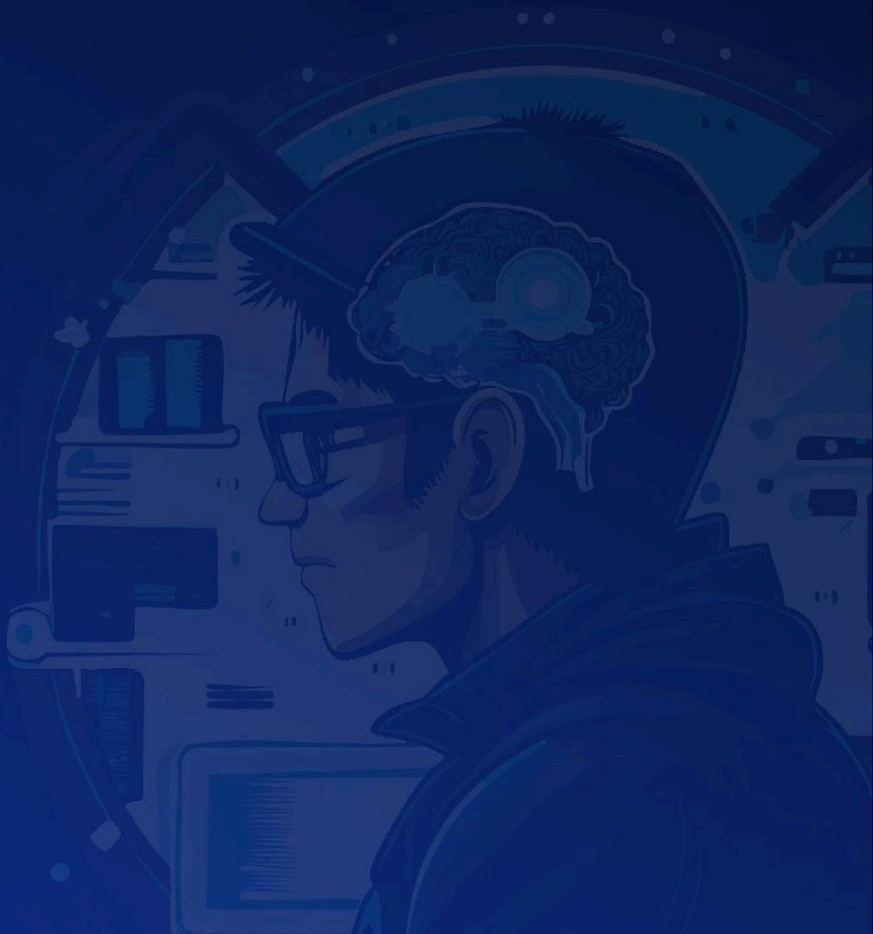
Category Breakdown

Suggestion	13
Documentation	0
Bug	19
Optimization	0
Good Code Practices	0

Risk Section

Security Review

RWA [un]wind



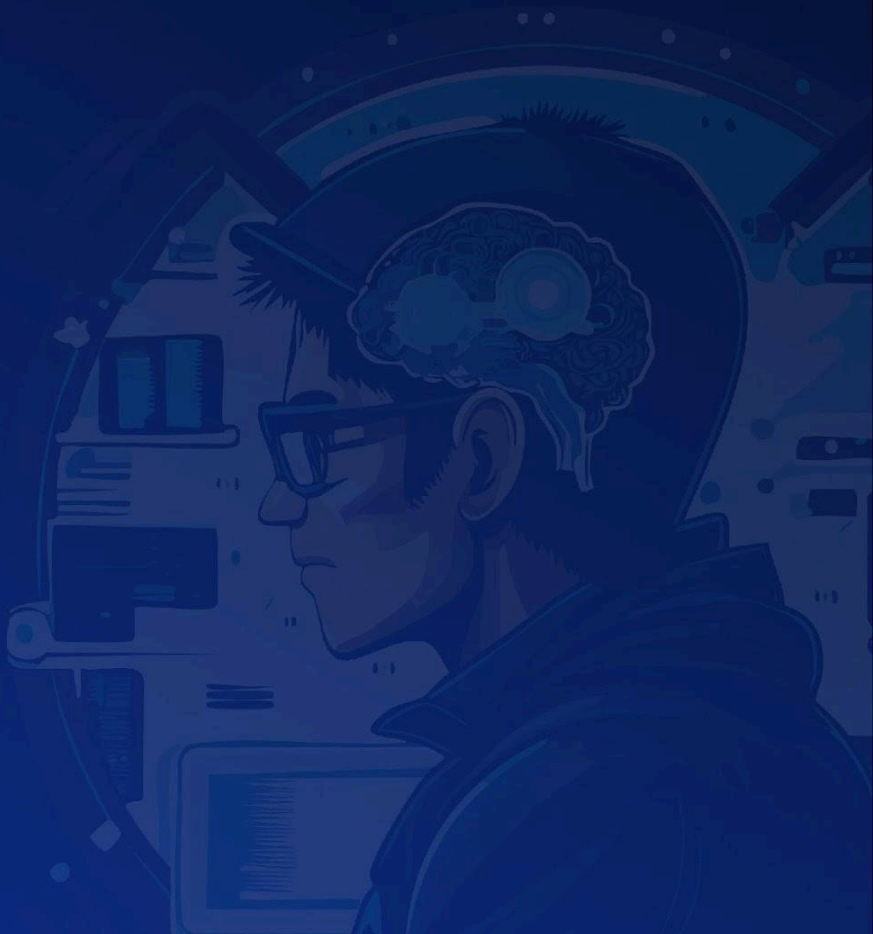
Risk Section

- **Liquidation Settlement Delay**: During the unwind process, positions cannot be liquidated until the asynchronous Pareto redemption settles, meaning a CDP that breaches its liquidation threshold must wait for off-chain settlement before any liquidation can occur, a delay that can allow the position to accrue additional bad debt beyond what would arise in a conventional lending market where liquidation is immediate. The liquidation scenario is highly unlikely.

Findings

Security Review

RWA [un]wind



Findings

3S-H01

Hardcoded 6-decimal scaling in **ParetoPriceOracleAdapter** enables collateral overvaluation for non-USDC underlyings

Id	3S--H01
Classification	High
Impact	High
Likelihood	Medium
Category	Bug
Status	Addressed in #d48d549.

Description

ParetoPriceOracleAdapter implements Euler's **IPriceOracle** interface to price Pareto CDO tranche tokens in USD. It is consumed by **UnwindManager:_getRequesterLtv()** and **UnwindManager._getEquivalentAssets()** for LTV checks, unwind eligibility, and liquidation thresholds.

The adapter's **getQuote** and **getQuotes** functions retrieve **virtualPrice** from the Pareto CDO and rescale it to 18-decimal output. **ParetoPriceOracleAdapter** hardcodes **USD_DECIMALS = 6** to produce a fixed scaling factor of **10^(18 - 6) = 1e12**:

```
uint256 scale = 10 ** (OUTPUT_DECIMALS - USD_DECIMALS);  // always 1e12
return (_inAmount * price * scale) / (10 ** uint256(baseDecimals));
```

Pareto's **IdleCDO.virtualPrice()** returns a value denominated in the underlying token's decimals, not in a fixed 6-decimal format. For a 6-decimal underlying like USDC, **virtualPrice** returns values around **1e6**, and the **1e12** scale factor produces the correct 18-decimal output. For an 18-decimal underlying like USDe, **virtualPrice** returns values around **1e18**, and the same **1e12** factor produces a result that is **1e12** times too large.

Different vaults use different underlying tokens depending on the RWA's subscription mechanics. Pareto supports USDe (18 decimals) in addition to USDC (6 decimals).

The downstream impact materializes in **UnwindManager:_getRequesterLtv()**, which calls **getQuote** for both collateral and debt, then computes **requesterLtv = (debtValue * BPS) /**

collateralValue. An inflated **collateralValue** produces an artificially low LTV, causing **UnwindManager._checkLtv()** to accept positions that are in reality dangerously undercollateralized. With a USDe-backed tranche, the **1e12x** overvaluation makes any position appear to have an LTV of essentially 0%, regardless of actual debt. Similarly, **UnwindManager._getEquivalentAssets()** uses inflated **collateralUnitPrice** values to compute incorrect collateral/debt conversion ratios, under-collateralizing escrow positions during settlement.

Recommendation

Replace the hardcoded **USD_DECIMALS** constant with the actual underlying token's decimals, read from the CDO at construction time. This matches the approach used by Euler's own **IdleTranchesOracle**, which calls **ScaleUtils.calcScale(trancheDecimals, underlyingDecimals, underlyingDecimals)** to derive the scaling dynamically.

```
  /// @notice Decimals used by Pareto virtual price (USD).
- uint256 internal constant USD_DECIMALS = 6;
+ uint256 internal immutable i_underlyingDecimals;
  // ... in constructor:
  i_cdo = _cdo;
  i_tranche = _tranche;
+ i_underlyingDecimals = IERC20Metadata(_cdo.token()).decimals();
```

```
  function getQuote(uint256 _inAmount, address _base, address _quote) external
view returns (uint256) {
      _requireExpectedBase(_base);
      _requireExpectedQuote(_quote);
      uint256 price = i_cdo.virtualPrice(_base);
      uint8 baseDecimals = IERC20Metadata(_base).decimals();
-     uint256 scale = 10 ** (OUTPUT_DECIMALS - USD_DECIMALS);
+     uint256 scale = 10 ** (OUTPUT_DECIMALS - i_underlyingDecimals);
      return (_inAmount * price * scale) / (10 ** uint256(baseDecimals));
  }
```

The same change applies to **ParetoPriceOracleAdapter::getQuotes**.

3S-H02

Liquidation threshold mismatch between **UnwindManagerExtension** and Euler vault allows bad debt accumulation

Id	3S--H02
Classification	High
Impact	High
Likelihood	High
Category	Bug
Status	Addressed in #c5d9ce2.

Description

UnwindManagerExtension::liquidateUnwind allows a liquidation operator to liquidate an unhealthy unwind escrow position. The function checks whether a position is eligible for liquidation, delegates the actual Euler vault liquidation to an **UnwindLiquidator** proxy, then settles all remaining debts using pending settlement funds and distributes the surplus to the liquidation bonus recipient and the original requester.

The liquidation eligibility check in the manager compares raw collateral value against raw debt value:

```
        cache.collateralAssetsPre =
_getPendingSettlementAssets(address(unwindEscrow));
        cache.collateralValuePre =
            _getAssetValue(debtVault, IEVault(collateralVault).asset(),
cache.collateralAssetsPre);
        cache.debtAssetsPre = IEVault(debtVault).debtOf(address(unwindEscrow));
        cache.debtValuePre = _getAssetValue(debtVault, IEVault(debtVault).asset(),
cache.debtAssetsPre);
        if (cache.debtValuePre <= cache.collateralValuePre) {
            revert
UnwindManagerExtension__UnwindRequestNotLiquidatable(_unwindRequestId);
        }
```

This means a position is only considered liquidatable when it is fully underwater (debt value exceeds collateral value). However, in the Euler vault, the liquidation threshold incorporates

the liquidation LTV. Inside **LiquidityUtils::getCollateralValue**, the collateral value is scaled down by the configured LTV before being compared to the liability:

```
    function getCollateralValue(VaultCache memory vaultCache, address account,
address collateral, bool liquidation)
        internal
        view
        virtual
        returns (uint256 value)
    {
        ConfigAmount ltv = getLTV(collateral, liquidation);
        if (ltv.isZero()) return 0;
        uint256 balance = IERC20(collateral).balanceOf(account);
        if (balance == 0) return 0;
        uint256 currentCollateralValue;
        if (liquidation) {
            // mid-point price
            currentCollateralValue = vaultCache.oracle.getQuote(balance, collateral,
vaultCache.unitOfAccount);
        } else {
            // bid price for collateral
            (currentCollateralValue,) = vaultCache.oracle.getQuotes(balance, collateral,
vaultCache.unitOfAccount);
        }
        return currentCollateralValue * ltv.toUint16() / CONFIG_SCALE;
    }
```

Because the manager does not factor in the LTV when assessing liquidation eligibility, there is a range of positions that are liquidatable in Euler but not in the manager (when **collateralValue * LTV < debtValue < collateralValue**). When the manager finally does liquidate an underwater position, the Euler vault computes a discount factor based on the LTV-adjusted collateral, which limits the maximum repayable debt to a fraction of the total liability. The manager then covers both the liquidator's and the escrow's residual Euler debt from its own pending settlement funds, but the recovered collateral is insufficient to offset the total debt repaid plus the bonus distributed to **s_liquidationBonusRecipient**.

