yAudit

March 2026

Prepared for
CoW

Audited by
HHK
adriro

CoW-Euler Integration

Smart Contract Security Assessment

Contents

1 Review Summary

1.1 Protocol Overview

The CoW-Euler Integration contracts enable atomic leveraged position management (open, close, collateral swap) by coordinating CoW Protocol settlements with Euler Vault Kit (EVK) operations through the Ethereum Vault Connector (EVC). The system uses a chained wrapper pattern where authenticated solvers call `wrappedSettle()`, which executes custom EVC batch operations atomically around a CoW Protocol settlement. Users authorize operations via either EVC permit signatures or on-chain pre-approved hashes.

1.2 Audit Scope

This audit covers 10 smart contracts totaling approximately 929 lines of code across 5 days of review.

```
src/
├── CowEvcBaseWrapper.sol
├── CowEvcClosePositionWrapper.sol
├── CowEvcCollateralSwapWrapper.sol
├── CowEvcOpenPositionWrapper.sol
├── CowWrapper.sol
├── CowWrapperHelpers.sol
├── Errors.sol
├── Inbox.sol
├── InboxFactory.sol
└── PreApprovedHashes.sol
```

1.3 Risk Assessment Framework

1.3.1 Severity Classification

Severity	Description	Potential Impact
Critical	Immediate threat to user funds or protocol integrity	Direct loss of funds, protocol compromise
High	Significant security risk requiring urgent attention	Potential fund loss, major functionality disruption
Medium	Important issue that should be addressed	Limited fund risk, functionality concerns
Low	Minor issue with minimal impact	Best practice violations, minor inefficiencies
Undetermined	Findings whose impact could not be fully assessed within the time constraints of the engagement. These issues may range from low to critical severity, and although their exact consequences remain uncertain, they present a sufficient potential risk to warrant attention and remediation.	Varies based on actual severity
Gas	Findings that can improve the gas efficiency of the contracts.	Increased transaction costs
Informational	Code quality and best practice recommendations	Reduced maintainability and readability

Table 1: severity classification

1.4 Key Findings

Breakdown of Finding Impacts

Impact Level	Count
Critical	0
High	0
Medium	0
Low	2
Informational	3

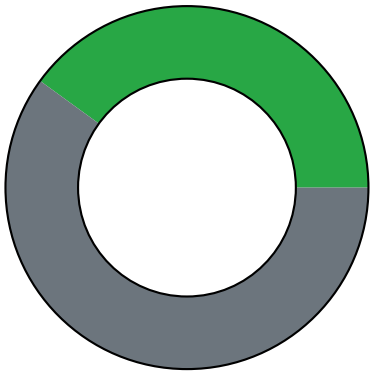

Figure 1: Distribution of security findings by impact level

1.5 Overall Assessment

The reviewed CoW-Euler integration is a well-structured and well-documented codebase that handles a non-trivial settlement and position-management flow with good architectural clarity. No major issues were identified during the engagement. The main risk area observed relates to solver-dependent settlement assumptions and missing wrapper-level post-settlement validation, which could enable low-severity griefing scenarios under a malicious or compromised solver.

2 Audit Overview

2.1 