The impact is that the manager's pending settlement pool absorbs bad debt on every such liquidation. These funds belong to other unwind requests in the settlement queue, meaning unrelated users' pending redemptions are used to subsidize the shortfall. Over multiple liquidations this can progressively drain the settlement pool, potentially leaving later unwind requests unable to finalize due to insufficient funds.

Steps to Reproduce

1. An unwind escrow position has 100 units of debt, 90 units of collateral, and the Euler liquidation LTV is 80%.

2. In the Euler vault, the position is unhealthy because the LTV-adjusted collateral value (90 * 0.8 = 72) is less than the debt (100). The discount factor is 72 / 100 = 0.72.

3. In the manager, the position is also liquidatable because **debtValuePre** (100) exceeds **collateralValuePre** (90). However, the manager does not account for LTV when computing the discount, whereas Euler does. This means the Euler vault uses a discount factor of 0.72 (derived from the LTV-adjusted collateral), which limits **maxRepayValue** to 90 * 0.72 = 64.8, seizing all 90 units of collateral.

4. After the Euler liquidation, the liquidator holds 90 collateral and 64.8 debt; the escrow retains 35.2 in unsettled debt (100 - 64.8).

5. The manager transfers debt tokens to both the liquidator (64.8) and the escrow (35.2) to settle their Euler debts, totaling 100 debt tokens.

6. Through **settleDebtsAndTransfer** and **withdrawPendingSettlementAssets**, the manager recovers only the equivalent of 90 units from the seized collateral.

7. The liquidation bonus recipient receives 90 - 64.8 = 25.2 units as bonus.

8. The manager is left with a deficit of 100 - (90 - 25.2) = 35.2 debt tokens, representing unrecoverable bad debt borne by the protocol's pending settlement pool.

Recommendation

Align the manager's liquidation eligibility check with the Euler vault's LTV-adjusted threshold. Rather than comparing raw collateral and debt values, the check should incorporate the liquidation LTV configured in the Euler vault for the relevant collateral so that the manager can trigger liquidation at the same point Euler considers the position unhealthy. This also allows the manager to act before the position becomes fully underwater, reducing or eliminating the bad debt exposure. Consider querying the liquidation LTV from the Euler vault's LTV configuration and applying it to the collateral value in the eligibility check, matching the logic in **LiquidityUtils::getCollateralValue**. Additionally, review the bonus calculation and debt settlement logic to ensure the manager does not distribute more surplus than it can recover from the liquidation proceeds.

3S-H03

Hook bypass roles are granted to **requester** instead of **subAccount** and pre-existing roles are unconditionally stripped in **WindManager::finalizeWind**

Id	3S--H03
Classification	High
Impact	High
Likelihood	Medium
Category	Bug
Status	Addressed in #67f57fe.

Description

WindManager::finalizeWind temporarily grants **WILDCARD_ROLE** and **PRIVILEGED_ACCOUNT_ROLE** on the Euler vault hook targets to **escrowState.requester**, then revokes them after the escrow finalizes. These roles allow the granted address to bypass KYC/credential checks enforced by the Keyring hook target on vault operations.

However, during **WindEscrow::finalizeWind**, the escrow calls **_transferCdpToSubAccount**, which executes vault operations on behalf of two accounts: the escrow itself (which already has roles from **requestWind**) and **s_subAccount**. Specifically, the **pullDebt** call in the EVC batch is executed with **onBehalfOfAccount: s_subAccount**. Inside the Euler vault, **Borrowing::pullDebt** calls **initOperation**, which resolves the EVC-authenticated caller via **EVCAuthenticateDeferred**. Since the batch item specifies **s_subAccount** as the account, the vault's hook is invoked with **s_subAccount** as the caller to check. The hook target then verifies whether **s_subAccount** holds **WILDCARD_ROLE** or the appropriate selector role.

Because the roles were granted to **escrowState.requester** rather than **escrowState.subAccount**, the hook check passes only when **requester == subAccount** (the default in current tests). If a user provides a different EVC sub-account (e.g., **requester | 0x01**), the hook target will not find the bypass roles on **s_subAccount** and the transaction will revert.

Additionally, the grant-then-revoke pattern unconditionally strips any pre-existing roles from the target address. If **escrowState.requester** had been independently granted permanent **WILDCARD_ROLE** or **PRIVILEGED_ACCOUNT_ROLE** on the hook target (e.g., as a whitelisted participant or partner integration), those roles would be removed after

finalization completes. The revoke path in **EulerHookLib::grantOrRevokeRolesIfHooked** does not distinguish between roles that were newly granted by the current call and roles that pre-existed.

Steps to Reproduce

1. Deploy the protocol with KYC hooks enabled (**s_policyId != 0**) on the vault cluster.

2. A user calls **WindManager::requestWind** with **_subAccount** set to an EVC sub-account that differs from their main address (e.g., **requester | 0x01**).

3. The operator settles the queue so the request becomes fully claimable.

4. The user or operator calls **WindManager::finalizeWind**.

5. The function grants **WILDCARD_ROLE** and **PRIVILEGED_ACCOUNT_ROLE** to **escrowState.requester** on the hook targets.

6. Inside **WindEscrow::_transferCdpToSubAccount**, the **pullDebt** EVC batch item executes on behalf of **s_subAccount**.

7. The debt vault invokes the hook target, which checks whether **s_subAccount** has **WILDCARD_ROLE**. Since roles were granted to **requester** (a different address), the check does not pass and the transaction reverts.

Recommendation

The recommended fix is to grant and revoke the hook bypass roles on **escrowState.subAccount** instead of **escrowState.requester** in **WindManager::finalizeWind**. The **subAccount** is the address that the Euler vault's hook target actually validates during the **pullDebt** operation, so it is the address that needs the temporary bypass. The **requester** address is not the EVC-authenticated account for any vault operation during the finalization flow. The same grant-then-revoke pattern should also account for pre-existing roles to avoid stripping permanent permissions from the target address. One approach is to check whether each role already existed before granting and skip the corresponding revoke if it did.

An alternative approach is to modify the hook target to be EVC-aware, resolving the owner of a sub-account and checking the owner's roles instead of the sub-account's roles directly. This would allow granting roles to the **requester** and having them apply to all of the requester's sub-accounts. However, this is not recommended because it offloads additional lookup and validation logic to the hook target, increasing gas costs on every hooked vault operation and introducing unnecessary coupling between the hook target and EVC sub-account semantics.

3S-H04

User can permanently trap protocol funds by refusing to finalize a wind position

Id	3S--H04
Classification	High
Impact	High
Likelihood	Medium
Category	Bug
Status	Addressed in #4521d26.

Description

WindManager::requestWind enables users to open leveraged RWA positions. The user deposits equity, the protocol borrows additional base assets on behalf of the user, and the combined amount is sent to the adapter for async settlement. The user provides a **_subAccount** parameter at request time, and the **WindEscrow** stores this address as the final destination for the completed CDP.

Once the adapter settles the deposit, **WindManager::finalizeWind** is called (by the user or an operator with **KEYRING_OPERATOR_ROLE**) to transfer the finalized CDP (RWA collateral shares + debt) to the user's sub-account. Internally, **WindEscrow::finalizeWind** calls **_transferCdpToSubAccount**, which executes an EVC batch that operates on **s_subAccount**, including **enableCollateral**, **enableController**, **pullDebt**, and **setAccountOperator**. These operations require the escrow to be authorized as an EVC operator for the sub-account.

The sub-account's operator authorization is entirely under the user's control. If the user never grants operator status to the escrow, or revokes it before finalization, the **_transferCdpToSubAccount** call reverts. Because the wind system has no on-chain liquidation mechanism by design, the protocol's borrowed funds remain permanently locked in the escrow.

The impact scales with the configured leverage. For example, with 3x leverage, the user deposits 1 USDC of equity but the protocol borrows 2 USDC. By refusing to finalize, the attacker sacrifices 1 USDC to trap 2 USDC of protocol-supplied debt indefinitely.

Additionally, **WindManager::finalizeWind** enforces **_requireToSAccepted(escrowState.requester)**. If Terms of Service are updated after the wind request, finalization becomes impossible through the normal path regardless of sub-account operator status.

Steps to Reproduce

1. Attacker calls **WindManager::requestWind** with a **_subAccount** address but never sets the **WindEscrow** as an EVC operator for that sub-account.

2. The protocol borrows base assets on behalf of the attacker and sends the total (equity + borrowed amount) to the adapter.

3. The adapter settles the deposit and shares become claimable. The operator calls **settleQueue** to earmark shares for the request.

4. An operator with **KEYRING_OPERATOR_ROLE** calls **WindManager::finalizeWind** for the request.

5. The call reverts inside **WindEscrow::_transferCdpToSubAccount** because the escrow is not an authorized operator for the sub-account.

6. The CDP (collateral + debt) remains permanently locked in the escrow with no alternative recovery path.

Recommendation

Introduce an emergency finalization path. When **WindManager::finalizeWind** reverts, whether due to the user revoking EVC operator authorization on their sub-account, **_requireToSAccepted** reverting because the Terms of Service were updated after the request was created, or **IWindEscrow(escrow).finalizeWind** reverting for any reason, the request should be recorded as failed in a mapping (e.g. **s_failedFinalizations**). A separate **emergencyFinalizeWind** function, restricted to **DEFAULT_ADMIN_ROLE**, should allow the admin to finalize the request by providing an admin-controlled sub-account, bypassing both the user-controlled **s_subAccount** and the ToS checks on the original requester. The **WindEscrow** should expose a corresponding **emergencyFinalizeWind** function that accepts the admin sub-account and delegates to a refactored **_transferCdpToSubAccount** that takes the target sub-account as a parameter.

This two-step design ensures the emergency path cannot be abused: the normal **finalizeWind** must have reverted at least once before the emergency function becomes available for a given request.

3S-H05

Unwind process lacks a cancellation flow, causing stuck queue and stranded collateral when write-off requests are deleted

Id	3S--H05
Classification	High
Impact	High
Likelihood	Medium
Category	Bug
Status	Acknowledged with note: *"The write-off escrow is the only way to redeem and thus, we would ensure that no cancellations are ever needed."*

Description

The **ParetoAdapter** bridges the **UnwindManager** to Pareto's **IdleCreditVaultWriteOffEscrow**, a P2P marketplace where the adapter (seller) sets a price and an external counterparty (buyer) fulfills the request. When a user calls **UnwindManager::requestUnwind**, the manager converts the user's collateral shares into collateral assets, forwards them to **ParetoAdapter::requestRedeem**, and the adapter deposits those assets into the escrow via **IdleCreditVaultWriteOffEscrow::createWriteOffRequest**. The request is then enqueued in **s_pendingRequestQueue** for later settlement by **UnwindManagerExtension::settleQueue**.

In practice, the adapter may need to cancel a pending write-off request in the escrow. This can happen when:

- The authorized signer issues a quote price that is incorrect.

- Market conditions change such that the quoted price becomes unattractive, and no counterparty is willing to fulfill the request at that price. The request remains unfulfilled indefinitely, blocking the entire unwind pipeline since new batches cannot be created while **s_hasPendingRedeem** is **true**.

The adapter provides two emergency functions for this scenario: **ParetoAdapter::batchExternalCalls** (which can call **IdleCreditVaultWriteOffEscrow::deleteWriteOffRequest**) and **ParetoAdapter::resetBatchState**. Both require the manager to be paused and the caller to

hold the **SAFETY_MANAGER_ROLE**. However, using these functions to cancel a request leaves the system in an inconsistent state:

1. The collateral assets that were transferred from the users' escrows to the adapter and then deposited into the write-off escrow are returned to the **ParetoAdapter** upon deletion, but there is no mechanism in **UnwindManager** to redistribute these assets back to the individual users' unwind escrows.

2. The cancelled requests remain in **s_pendingRequestQueue** in the **UnwindManager**. The queue is append-only with FIFO ordering, and only **_settleQueueIfClaimable** and **liquidateUnwind** can pop entries. When **settleQueue** is subsequently called after a new batch is submitted, it attempts to settle the stale entries at the head of the queue. These entries reference collateral amounts that no longer correspond to any active write-off request in the adapter, resulting in incorrect asset distribution.

Steps to Reproduce

1. Alice calls **UnwindManager::requestUnwind** with a signed quote price of X. The manager transfers her collateral to the adapter, the adapter calls **IdleCreditVaultWriteOffEscrow::createWriteOffRequest**, and the request is enqueued in **s_pendingRequestQueue**.

2. Bob calls **UnwindManager::requestUnwind** with the same quote price. His collateral is added to the same batch in the adapter, and his request is also enqueued.

3. The market price moves significantly away from X, and no counterparty fulfills the write-off request.

4. The safety manager pauses the **UnwindManager** and calls **ParetoAdapter::batchExternalCalls** targeting **IdleCreditVaultWriteOffEscrow::deleteWriteOffRequest** to cancel the stale escrow request.

5. The safety manager calls **ParetoAdapter::resetBatchState** to clear the adapter's batch tracking.

6. The collateral assets are now stranded in the **ParetoAdapter** with no function in **UnwindManager** to return them to Alice's and Bob's unwind escrows.

7. After unpausing, a new batch is submitted with a corrected quote price. When **settleQueue** is called, Alice's and Bob's stale entries are still at the front of the queue. The settlement logic in **_settleQueueIfClaimable** allocates assets from the new batch to these stale entries, corrupting the accounting for all subsequent requests.

Recommendation

Introduce a cancellation path in the **UnwindManager** (via a new extension function) that is callable by the **SAFETY_MANAGER_ROLE** while the contract is paused. This function

should remove specified request IDs from **s_pendingRequestQueue**, return the collateral from the adapter back to each user's unwind escrow (or directly to the receipt NFT holder), and clean up the associated accounting state (**s_requestIdToClaimableAssets**, **s_requestIdToClaimableShares**, **s_pendingRequestIdsByUser**).

3S-H06

Requester payout in **liquidateUnwind** is incorrectly derived from caller-supplied **_repayAssets** instead of actual accounting

Id	3S--H06
Classification	High
Impact	High
Likelihood	High
Category	Bug
Status	Addressed in #f321f55.

Description

UnwindManagerExtension::liquidateUnwind liquidates an underwater unwind escrow position. After executing the Euler vault liquidation and settling all debts, the function distributes the remaining debt asset balance: a liquidation bonus goes to the configured recipient and the residual goes to the original position owner (the requester).

The function computes **cache.availableBaseAssets** as the debt token balance difference on the manager before and after the settlement calls (**settleDebtsAndTransfer** and **withdrawPendingSettlementAssets**). This value is then used to derive **cache.requesterPayout**:

```
    cache.availableBaseAssets = cache.debtAsset.balanceOf(address(this)) -
balanceBefore;
    if (cache.bonusAssets > cache.availableBaseAssets) {
      revert UnwindManagerExtension__LiquidationBonusExceedsAvailable(
        cache.bonusAssets, cache.availableBaseAssets
      );
    }
    if (cache.bonusAssets > 0) {
      SafeERC20.safeTransfer(cache.debtAsset, s_liquidationBonusRecipient,
cache.bonusAssets);
    }
    cache.requesterPayout = cache.availableBaseAssets - cache.bonusAssets;
```

However, the balance difference does not reflect the correct accounting. The flow proceeds as follows:

1. The manager transfers **_repayAssets** (a caller-supplied parameter) in debt tokens to the liquidator before calling **executeLiquidation**. This transfer is unnecessary as the Euler vault **liquidate** call only performs internal accounting; this is addressed in a separate issue.

2. The manager then transfers **cache.liquidatorDebt** and **cache.escrowDebt** to the liquidator and escrow respectively for debt repayment.

3. **balanceBefore** is recorded on the manager.

4. **UnwindLiquidator::settleDebtsAndTransfer** repays the liquidator's debt using the **cache.liquidatorDebt** tokens and transfers the remaining balance back to the manager. The remaining balance is exactly **_repayAssets** (the unused tokens from step 1).

5. **UnwindEscrow::withdrawPendingSettlementAssets** repays the escrow's debt and withdraws pending settlement base asset (PSBA) vault shares to the manager. It does not transfer any debt tokens back to the manager.

As a result, **cache.availableBaseAssets** equals **_repayAssets** (the only debt tokens returned to the manager between the balance snapshots), and **cache.requesterPayout** equals **_repayAssets - cache.bonusAssets**. Since **_repayAssets** is an arbitrary caller-supplied parameter constrained only by **cache.maxRepay**, the payout to the requester is disconnected from the actual settled value of the position.

The correct requester payout should be derived from **cache.claimableAssets** (the settled redemption proceeds for this request) minus the total debt repaid (**cache.liquidatorDebt + cache.escrowDebt**) minus **cache.bonusAssets**.

Steps to Reproduce

1. An unwind escrow position becomes underwater (debt value exceeds collateral value).

2. A liquidation operator calls **liquidateUnwind** with a **_repayAssets** value that differs from **cache.claimableAssets - cache.liquidatorDebt - cache.escrowDebt**.

3. The Euler vault liquidation executes and debts are settled on both the liquidator and escrow.

4. **cache.availableBaseAssets** is computed as **_repayAssets** because only the refunded first transfer contributes to the manager's balance difference.

5. The requester receives **_repayAssets - cache.bonusAssets** instead of the correct residual: **cache.claimableAssets - cache.liquidatorDebt - cache.escrowDebt - cache.bonusAssets**.

6. Depending on the **_repayAssets** value relative to the correct residual, the requester is either overpaid (draining the manager's funds from other requests) or underpaid (funds remain stuck in the manager).

Recommendation

Compute **cache.availableBaseAssets** directly from the position's accounting rather than relying on a balance difference.

```
-       uint256 balanceBefore = cache.debtAsset.balanceOf(address(this));
        (, cache.liquidatorPendingSettlementAssets) =
liquidator.settleDebtsAndTransfer(address(this));
        cache.escrowPendingSettlementAssets =
unwindEscrow.withdrawPendingSettlementAssets(address(this));
        cache.totalPendingSettlementAssets =
            cache.liquidatorPendingSettlementAssets +
cache.escrowPendingSettlementAssets;
        if (cache.totalPendingSettlementAssets > 0) {
            s_pendingSettlementBaseAsset.burn(address(this),
cache.totalPendingSettlementAssets);
        }
-       cache.availableBaseAssets = cache.debtAsset.balanceOf(address(this)) -
balanceBefore;
+        cache.availableBaseAssets = cache.claimableAssets - cache.liquidatorDebt -
cache.escrowDebt;
```

3S-H07

liquidateUnwind executes on partially settled redemption requests

Id	3S--H07
Classification	High
Impact	High
Likelihood	Medium
Category	Bug
Status	Addressed in #9b12c1f.

Description

UnwindManagerExtension::liquidateUnwind allows a liquidation operator to liquidate an unhealthy unwind escrow whose debt value exceeds its collateral value. The function settles the escrow's full collateral position, distributes redeemed assets between the liquidation bonus recipient and the original requester, and burns the corresponding pending settlement assets.

Before executing, **liquidateUnwind** verifies that **s_requestIdToClaimableAssets[_requestId]** is non-zero. However, it does not verify that all shares associated with the request have been fully claimed. This is significant because **settleQueue** can partially settle a request when the **_maxShares** parameter limits how many shares are claimed per call. In that scenario, **s_requestIdToClaimableAssets** is non-zero (reflecting a partial claim), but **s_requestIdToClaimableShares** is still less than **collateralAssetsToRedeem**.

By contrast, **finalizeUnwind** includes the correct guard:

```
        if (s_requestIdToClaimableShares[_requestId] <
vaultState.collateralAssetsToRedeem) {
            revert
UnwindManagerExtension__UnwindRequestNotFullySettled(_requestId);
        }
```

When **liquidateUnwind** proceeds on a partially settled request, it zeroes out both **s_requestIdToClaimableAssets** and **s_requestIdToClaimableShares**, and settles the escrow's entire pending settlement position. However, only a fraction of the shares have actually been redeemed into base assets. The remaining shares eventually become claimable at the adapter level but have no corresponding request to settle against, since the

request's accounting has already been cleared. These assets become permanently stranded.

Steps to Reproduce

1. A user initiates an unwind request that requires redeeming 100 collateral shares.

2. An operator calls **settleQueue** with **_maxShares** set to 50, partially settling the request. At this point, **s_requestIdToClaimableShares** is 50 and **s_requestIdToClaimableAssets** is non-zero.

3. Before the remaining 50 shares become claimable and a second **settleQueue** call is made, the escrow's position becomes unhealthy (debt value exceeds collateral value).

4. A liquidation operator calls **liquidateUnwind**. The function passes the **claimableAssets == 0** check because 50 shares' worth of assets have been claimed.

5. **liquidateUnwind** zeroes out **s_requestIdToClaimableAssets** and **s_requestIdToClaimableShares**, then settles and burns the escrow's full pending settlement position.

6. When the remaining 50 shares become claimable at the adapter, no request exists to absorb them. These assets are stranded with no recovery path.

Recommendation

Add the same full-settlement guard present in **finalizeUnwind** to **liquidateUnwind**, immediately after the existing **claimableAssets** check:

```
        LiquidateUnwindCache memory cache;
        cache.claimableAssets = s_requestIdToClaimableAssets[_requestId];
        if (cache.claimableAssets == 0) {
            revert UnwindManagerExtension__UnwindRequestNotClaimable(_requestId);
        }
+       if (s_requestIdToClaimableShares[_requestId] <
state.collateralAssetsToRedeem) {
+           revert
UnwindManagerExtension__UnwindRequestNotFullySettled(_requestId);
+       }

        address requester = ownerOf(state.receiptNftId);
```

3S-M01

ParetoPriceOracleAdapter::getQuote returns underlying-denominated value instead of USD

Id	3S--M01
Classification	Medium
Impact	Medium
Likelihood	High
Category	Bug
Status	Addressed in #aaf6761.

Description

ParetoPriceOracleAdapter::getQuote serves as the Euler-compatible oracle adapter for pricing Pareto tranche tokens (e.g., pACRDX) against the vault's unit of account, which is USD (**address(0x348)**). It is consumed by **UnwindManagerExtension::_getAssetValue** to compute **cache.liquidatorCollateralValue** during liquidation bonus accounting in **liquidateUnwind**.

The function retrieves **virtualPrice** from the Pareto CDO via **IIdleCDOEpochVariant::virtualPrice**. This value represents the tranche token's worth in terms of the CDO's underlying asset (e.g., USDC), scaled to the underlying's decimals. The Idle CDO computes it as:

```
_virtualPrice = uint256(int256(_lastTrancheNAV) + _totalTrancheGain) *
ONE_TRANCHE_TOKEN / trancheSupply;
```

where **_lastTrancheNAV** is denominated in underlying tokens. For a USDC-backed CDO, **virtualPrice** is in 6-decimal USDC terms (e.g., **1.05e6** meaning 1 tranche = 1.05 USDC).

The adapter then scales this value from **USD_DECIMALS** (6) to **OUTPUT_DECIMALS** (18) and returns it as though it were a USD quote:

```
    function getQuote(uint256 _inAmount, address _base, address _quote) external
view returns (uint256) {
        _requireExpectedBase(_base);
        _requireExpectedQuote(_quote);
        uint256 price = i_cdo.virtualPrice(_base);
```

```
        uint8 baseDecimals = IERC20Metadata(_base).decimals();
        uint256 scale = 10 ** (OUTPUT_DECIMALS - USD_DECIMALS);
        return (_inAmount * price * scale) / (10 ** uint256(baseDecimals));
    }
```

This implicitly treats 1 USDC as exactly $1.00. However, the debt-side oracle (a Chainlink USDC/USD adapter, e.g., 0x6213...3F) returns actual USD prices that account for potential USDC depegs. The two values are then compared in **liquidateUnwind**:

```
        cache.liquidatorCollateralAssets =
_getPendingSettlementAssets(address(liquidator));
        cache.liquidatorCollateralValue =
            _getAssetValue(debtVault, IEVault(collateralVault).asset(),
cache.liquidatorCollateralAssets);
        cache.liquidatorDebtAssets = IEVault(debtVault).debtOf(address(liquidator));
        cache.liquidatorDebtValue = _getAssetValue(debtVault,
IEVault(debtVault).asset(), cache.liquidatorDebtAssets);
        if (cache.liquidatorCollateralValue > cache.liquidatorDebtValue) {
            cache.bonusAssets =
                _getDebtAssetAmountFromValue(debtVault, cache.liquidatorCollateralValue
- cache.liquidatorDebtValue);
```

Because **cache.liquidatorCollateralValue** is denominated in USDC (via **ParetoPriceOracleAdapter**) while **cache.liquidatorDebtValue** is denominated in USD (via the Chainlink adapter), the two values are in different denominations. Their comparison and subtraction is therefore incorrect, and the resulting **_value** passed into **_getDebtAssetAmountFromValue** is derived from the difference of two incompatible units, producing an incorrect **bonusAssets** calculation.

The **getQuotes** function contains the same issue.

Recommendation

Introduce a reference to the underlying asset's USD oracle so that **getQuote** first converts the tranche amount to underlying units via **virtualPrice**, then converts the underlying amount to USD via a proper price feed:

```
+ import { IPriceOracle } from "src/vendors/euler/interfaces/IPriceOracle.sol";
  contract ParetoPriceOracleAdapter is IParetoPriceOracleAdapter {
-     uint256 internal constant USD_DECIMALS = 6;
-     uint256 internal constant OUTPUT_DECIMALS = 18;
```

```
    address internal constant QUOTE_ASSET = address(0x348);
    IIdleCDOEpochVariant internal immutable i_cdo;
    address internal immutable i_tranche;
+    IPriceOracle internal immutable i_underlyingOracle;
+    address internal immutable i_underlying;
-    constructor(IIdleCDOEpochVariant _cdo, address _tranche) {
+    constructor(
+        IIdleCDOEpochVariant _cdo,
+        address _tranche,
+        IPriceOracle _underlyingOracle,
+        address _underlying
+    ) {
        _requireNonZeroAddress(address(_cdo));
        _requireNonZeroAddress(_tranche);
+        _requireNonZeroAddress(address(_underlyingOracle));
+        _requireNonZeroAddress(_underlying);
        i_cdo = _cdo;
        i_tranche = _tranche;
+        i_underlyingOracle = _underlyingOracle;
+        i_underlying = _underlying;
    }
    function getQuote(uint256 _inAmount, address _base, address _quote) external
view returns (uint256) {
        _requireExpectedBase(_base);
        _requireExpectedQuote(_quote);
        uint256 price = i_cdo.virtualPrice(_base);
        uint8 baseDecimals = IERC20Metadata(_base).decimals();
-        uint256 scale = 10 ** (OUTPUT_DECIMALS - USD_DECIMALS);
-
-        return (_inAmount * price * scale) / (10 ** uint256(baseDecimals));
+        uint256 underlyingAmount = (_inAmount * price) / (10 **
uint256(baseDecimals));
+        return i_underlyingOracle.getQuote(underlyingAmount, i_underlying, _quote);
    }
    function getQuotes(uint256 _inAmount, address _base, address _quote) external
view returns (uint256, uint256) {
        _requireExpectedBase(_base);
        _requireExpectedQuote(_quote);
        uint256 price = i_cdo.virtualPrice(_base);
        uint8 baseDecimals = IERC20Metadata(_base).decimals();
-        uint256 scale = 10 ** (OUTPUT_DECIMALS - USD_DECIMALS);
-        uint256 amountOut = (_inAmount * price * scale) / (10 **
uint256(baseDecimals));
-
```

```
-         return (amountOut, amountOut);
+         uint256 underlyingAmount = (_inAmount * price) / (10 **
uint256(baseDecimals));
+         return i_underlyingOracle.getQuotes(underlyingAmount, i_underlying,
_quote);
      }
  }
```

This ensures that **getQuote** returns a value in the same denomination (USD) as the debt-side Chainlink adapter, making the **bonusAssets** computation in **liquidateUnwind** and the conversion in **_getDebtAssetAmountFromValue** arithmetically consistent.

3S-M02

Implicit price-consistency assumption in **paretoAdapter** batching

Id	3S--M02
Classification	Medium
Impact	Medium
Likelihood	Medium
Category	Bug
Status	Acknowledged with note: *"All users within a single batch redeem at the same NAV."*

Description

In the **UnwindManager::requestUnwind** flow, **_params.adapterData** is forwarded to **ParetoAdapter::requestRedeem**, where it carries a signed **RedeemAuthorization(shares, underlyingsRequested, expiry)**. The adapter uses this to call **i_writeOffEscrow.createWriteOffRequest(shares, underlyingsRequested)**, which is *additive*: successive calls accumulate into a single pending write-off request on the escrow.

The write-off escrow operates as a P2P settlement mechanism. The adapter (as the seller/requester) specifies a price, i.e. how many underlying assets it wants per tranche share, and the borrower fulfills the request by delivering those assets. An authorized signer signs each quote to attest the share-to-underlying exchange rate.

When the batch is later fulfilled and the manager calls **claimRedeem**, assets are distributed pro-rata by shares across the batch:

uint256 assetsToSend = (_shares * remainingAssets) / remainingShares;

This pro-rata distribution is only correct if every request in the batch was priced at the same share-to-underlying rate. If the signer issues quotes at different rates within a single batch, the pro-rata split will redistribute value between users: those who were quoted a lower rate receive more than expected, while those quoted a higher rate receive less.

Steps to Reproduce

1. At **t=100**, Alice submits an unwind with a signed quote of **1 share = 2 underlying**. The adapter calls **createWriteOffRequest(100 shares, 200 underlyings)**. Batch state: **remainingBatchShares = 100**.

2. At **t=200**, Alice's quote has expired but the batch has not been fulfilled yet. The signer issues a new quote for Bob at a different rate: **1 share = 3 underlying**.

3. Bob submits his unwind. The adapter calls **createWriteOffRequest(100 shares, 300 underlyings)**. The escrow accumulates this on top of Alice's request. Batch state: **remainingBatchShares = 200**.

4. The borrower fulfills the combined request, delivering **500 underlyings** (200 + 300) to the adapter.

5. During settlement via **claimRedeem**, the 500 underlyings are split pro-rata by shares:

- Alice (100 shares): receives **100 * 500 / 200 = 250 underlyings**, **more** than her quoted 200.

- Bob (100 shares): receives **100 * 500 / 200 = 250 underlyings**, **less** than his quoted 300.

Alice benefits at Bob's expense because the pro-rata distribution averages the prices, ignoring the per-request rate each user was quoted.

Recommendation

Either:

1. **Enforce on-chain**: Record the batch price (underlyings per share) on the first **requestRedeem** call within a batch and revert subsequent requests that specify a different rate.

2. **Document as a trust assumption**: If price consistency is intended to be enforced off-chain (i.e. the signer must never change the rate within an active batch), explicitly document this as a trust assumption on the signer role.

3S-M03

Pause guard on admin-only functions blocks emergency response during protocol pause

Id	3S--M03
Classification	Medium
Impact	High
Likelihood	Low
Category	Bug
Status	Addressed in #73027ec.

Description

Several privileged administrative functions in **UnwindManager**, **UnwindManagerExtension**, and **WindManager** include a **_requireNotPaused()** check alongside the **_checkRole(DEFAULT_ADMIN_ROLE)** gate. The affected functions are **UnwindManager::setExtensions**, **UnwindManagerExtension::upgradeUnwindEscrow**, **UnwindManagerExtension::upgradeUnwindLiquidator**, **WindManager::upgradeWindEscrow**, and **WindManager::setPolicyId**.

These functions are configuration and upgrade entry points that the admin is most likely to need precisely when the protocol is paused, for example to deploy a patched escrow or liquidator implementation, reroute selectors to a fixed extension, or update a stale policy id. Because **_requireNotPaused()** runs before the role check, the admin must first unpause the protocol, execute the fix, and then re-pause. This creates a window during which user-facing operations (requesting unwinds, settling, liquidating) are unblocked, exposing the protocol to the very risk that triggered the pause.

The **DEFAULT_ADMIN_ROLE** is the highest-privilege role in the system. If this role were compromised, the attacker could already unpause at will, so the pause guard adds no meaningful defense against a compromised admin. It only hinders legitimate emergency workflows.

Recommendation

Remove the **_requireNotPaused()** call from the five affected functions. The **_checkRole(DEFAULT_ADMIN_ROLE)** gate is sufficient access control, and removing the pause guard allows the admin to perform emergency upgrades and configuration changes without exposing the protocol to an unsafe unpaused window.

3S-M04

Blacklisted **requester** blocks liquidation in **liquidateUnwind**

Id	3S--M04
Classification	Medium
Impact	High
Likelihood	Low
Category	Bug
Status	Addressed in #57b5a61.

Description

UnwindManagerExtension::liquidateUnwind is a privileged function callable by accounts with **LIQUIDATION_OPERATOR_ROLE**. Its purpose is to liquidate unhealthy unwind escrow positions whose debt value exceeds collateral value. After executing the liquidation, settling debts, and extracting a liquidation bonus, the function transfers any remaining base assets (**requesterPayout**) back to the original **requester**, which is the **ownerOf** the receipt NFT:

```
cache.requesterPayout = cache.availableBaseAssets - cache.bonusAssets;
if (cache.requesterPayout > 0) {
    SafeERC20.safeTransfer(cache.debtAsset, requester, cache.requesterPayout);
}
```

The **debtAsset** in this protocol is USDC, which implements an address blacklist. If the **requester** address has been added to USDC's blacklist, the **safeTransfer** call reverts, causing the entire **liquidateUnwind** transaction to revert. Since this is the only code path for liquidating unhealthy unwind positions, a blacklisted **requester** effectively makes their position unliquidatable. The position continues to accrue bad debt with no mechanism for the protocol to resolve it.

Recommendation

Use **SafeERC20.trySafeTransfer** (already available in OpenZeppelin 5.4.0) instead of **safeTransfer**. If the transfer is not successful, store the payout amount alongside the current **requester** address in a mapping, allowing the recorded recipient to pull the funds later via a dedicated **claimPayout** function.

```
+ struct UnclaimedPayout {
+     address recipient;
+     uint256 amount;
+ }
+ mapping(bytes32 => UnclaimedPayout) public s_unclaimedPayouts;
  function liquidateUnwind(
      bytes32 _requestId,
      uint256 _repayAssets,
      uint256 _minYieldBalance,
      bytes[] memory _pythUpdateData
  )
      external
      payable
  {
      // ... existing logic ...
      cache.requesterPayout = cache.availableBaseAssets - cache.bonusAssets;
      if (cache.requesterPayout > 0) {
-         SafeERC20.safeTransfer(cache.debtAsset, requester, cache.requesterPayout);
+         bool success = SafeERC20.trySafeTransfer(cache.debtAsset, requester,
cache.requesterPayout);
+         if (!success) {
+             s_unclaimedPayouts[_requestId] = UnclaimedPayout(requester,
cache.requesterPayout);
+         }
      }
      // ... rest of function ...
  }
+ function claimPayout(bytes32 _requestId) external {
+     UnclaimedPayout memory payout = s_unclaimedPayouts[_requestId];
+     if (payout.amount == 0) revert
UnwindManagerExtension__NoPayout(_requestId);
+     if (msg.sender != payout.recipient) revert
UnwindManagerExtension__UnauthorizedClaim(msg.sender);
+     delete s_unclaimedPayouts[_requestId];
+     IERC20 debtAsset =
IERC20(IEVault(IUnwindEscrow(s_requestIdToEscrow[_requestId]).getDebtVault()).as
set());
+     SafeERC20.safeTransfer(debtAsset, payout.recipient, payout.amount);
+ }
```

3S-M05

Dust repayment can grief **UnwindManagerExtension::liquidateUnwind** by reducing **maxRepay** below **_repayAssets**

Id	3S--M05
Classification	Medium
Impact	High
Likelihood	Low
Category	Bug
Status	Addressed in #8ddf72f.

Description

The **UnwindManagerExtension::liquidateUnwind** function allows liquidation operators to liquidate unhealthy unwind escrow positions. During execution, the function queries the Euler vault via **IEVault(debtVault).checkLiquidation** to determine the maximum repayable amount (**maxRepay**) and then validates the caller-supplied **_repayAssets** against it:

```
        (cache.maxRepay,) = IEVault(debtVault)
            .checkLiquidation(address(liquidator), address(unwindEscrow),
address(s_pendingSettlementBaseAssetVault));
        if (cache.maxRepay == 0 || _repayAssets > cache.maxRepay) {
            revert
UnwindManagerExtension__UnwindRequestNotLiquidatable(_requestId);
        }
```

The check enforces a strict upper bound: if **_repayAssets** exceeds **cache.maxRepay**, the entire transaction reverts. This becomes problematic when the liquidation operator intends to repay the full liquidatable amount by setting **_repayAssets** equal to the expected **cache.maxRepay**. An adversary can frontrun the liquidation transaction by repaying a dust amount of debt (e.g., 1 wei) directly to the **unwindEscrow** via the Euler vault. This reduces the escrow's outstanding debt, which in turn lowers the **maxRepay** value returned by **checkLiquidation**. The operator's **_repayAssets** now exceeds the updated **maxRepay**, causing the transaction to revert.

Since this attack is cheap (only a dust amount of the debt asset) and repeatable, an attacker can persistently block liquidations of unhealthy positions. This delays or prevents

the protocol from settling bad debt, potentially leading to further losses as the position continues to deteriorate.

Steps to Reproduce

1. An unwind escrow position becomes unhealthy (debt value exceeds collateral value).

2. A liquidation operator submits a **liquidateUnwind** transaction with **_repayAssets** set to the full liquidatable amount observed off-chain.

3. An attacker monitors the mempool and frontruns the transaction by calling **IEVault(debtVault).repay** on behalf of the **unwindEscrow** with a 1 wei amount.

4. The escrow's outstanding debt decreases, reducing the **maxRepay** returned by **checkLiquidation**.

5. The operator's **_repayAssets** now exceeds **cache.maxRepay**, and the liquidation transaction reverts with **UnwindManagerExtension__UnwindRequestNotLiquidatable**.

6. The attacker can repeat this at negligible cost to continuously block liquidation.

Recommendation

Cap **_repayAssets** at **cache.maxRepay** instead of reverting when the caller-supplied value exceeds the maximum. This makes the liquidation resilient to dust repayments without changing the intended behavior when **_repayAssets** is within bounds.

```
        (cache.maxRepay,) = IEVault(debtVault)
            .checkLiquidation(address(liquidator), address(unwindEscrow),
address(s_pendingSettlementBaseAssetVault));
-       if (cache.maxRepay == 0 || _repayAssets > cache.maxRepay) {
+        if (cache.maxRepay == 0) {
            revert
UnwindManagerExtension__UnwindRequestNotLiquidatable(_requestId);
        }
+       if (_repayAssets > cache.maxRepay) {
+           _repayAssets = cache.maxRepay;
+       }
```

3S-M06

Pending CDP collateral is minted from pre-fee notional, inflating health at request time

Id	3S--M06
Classification	Medium
Impact	High
Likelihood	Low
Category	Bug
Status	Addressed in #f4e510f.

Description

WindManager::requestWind allows a user to open a leveraged position on a tokenized RWA asset. The function computes **notionalAssets** (equity times leverage), deducts a Keyring fee to arrive at **totalAssets**, then creates a **WindEscrow** that holds a temporary CDP (pending-settlement base asset collateral against borrowed debt) until the RWA deposit settles and the position is finalized.

The pending-settlement base asset (PSBA) collateral minted into the escrow is derived from **notionalAssets** (the pre-fee amount), but only **totalAssets** (the post-fee amount) is forwarded to the adapter for the actual RWA deposit. When **WindEscrow::finalizeWind** swaps the PSBA collateral for the received RWA tokens, the collateral backing the debt shrinks by exactly **feeAmount**, producing a worse LTV than what Euler validated at borrow time. Depending on how tight the liquidation LTV margin is configured for the RWA collateral vault, this collateral reduction can push the position past the liquidation threshold, making it instantly unhealthy upon finalization.

Steps to reproduce:

1. A Keyring fee of 10% (**s_keyringFeeInBps = 1000**) is configured.

2. A user calls **WindManager::requestWind** with **_equityAmount = 50** and **_leverageBps = 20000** (2x leverage).

3. The function computes:

   - **notionalAssets = (50 * 20000) / 10000 = 100**

   - **borrowAmount = 100 - 50 = 50**

   - **feeAmount = (100 * 1000) / 10000 = 10**

- **totalAssets = 100 - 10 = 90**

4. 100 PSBA tokens are minted to the escrow. The escrow deposits 100 PSBA as collateral and borrows 50 debt. Euler sees an LTV of **50 / 100 = 0.50** and the health check passes.

5. Only 90 of base asset is sent to the adapter via **requestDeposit**. The remaining 10 is transferred to the fee collector.

6. When the RWA deposit settles, **finalizeWind** replaces the 100 PSBA collateral with RWA tokens worth ~90.

7. The effective LTV jumps to **50 / 90 ≈ 0.556**. If this exceeds the liquidation LTV configured for the RWA collateral vault, the position becomes immediately liquidatable upon finalization.

Recommendation

Mint PSBA collateral equal to **totalAssets** (post-fee) instead of **notionalAssets**, so that the pending CDP reflects the same collateral value the finalized position will hold.

```
-       cache.pendingSettlementAmount = cache.notionalAssets;
+       cache.pendingSettlementAmount = cache.totalAssets;
        i_pendingSettlementBaseAsset.mint(cache.escrow,
cache.pendingSettlementAmount);
```

```
IWindEscrow(cache.escrow).openPendingCdpAndBorrow(cache.pendingSettlement
Amount, cache.borrowAmount);
```

This ensures the Euler health check at borrow time evaluates against the same collateral value that will back the position after finalization.

3S-M07

Missing sub-account ownership validation in **requestWind** permanently locks user funds

Id	3S--M07
Classification	Medium
Impact	High
Likelihood	Low
Category	Bug
Status	Acknowledged with note: *“Expected. The user can request their finalized leveraged position to be sent to any account of their liking, though this is probably a rare case.”*

Description

WindManager::requestWind accepts a **_subAccount** parameter that identifies the Euler sub-account into which the leveraged position will ultimately be finalized. However, the function never verifies that the caller actually owns the provided sub-account. A user who supplies an incorrect or unowned **_subAccount** (whether by mistake or intentionally) will pass all existing validation checks and have their request enqueued and their equity transferred.

The problem surfaces during finalization. **WindEscrow::_transferCdpToSubAccount** executes an EVC batch that operates **onBehalfOfAccount: s_subAccount** — enabling collateral, enabling the debt controller, and pulling debt. For this batch to succeed, the escrow must be an authorized operator of the target sub-account. The NatSpec on **requestWind** documents this requirement: *"This subaccount must set the wind escrow as operator, otherwise it won't be able to receive the CDP."* If the user provided a sub-account they do not own, they cannot grant operator permissions to the escrow, and every call to **finalizeWind** will revert.

At this point the request has already been settled and removed from **s_pendingRequestQueue** by **settleQueue**, and the user's equity has been forwarded to the adapter. With the cancellation flow (**triggerCancellation**, **claimCanceledAssets**, **refundRequests**) removed from the current codebase and no admin rescue or emergency withdrawal mechanism present, there is no path to recover the locked funds. The user's equity is permanently lost.

The analogous function in the unwind flow, **UnwindManager::requestUnwind**, already validates sub-account ownership via **_validateSubAccount** before proceeding.

Recommendation

Validate that **_msgSender()** owns **_subAccount** at the start of **WindManager::requestWind**, mirroring the check used in **UnwindManager::_validateSubAccount**.

```
 function requestWind(
     uint256 _equityAmount,
     uint256 _leverageBps,
     address _subAccount,
     bytes[] memory _pythUpdateData,
     bytes memory _adapterData
 )
     external
     payable
     returns (bytes32)
 {
     _requireNotPaused();
     _requireToSAccepted(_msgSender());
     _requireKyced(_msgSender());
+    _validateSubAccount(_msgSender(), _subAccount);
```

3S-M08

EVC-aware **_msgSender()** in role checks allows operators to execute privileged admin functions

Id	3S--M08
Classification	Medium
Impact	High
Likelihood	Low
Category	Bug
Status	Addressed in #0d9c000.

Description

The **WindManager** contract gates its administrative functions with OpenZeppelin's **_checkRole**, which internally resolves the caller via **_msgSender()**. Because **WindManager** overrides **_msgSender()** with EVC context resolution (**EVCUtil._msgSender()**), the role check does not verify the direct caller (**msg.sender**) but instead the account on whose behalf the EVC is operating. This means that any EVC operator authorized by an admin address can invoke privileged governance functions such as **WindManager::upgradeWindEscrow**, **WindManager::setKeyringFeeRecipient**, **WindManager::setPolicyId**, and all other **DEFAULT_ADMIN_ROLE**-gated setters.

The contract already acknowledges this class of risk in **WindManager::acceptTos**, which deliberately passes **msg.sender** to **_acceptTos** and documents the rationale: **/// @dev Uses msg.sender to avoid EVC operator acceptance on behalf of users.** However, this same defensive pattern is not applied to any of the role-gated administrative functions.

The affected functions using **_checkRole(DEFAULT_ADMIN_ROLE)** are: **setTermsOfService**, **setKeyringFeeInBps**, **setKeyringFeeRecipient**, **upgradeWindEscrow**, **setPolicyId**, **setMinEquityAmount**, **setMinLeverageBps**, and **setMaxLeverageBps**. The **SAFETY_MANAGER_ROLE**-gated function **setPaused** and the **KEYRING_OPERATOR_ROLE**-gated function **settleQueue** are similarly affected. The exception is **finalizeWind**, where EVC context resolution is intentional: the conditional check allows the original requester (resolved through EVC) to finalize their own request without holding the operator role.

If the **DEFAULT_ADMIN_ROLE** address also holds a position in the Euler cluster and has authorized keeper bots as EVC operators, those bots could call through the EVC to execute governance functions on behalf of the admin, bypassing the intended access control boundary.

The unwind module is affected by the same issue.

Recommendation

Replace **_checkRole** calls with a helper that validates **msg.sender** directly, for all privileged role checks except the conditional operator check in **finalizeWind** where EVC resolution is by design.

```
+ function _checkRoleMsgSender(bytes32 role) internal view {
+     if (!hasRole(role, msg.sender)) {
+         revert AccessControlUnauthorizedAccount(msg.sender, role);
+     }
+ }
```

Then replace each **_checkRole(DEFAULT_ADMIN_ROLE)**, **_checkRole(SAFETY_MANAGER_ROLE)**, and the unconditional **_checkRole(KEYRING_OPERATOR_ROLE)** in **settleQueue** with **_checkRoleMsgSender(...)**. Leave the conditional **_checkRole(KEYRING_OPERATOR_ROLE)** in **finalizeWind** unchanged, as EVC context resolution is required there for the requester bypass logic.

Apply the same fix to the unwind module functions such as **upgradeUnwindEscrow**, **upgradeUnwindLiquidator**, **setExtensions**, and **grantRole**.

3S-L01

LTV check against RWA vault instead of PSBA vault

Id	3S--L01
Classification	Low
Impact	Low
Likelihood	Low
Category	Bug
Status	Acknowledged with note: *"The LTV check must always occur on the RWA collateral vault because it is the place where the finalized leveraged CDP ends up. If we check LTV against the PSBA vault and the RWA collateral vault is not configured in alignment with the PSBA vault, then that could be a bigger problem."*

Description

UnwindManager::_checkLtv validates that the requester's position is healthy enough to proceed with an unwind by comparing the computed LTV against the **liquidationLtv** threshold. It fetches this threshold from the RWA collateral vault (**s_rwaCollateralVault**) via the vault lens.

However, during the same transaction, **UnwindEscrow::prepareRedemption** replaces the RWA collateral with PSBA (pending settlement base asset) collateral within an EVC batch. It disables the RWA vault as collateral and enables the PSBA vault instead. At the end of the batch, when the EVC performs its deferred account status check, it evaluates the escrow's health using the PSBA vault's **liquidationLtv**, not the RWA vault's.

If the PSBA vault has a lower **liquidationLtv** than the RWA vault, a position that passes **_checkLtv** will nonetheless fail the EVC's account status check. The unwind request reverts even though the contract's own validation approved it. Users whose positions sit between the two thresholds are unable to unwind.

The root cause is that **_checkLtv** reads **liquidationLtv** from **s_rwaCollateralVault** while the post-swap collateral is held in **s_pendingSettlementBaseAssetVault**. The two vaults can have independently configured LTV parameters that diverge over time, even if they match at deployment.

Recommendation

_checkLtv should fetch the **liquidationLtv** from the PSBA vault (**s_pendingSettlementBaseAssetVault**) rather than the RWA vault, since that is the collateral vault the EVC will evaluate at the end of the batch.

```
    function _checkLtv(uint256 _collateralShares, uint256 _debt) internal view {
        IVaultLens.LTVInfo[] memory collateralLtvInfo =

s_vaultLens.getRecognizedCollateralsLTVInfo(address(s_baseAssetDebtVault));
        uint16 liquidationLtv = _getTargetCollateralVaultLtvInfo(
-           collateralLtvInfo, address(s_rwaCollateralVault),
address(s_baseAssetDebtVault)
+           collateralLtvInfo, address(s_pendingSettlementBaseAssetVault),
address(s_baseAssetDebtVault)
        );
```

Additionally, consider adding a deployment-time or administrative invariant check that requires both vaults to share the same **liquidationLtv**, so that any future governance-driven LTV change on one vault does not silently introduce a mismatch.

3S-L02

Fee deduction gap between **finalizeUnwind** and **liquidateUnwind** leads to stuck positions

Id	3S--L02
Classification	Low
Impact	Medium
Likelihood	Low
Category	Bug
Status	Acknowledged with note: *“When a position on the escrow is liquidated, the user's request is implicitly unsuccessful. Thus, applying a fee would be unfair.* *As for there not being enough stablecoins to repay the debt, we ensure that there would be enough collateral to back up the remaining debt by means of the buffer LTV percentage (in bps). This is part of a soft guarantee from risk management/modelling.”*

Description

UnwindManagerExtension::finalizeUnwind settles a completed unwind by deducting protocol and provider fees from **claimableAssets**, then repaying the escrow's outstanding debt with the remainder. **UnwindManagerExtension::liquidateUnwind** provides an alternative path for unhealthy positions, but requires **debtValue > collateralValue** before it can execute.

A gap exists between these two code paths. In **finalizeUnwind**, fees are computed on the full **claimableAssets** amount (line 252) and subtracted to produce **availableAfterFees** (line 261). This reduced amount is then used to repay debt via the escrow's **finalizeUnwind** call (line 290), which transfers the CDP back to the user's sub-account. If **availableAfterFees** is insufficient to bring the position back to health, Euler's account status check will revert the entire transaction because the restored CDP violates the vault's LTV constraints.

Meanwhile, **liquidateUnwind** evaluates position health using raw collateral and debt values without any fee deduction (lines 346-353). If the pre-fee position is healthy (**debtValue <= collateralValue**), the call reverts with **UnwindRequestNotLiquidatable**.

This creates a deadlock when accrued interest during the settlement period pushes the position close to its debt ceiling. The fees applied by **finalizeUnwind** make the position underwater from Euler's perspective, causing it to revert. At the same time, the position remains technically healthy before fees, so **liquidateUnwind** also reverts. Neither code path can execute.

The only resolution is to wait for the position to deteriorate further until **debtValue** naturally exceeds **collateralValue**, at which point **liquidateUnwind** becomes callable. This delay harms the user (who loses additional value to accruing interest) and the protocol (which forfeits its fee revenue entirely, since **liquidateUnwind** does not charge fees).

Recommendation

The cleanest fix would be to apply the recommendation suggested in issue #82 **liquidateUnwind does not deduct or distribute accrued keyring and provider fees**. Alternatively, cap the total fee in **finalizeUnwind** so that it never exceeds the surplus of **claimableAssets** over **debtOutstanding**.

3S-L03

Pause guard on **liquidateUnwind** blocks loss mitigation during protocol emergencies

Id	3S--L03
Classification	Low
Impact	Medium
Likelihood	Low
Category	Bug
Status	Addressed in #290dfc0.

Description

UnwindManagerExtension::liquidateUnwind allows liquidation operators to settle unhealthy unwind requests by repaying debt and seizing collateral. The function is restricted to addresses holding the **LIQUIDATION_OPERATOR_ROLE**, ensuring only trusted operators can trigger liquidations.

However, the function also enforces **_requireNotPaused()** at line 316, which prevents any liquidation from proceeding while the protocol is paused. In practice, a protocol pause is most likely triggered during an emergency, precisely when positions are at elevated risk of becoming undercollateralized. Blocking liquidations in this state means unhealthy positions continue to accrue bad debt while no operator can intervene, potentially deepening losses for the protocol and its users.

Because **liquidateUnwind** is already gated behind **LIQUIDATION_OPERATOR_ROLE**, the pause guard provides no meaningful additional protection. The role check alone is sufficient to prevent unauthorized calls.

Recommendation

Remove the **_requireNotPaused()** check from **UnwindManagerExtension::liquidateUnwind** so that liquidation operators can continue settling unhealthy positions even while the protocol is paused.

```
function liquidateUnwind(
    bytes32 _requestId,
    uint256 _repayAssets,
    uint256 _minYieldBalance,
```

```
    bytes[] memory _pythUpdateData
 )
    external
    payable
 {
-    _requireNotPaused();
    if (!hasRole(LIQUIDATION_OPERATOR_ROLE, _msgSender())) {
       revert
UnwindManagerExtension__UnauthorizedLiquidationCaller(_msgSender());
    }
```

3S-L04

Redundant debt asset transfer in **liquidateUnwind** inflates token requirements and may block liquidation

Id	3S--L04
Classification	Low
Impact	Medium
Likelihood	Low
Category	Bug
Status	Addressed in #5248c88.

Description

UnwindManagerExtension::liquidateUnwind is responsible for liquidating an underwater unwind escrow position. It deploys a fresh **UnwindLiquidator** proxy, executes an Euler vault liquidation that moves debt from the escrow to the liquidator, settles the debts on both the liquidator and escrow, and distributes any remaining assets.

During this process, the function performs two separate **safeTransfer** calls of debt assets to the liquidator:

```
cache.debtAsset = IERC20(IEVault(debtVault).asset());
SafeERC20.safeTransfer(cache.debtAsset, address(liquidator), _repayAssets);
liquidator.executeLiquidation(_repayAssets, _minYieldBalance);
```

```
cache.liquidatorDebt = IEVault(debtVault).debtOf(address(liquidator));
cache.escrowDebt = IEVault(debtVault).debtOf(address(unwindEscrow));
if (cache.liquidatorDebt > 0) {
    SafeERC20.safeTransfer(cache.debtAsset, address(liquidator),
cache.liquidatorDebt);
}
```

The first transfer sends **_repayAssets** to the liquidator before calling **executeLiquidation**. However, **UnwindLiquidator::executeLiquidation** only calls **s_debtVault.liquidate(...)**, which is an Euler vault accounting operation that transfers debt from the escrow to the liquidator and seizes collateral shares. This operation does not consume any debt tokens

from the liquidator's balance; the Euler vault defers the health check to the end of the batch call.

The second transfer sends **cache.liquidatorDebt** (the full post-liquidation debt of the liquidator) and is the transfer that **settleDebtsAndTransfer** actually uses to call **s_debtVault.repay(...)**. The tokens from the first transfer remain idle in the liquidator and are returned to the manager at the end of **settleDebtsAndTransfer** as part of the remaining balance sweep.

While this does not result in a permanent loss of funds, it requires the **UnwindManager** to hold **_repayAssets + cache.liquidatorDebt + cache.escrowDebt** worth of debt tokens instead of just **cache.liquidatorDebt + cache.escrowDebt**. These tokens come from settled redemption requests via **_settleQueueIfClaimable**. When the manager's available balance is sufficient for the true obligations but not for the inflated total, the liquidation reverts, preventing timely liquidation of unhealthy positions.

Steps to Reproduce

1. An unwind escrow becomes underwater (debt value exceeds collateral value).

2. A liquidation operator calls **liquidateUnwind** with a **_repayAssets** value.

3. The **UnwindManager** holds enough settled debt assets to cover **cache.liquidatorDebt + cache.escrowDebt** but less than **_repayAssets + cache.liquidatorDebt + cache.escrowDebt**.

4. The first **safeTransfer** of **_repayAssets** to the liquidator succeeds, but the second transfer of **cache.liquidatorDebt** reverts due to insufficient balance in the manager, or vice versa. The liquidation is blocked despite sufficient funds for the actual debt obligations.

5. This is more likely when the request being liquidated is the only (or primary) unwind request in the adapter's redemption batch, leaving the manager with limited settled funds.

Recommendation

Remove the first, unnecessary transfer of **_repayAssets** to the liquidator. Only the post-liquidation debt transfers are needed.

```
        cache.debtAsset = IERC20(IEVault(debtVault).asset());
-       SafeERC20.safeTransfer(cache.debtAsset, address(liquidator), _repayAssets);
        liquidator.executeLiquidation(_repayAssets, _minYieldBalance);
```

3S-N01

Redundant liquidation-status checks in settlement loop

Id	3S--N01
Classification	None
Category	Suggestion
Status	Addressed in #8c31b24.

Description

UnwindManagerExtension::_settleQueueIfClaimable processes the pending request queue in two phases. First, a preliminary **while** loop skips past any front-of-queue entries whose escrow status is **Liquidated**. Second, the main allocation loop iterates over remaining entries, again checking for **Liquidated** status and popping those entries with a **continue**.

Both checks are dead code. A request's escrow position can only be liquidated after the request has been settled and its collateral is live on Euler. While a request sits in **s_pendingRequestQueue**, its escrow has not yet settled, so the position cannot be subject to liquidation. The **Liquidated** status is therefore unreachable for any request that is still enqueued.

Recommendation

Remove both dead code blocks from **UnwindManagerExtension::_settleQueueIfClaimable**. The preliminary cleanup loop should be deleted entirely. The **Liquidated** guard inside the main allocation loop should also be removed.

```
  function _settleQueueIfClaimable(uint256 _maxShares, bytes calldata _adapterData)
internal {
    if (!QueueLib.hasItems(s_pendingRequestQueue)) {
        return;
    }
-   while (QueueLib.hasItems(s_pendingRequestQueue)) {
-       bytes32 requestId = QueueLib.peekFront(s_pendingRequestQueue);
-       IUnwindEscrow.EscrowState memory requestState =
-           IUnwindEscrow(s_requestIdToEscrow[requestId]).getEscrowState();
-       if (requestState.status != IUnwindEscrow.Status.Liquidated) {
-           break;
```

```
-       }
-       QueueLib.popFront(s_pendingRequestQueue);
-   }
-
    uint256 claimableShares = s_adapter.claimableRedeemRequest(_adapterData);



    while (sharesRemaining > 0 && QueueLib.hasItems(s_pendingRequestQueue)) {
        bytes32 requestId = QueueLib.peekFront(s_pendingRequestQueue);
        IUnwindEscrow.EscrowState memory requestState =
            IUnwindEscrow(s_requestIdToEscrow[requestId]).getEscrowState();
-       if (requestState.status == IUnwindEscrow.Status.Liquidated) {
-           QueueLib.popFront(s_pendingRequestQueue);
-           continue;
-       }
        uint256 totalSharesForRequest = requestState.collateralAssetsToRedeem;
```

3S-N02

Stale snapshotted **providerFeeInBps** combined with live **feeRecipient** lookup can permanently block finalization

Id	3S--N02
Classification	None
Category	Suggestion
Status	Addressed in #9ab65fc.

Description

UnwindManagerExtension::finalizeUnwind is responsible for settling a completed unwind request: it distributes fees, repays the escrow's outstanding debt, transfers any surplus to the NFT owner, and restores the CDP to the user's sub-account.

During finalization, the function reads **providerFeeInBps** from the escrow state, which was snapshotted at request creation time in **UnwindManager::requestUnwind**. However, the **providerFeeRecipient** is not snapshotted alongside it. Instead, it is queried live from the current adapter via **s_adapter.feeRecipient()**. This creates a temporal mismatch between two coupled values: the fee amount is determined by state captured in the past, while the fee destination is resolved from the present.

If the adapter or its configuration changes between request creation and finalization (e.g., the provider fee is disabled and the fee recipient is set to **address(0)**, or the adapter is replaced entirely), the snapshotted **providerFeeInBps** will still produce a non-zero **providerFees** value, but the live **feeRecipient** call will return **address(0)**. The function then reverts with **UnwindManagerExtension__ProviderFeeRecipientZero**, permanently bricking finalization for any in-flight request that was created with a non-zero provider fee.

Note: under the current **ParetoAdapter**, **onRequestUnwind** returns a **feeInBps** of **0** and **feeRecipient()** returns **address(0)**, so this path is not currently reachable. The issue would surface if a future adapter returns a non-zero provider fee.

Recommendation

Snapshot the **providerFeeRecipient** in the **UnwindEscrow** at initialization time, alongside **providerFeeInBps**, so that both values are consistent for the lifetime of the request.

```
// In UnwindEscrow storage
uint16 internal s_providerFeeInBps;
```

```
+ address internal s_providerFeeRecipient;
  // In UnwindEscrow.initialize
  s_providerFeeInBps = _params.providerFeeInBps;
+ s_providerFeeRecipient = _params.providerFeeRecipient;
  // In EscrowState struct and getEscrowState()
  struct EscrowState {
      bytes32 requestId;
      uint256 collateralAssetsToRedeem;
      uint16 providerFeeInBps;
+     address providerFeeRecipient;
      uint16 keyringFeeInBps;
      uint256 receiptNftId;
      Status status;
  }
  // In UnwindManagerExtension::finalizeUnwind, replace the live lookup:
- cache.providerFeeRecipient = s_adapter.feeRecipient();
+ cache.providerFeeRecipient = vaultState.providerFeeRecipient;
```

3S-N03

Unused **totalFeeInBps** computation in **UnwindManager::requestUnwind**

Id	3S--N03
Classification	None
Category	Suggestion
Status	Addressed in #43693dd.

Description

UnwindManager::requestUnwind computes **cache.totalFeeInBps** as the sum of **cache.providerFeeInBps** and **cache.keyringFeeInBps**, but this value is never read afterward. It is not passed to the escrow via **initArgs**, not included in the **UnwindRequested** event, and not used in any validation. In contrast, **UnwindManagerExtension** computes the same sum during finalization and actively uses it to derive fee amounts. The unused assignment suggests either dead code or a missing validation (such as a total fee cap) that was intended but not implemented.

Recommendation

If no total fee validation or emission is intended, remove the field and its assignment to eliminate dead code and save gas.

```
 struct RequestCache {
     uint256 collateralAssets;
     uint256 collateralAssetsToRedeem;
-    uint16 totalFeeInBps;
     uint256 pendingSettlementAmount;
     uint16 providerFeeInBps;
     uint16 keyringFeeInBps;
     address providerFeeRecipient;
     IUnwindEscrow unwindEscrow;
     bytes32 requestId;
     uint256 receiptNftId;
 }
```

```
  cache.keyringFeeInBps = s_keyringFeeInBps;
- cache.totalFeeInBps = cache.providerFeeInBps + cache.keyringFeeInBps;
  cache.collateralAssets =
s_rwaCollateralVault.convertToAssets(_params.collateralShares);
```

If a total fee cap was intended, add the appropriate validation before proceeding with escrow deployment.

3S-N04

Rounding dust in fee splitting leaves dust permanently stuck in **UnwindManagerExtension**

Id	3S--N04
Classification	None
Category	Suggestion
Status	Addressed in #5c15817.

Description

UnwindManagerExtension::finalizeUnwind settles a completed redemption by distributing the claimable debt assets across three destinations: Keyring fees, provider fees, and the remainder used to repay debt and/or refund the NFT owner. The fee amounts are computed by first calculating **totalFees** from the combined basis points, then splitting **totalFees** proportionally between **keyringFees** and **providerFees** using integer division. Because of Solidity's truncating division, **keyringFees + providerFees** can be strictly less than **totalFees**. However, **availableAfterFees** is computed as **claimableAssets - totalFees**, meaning the contract accounts for the full **totalFees** being removed, but only actually transfers out **keyringFees + providerFees**. The rounding remainder is never transferred to any recipient and remains trapped in the contract with no recovery mechanism.

```
        cache.totalFeeInBps = vaultState.keyringFeeInBps +
vaultState.providerFeeInBps;
        cache.totalFees = cache.totalFeeInBps == 0 ? 0 : (cache.claimableAssets *
cache.totalFeeInBps) / BPS;
        if (cache.totalFees > 0) {
            cache.keyringFees = (cache.totalFees * vaultState.keyringFeeInBps)
                / (vaultState.keyringFeeInBps + vaultState.providerFeeInBps);
            cache.providerFees = (cache.totalFees * vaultState.providerFeeInBps)
                / (vaultState.keyringFeeInBps + vaultState.providerFeeInBps);
        }
        cache.providerFeeRecipient = s_adapter.feeRecipient();
        cache.availableAfterFees = cache.claimableAssets - cache.totalFees;
```

This issue does not manifest under the current adapter because the active adapter sets **providerFeeInBps** to zero, which eliminates the splitting step entirely. If a future adapter introduces a non-zero provider fee, the dust will accumulate with every finalization.

Steps to Reproduce

1. Configure an adapter where both **keyringFeeInBps** and **providerFeeInBps** are non-zero (e.g., **keyringFeeInBps = 300** and **providerFeeInBps = 700**).

2. Initiate an unwind with **claimableAssets = 110**.

3. **totalFees** is computed as **(110 * 1000) / 10000 = 11**.

4. **keyringFees** is computed as **(11 * 300) / 1000 = 3**.

5. **providerFees** is computed as **(11 * 700) / 1000 = 7**.

6. **availableAfterFees** is computed as **110 - 11 = 99**.

7. Total tokens transferred out: **3 + 7 + 99 = 109**.

8. 1 wei token of the debt asset remains in the contract, permanently unrecoverable.

Recommendation

Compute **availableAfterFees** from the actual fee amounts transferred rather than from **totalFees**, so no rounding remainder is left behind:

```
- cache.availableAfterFees = cache.claimableAssets - cache.totalFees;
+ cache.availableAfterFees = cache.claimableAssets - cache.keyringFees - cache.providerFees;
```

3S-N05

Receipt NFT is not burned on finalization or liquidation of unwind requests

Id	3S--N05
Classification	None
Category	Suggestion
Status	Addressed in #9e7ed8a.

Description

UnwindManager inherits from ERC721 and mints a receipt NFT to the requester during **UnwindManager::requestUnwind**. This NFT serves as proof of ownership for a pending unwind request and is used to authorize callers in both **UnwindManagerExtension::finalizeUnwind** and **UnwindManagerExtension::liquidateUnwind** (via **ownerOf(vaultState.receiptNftId)**).

However, neither **finalizeUnwind** nor **liquidateUnwind** burns the receipt NFT after the unwind request has been fully settled. Once an unwind is finalized or liquidated, the NFT continues to exist and remains owned by the original requester (or any subsequent transferee), even though the underlying request is no longer active.

Re-entrancy into these functions using the same NFT is prevented by secondary guards:

- In **finalizeUnwind**, the function zeroes out **s_requestIdToClaimableAssets[_requestId]** and **s_requestIdToClaimableShares[_requestId]**, causing a subsequent call to revert with **UnwindManagerCore__UnwindRequestNotClaimable**.

- In **liquidateUnwind**, the escrow status is set to **Liquidated**, causing a subsequent call to revert with **UnwindManagerExtension__UnwindRequestAlreadyLiquidated**.

Because the NFT persists, any off-chain system or marketplace that relies on NFT ownership as a proxy for an active unwind position will return stale data unless it independently verifies the on-chain request status.

Recommendation

Burn the receipt NFT at the end of both **finalizeUnwind** and **liquidateUnwind** to ensure token ownership accurately reflects active request status.

// In finalizeUnwind, after all settlement logic:

```
  s_pendingSettlementBaseAsset.burn(address(this),
cache.pendingSettlementAssets);
+ _burn(vaultState.receiptNftId);
  _grantOrRevokeClusterAccess(
      collateralVault, debtVault, address(s_pendingSettlementBaseAssetVault),
address(unwindEscrow), false, true
  );


 // In liquidateUnwind, after all settlement logic:
+ _burn(state.receiptNftId);
  _grantOrRevokeClusterAccess(
      collateralVault, debtVault, address(s_pendingSettlementBaseAssetVault),
address(liquidator), false, false
  );
```

3S-N06

Incorrect natspec on beacon upgrade functions

Id	3S--N06
Classification	None
Category	Suggestion
Status	Addressed in #5431295.

Description

UnwindManagerExtension::upgradeUnwindEscrow and **UnwindManagerExtension::upgradeUnwindLiquidator** allow a **DEFAULT_ADMIN_ROLE** holder to change the implementation contract behind the escrow and liquidator beacons, respectively. Both functions call **upgradeTo** on their corresponding **UpgradeableBeacon** (**s_escrowBeacon** and **s_unwindLiquidatorBeacon**), which updates the single implementation address that every beacon proxy delegates to.

The **@dev** natspec on both functions states "affects new proxies only". This is incorrect. The beacon proxy pattern works by having all proxies query the beacon for the current implementation address at call time. Upgrading the beacon therefore changes the logic for every existing proxy, not only proxies deployed after the upgrade. An admin relying on this comment could upgrade a beacon under the false assumption that already-deployed escrow or liquidator proxies will continue running the previous implementation, when in reality all in-flight positions are immediately affected.

Recommendation

Correct the natspec on both functions to accurately reflect that a beacon upgrade applies to all existing and future proxies.

```
    /// @notice Upgrades the unwind escrow beacon implementation.
-   /// @dev Caller must have DEFAULT_ADMIN_ROLE; affects new proxies only.
+   /// @dev Caller must have DEFAULT_ADMIN_ROLE; affects all existing and future proxies.
    /// @param _newImplementation The new unwind escrow implementation.
    function upgradeUnwindEscrow(address _newImplementation) external {


    /// @notice Upgrades the unwind liquidator beacon implementation.
```

```
-   /// @dev Caller must have DEFAULT_ADMIN_ROLE; affects new proxies only.
+   /// @dev Caller must have DEFAULT_ADMIN_ROLE; affects all existing and future
proxies.
    /// @param _newImplementation The new unwind liquidator implementation.
    function upgradeUnwindLiquidator(address _newImplementation) external {
```

3S-N07

Missing **_disableInitializers** in beacon implementation contracts

Id	3S--N07
Classification	None
Category	Suggestion
Status	Addressed in #8d117d2.

Description

UnwindEscrow, **UnwindLiquidator**, and **WindEscrow** are upgradeable contracts deployed behind **UpgradeableBeacon** proxies. Each inherits OpenZeppelin's **Initializable** and exposes an **initialize** function gated by the **initializer** modifier. The beacon's **implementation** address points to a deployed instance of these contracts that is intended to serve only as a logic template; it should never hold meaningful state or be callable directly.

However, none of the three contracts call **_disableInitializers()** in their constructor. This means the implementation contract itself remains uninitialised after deployment. An attacker can call **initialize** directly on the implementation address, supplying arbitrary parameters.

While initialising the implementation does not affect existing proxies (each proxy maintains its own storage), an uninitialised implementation is a well-known footgun. If any off-chain component, integration, or future upgrade path references the implementation address directly rather than via the beacon proxy, it would interact with attacker-controlled state.

Recommendation

Add a constructor to each of the three contracts that calls **_disableInitializers()**, permanently locking the implementation against direct initialisation.

```
+ /// @custom:oz-upgrades-unsafe-allow constructor
+ constructor() {
+     _disableInitializers();
+ }
```

This should be applied to **UnwindEscrow**, **UnwindLiquidator**, and **WindEscrow**.

3S-N08

UnwindManager::requestUnwind uses **_mint** instead of **_safeMint** for receipt NFT

Id	3S--N08
Classification	None
Category	Suggestion
Status	Addressed in #0913729.

Description

UnwindManager::requestUnwind creates an unwind position and mints a receipt NFT to the caller via **_mint**. Unlike **_safeMint**, **_mint** does not check whether the recipient implements **IERC721Receiver**. If the caller is a contract that does not support ERC721 tokens, the receipt NFT will be permanently locked, making the associated unwind request unclaimable.

Recommendation

Replace **_mint** with **_safeMint** to ensure contract recipients support ERC721 token reception.

```
-       _mint(_msgSender(), cache.receiptNftId);
+       _safeMint(_msgSender(), cache.receiptNftId);
```

3S-N09

BatchCall__CallFailed error omits revert reason, hindering failure debugging

Id	3S--N09
Classification	None
Category	Suggestion
Status	Addressed in #c19e43f.

Description

BatchCall::batchCall executes a sequence of arbitrary low-level calls and, when **_revertOnFail** is true, reverts on the first failure. The revert uses the custom error **BatchCall__CallFailed(uint256, address)**, which includes the call index and the target address but discards the **result** bytes returned by the failed call. Those **result** bytes typically contain the downstream revert reason (a custom error selector, a revert string, or a panic code). Without forwarding them, callers and off-chain tooling receive no indication of *why* the sub-call failed, forcing developers to reproduce the failure independently to diagnose the root cause.

Recommendation

Include the **result** bytes in the error so that the downstream revert reason is surfaced to callers and transaction tracers.

```
- error BatchCall__CallFailed(uint256 _index, address _target);
+ error BatchCall__CallFailed(uint256 _index, address _target, bytes _reason);
```

```
- if (_revertOnFail && !success) revert BatchCall__CallFailed(i, callData.target);
+ if (_revertOnFail && !success) revert BatchCall__CallFailed(i, callData.target,
result);
```

3S-N10

Missing bounds check in **WindManager::getPendingRequestAt** returns zero for out-of-bounds index

Id	3S--N10
Classification	None
Category	Suggestion
Status	Addressed in #e26c683.

Description

WindManager::getPendingRequestAt is a view function that returns the pending wind request ID at a given queue index. It delegates directly to **QueueLib::peekAt**, which reads from the underlying mapping at **head + _index** without verifying that the index falls within the queue's bounds. Because Solidity mappings return the zero value for uninitialized slots, an out-of-bounds **_index** silently returns **bytes32(0)** instead of reverting.

The parallel function **UnwindManagerExtension::getPendingUnwindRequestIdAt** already includes the correct pattern: it calls **QueueLib::getCount** and reverts with **UnwindManagerExtension__PendingUnwindRequestIndexOutOfBounds** when **_index >= totalItems**. **WindManager::getPendingRequestAt** omits this check.

The practical impact is low. A frontend or backend integration that passes an invalid index will receive a zero-value request ID rather than a revert, which could lead to silent misinterpretation of queue state. No funds are at risk.

Recommendation

Add the same bounds check used in **UnwindManagerExtension::getPendingUnwindRequestIdAt** to **WindManager::getPendingRequestAt**.

```
 function getPendingRequestAt(uint256 _index) external view returns (bytes32) {
+    uint256 totalItems = QueueLib.getCount(s_pendingRequestQueue);
+    if (_index >= totalItems) {
+        revert WindManager__PendingRequestIndexOutOfBounds(_index, totalItems);
+    }
     return QueueLib.peekAt(s_pendingRequestQueue, _index);
 }
```

3S-N11

Uninitialized **s_minLeverageBps** allows users to open no-leverage positions

Id	3S--N11
Classification	None
Category	Suggestion
Status	Acknowledged with note: *“Expected. We follow the pattern to not initialize any admin-set vars in the constructor. It is expected that the admin will set these and other vars appropriately right after deployment, as is reflected in this deployment script”*

Description

WindManager::requestWind validates the caller-supplied leverage against **s_minLeverageBps** and **s_maxLeverageBps** to ensure positions fall within the admin-configured range. Both setters (**setMinLeverageBps**, **setMaxLeverageBps**) enforce that the value exceeds **BPS** (10 000), so any intentionally configured leverage bound is always greater than 1x. However, neither variable is initialized in the constructor; they both default to **0**.

The validation in **requestWind** compares with a simple less-than check:

```
if (_leverageBps < s_minLeverageBps) {
    revert WindManager__LeverageBelowMinimum(_leverageBps,
s_minLeverageBps);
}
if (_leverageBps > s_maxLeverageBps) {
    revert WindManager__LeverageTooHigh(_leverageBps, s_maxLeverageBps);
}
```

If the admin sets **s_maxLeverageBps** but neglects to set **s_minLeverageBps** (or vice versa), **s_minLeverageBps** remains **0** and the lower-bound check passes for any value, including leverage below **BPS**. A user could therefore submit a **_leverageBps** of, for example, **1**, resulting in a **notionalAssets** calculation of essentially zero and a position with no meaningful leverage. The same reasoning applies symmetrically: setting only the

minimum leaves **s_maxLeverageBps** at **0**, causing the upper-bound check to revert on every call since any valid leverage exceeds **0**.

While the practical impact is limited (the resulting position is economically pointless rather than exploitable), the contract silently permits a state the protocol does not intend to support.

Recommendation

Accept both bounds as constructor parameters so the contract is never deployed in an unconfigured state.

Alternatively, require both **s_minLeverageBps** and **s_maxLeverageBps** to be set before any position can be opened. Adding a guard at the top of **WindManager::requestWind** is the least invasive approach:

```
+ if (s_minLeverageBps == 0 || s_maxLeverageBps == 0) {
+     revert WindManager__LeverageRangeNotConfigured();
+ }
```

3S-N12

Operator can drain adapter batch assets without allocating tranche shares via repeated small **settleQueue** calls

Id	3S--N12
Classification	None
Category	Suggestion
Status	Addressed in #188e70b.

Description

WindManager::settleQueue is an operator-restricted function that claims processed deposit assets from the adapter and distributes the resulting tranche shares across the pending wind request queue on a pro-rata basis. It delegates to **WindManager::_settleQueueIfClaimable**, which calls **i_adapter.claimDeposit** to convert a specified amount of underlying assets into tranche shares, then iterates the FIFO queue to allocate those shares proportionally.

Inside **ParetoAdapter::claimDeposit**, the number of tranche shares to return is calculated as:

```
uint256 sharesToSend = (assetsToSettle * remainingShares) / remainingAssets;
```

When the tranche share token has fewer decimals than the underlying asset (or more generally, when 1 tranche share represents more than 1 unit of the underlying), **remainingShares < remainingAssets**. In this scenario, calling **claimDeposit** with **assetsToSettle = 1** causes **sharesToSend** to round down to zero. Despite returning zero shares, the adapter still decrements **s_remainingBatchAssets** by the requested amount:

```
s_remainingBatchAssets = remainingAssets - assetsToSettle;
s_remainingBatchShares = remainingShares - sharesToSend;
```

Back in **WindManager::_settleQueueIfClaimable**, when **mintedShares** is zero the allocation loop still increments **s_requestIdToClaimableAssets** for the front-of-queue request while adding zero to **s_requestIdToClaimableShares**:

```
uint256 sharesToAllocate = (assetsForRequest * mintedSharesRemaining) / claimableAssetsRemaining;
```

```
            s_requestIdToClaimableShares[requestId] += sharesToAllocate;
            s_requestIdToClaimableAssets[requestId] = allocatedAssets +
assetsForRequest;
            mintedSharesRemaining -= sharesToAllocate;
            claimableAssetsRemaining -= assetsForRequest;
```

Steps to reproduce:

1. Assume a deployment where 1 tranche share is worth X underlying tokens, with X > 1 (i.e., **remainingShares < remainingAssets** in the adapter).

2. A batch of deposit requests is pending and has been processed by the Pareto epoch queue (epoch price is set).

3. An operator calls **WindManager::settleQueue** with **_maxAssets = 1**.

4. **ParetoAdapter::claimDeposit** is invoked with **assetsToSettle = 1**. Due to integer division, **sharesToSend** rounds to zero. **s_remainingBatchAssets** is decremented by 1, but **s_remainingBatchShares** remains unchanged.

5. **WindManager::_settleQueueIfClaimable** receives **mintedShares = 0**. The queue loop increments **s_requestIdToClaimableAssets** by 1 but adds 0 to **s_requestIdToClaimableShares**.

6. The operator repeats steps 3-5 until **s_remainingBatchAssets** reaches zero and the batch is cleared.

7. Wind requests in the queue are marked as fully asset-allocated but hold zero claimable shares, resulting in escrows that finalize with no RWA collateral.

Note: Under the current deployment configuration (Pareto tranche tokens use 18 decimals, underlying is USDC with 6 decimals), **remainingShares** is always significantly larger than **remainingAssets**, so this rounding path is not reachable. The issue becomes exploitable only if the protocol integrates an adapter or tranche where the share-to-asset ratio inverts.

Recommendation

Add a zero-share guard in **WindManager::_settleQueueIfClaimable** immediately after calling **claimDeposit** to prevent silent asset consumption without corresponding share allocation:

```
    function _settleQueueIfClaimable(uint256 _maxAssets, bytes calldata
_adapterData) internal {
        uint256 claimableAssets = i_adapter.claimableDepositRequest(_adapterData);
        if (claimableAssets == 0) {
            return;
```

```
        }
        uint256 assetsToClaim = claimableAssets;
        if (_maxAssets > 0 && _maxAssets < claimableAssets) {
            assetsToClaim = _maxAssets;
        }
        uint256 mintedShares = i_adapter.claimDeposit(assetsToClaim, _adapterData);
+       require(mintedShares > 0, "WindManager: zero shares minted");
        uint256 claimableAssetsRemaining = assetsToClaim;
        uint256 mintedSharesRemaining = mintedShares;
```

3S-N13

WindManager::setMaxLeverageBps incorrectly prevents setting max leverage equal to min leverage

Id	3S--N13
Classification	None
Category	Suggestion
Status	Addressed in #4de6e2f.

Description

WindManager::setMaxLeverageBps is an admin-restricted function that updates **s_maxLeverageBps**, the upper bound on user leverage in basis points. A corresponding function, **WindManager::setMinLeverageBps**, sets the lower bound. Together, these two parameters define the valid leverage range for wind requests.

Setting **s_maxLeverageBps** equal to **s_minLeverageBps** is a valid configuration that restricts all users to a single fixed leverage value. However, **setMaxLeverageBps** uses a **<=** comparison when validating against **s_minLeverageBps**, which causes the transaction to revert when the admin attempts to set them equal.

This behavior is inconsistent with **setMinLeverageBps**, which uses a strict **>** comparison and correctly allows **_minLeverageBps == s_maxLeverageBps**.

Recommendation

Change the **<=** comparison to **<** so that **_maxLeverageBps == s_minLeverageBps** is permitted, consistent with the logic in **setMinLeverageBps**:

```
    function setMaxLeverageBps(uint256 _maxLeverageBps) external {
        _requireNotPaused();
        _checkRole(DEFAULT_ADMIN_ROLE);
        if (_maxLeverageBps <= BPS) {
            revert WindManager__InvalidMaxLeverageBps(_maxLeverageBps, BPS + 1);
        }
-       if (s_minLeverageBps > 0 && _maxLeverageBps <= s_minLeverageBps) {
+       if (s_minLeverageBps > 0 && _maxLeverageBps < s_minLeverageBps) {
            revert WindManager__InvalidMaxLeverageBps(_maxLeverageBps,
s_minLeverageBps + 1);
        }
```

```
    s_maxLeverageBps = _maxLeverageBps;
    emit MaxLeverageBpsSet(_maxLeverageBps);
}
```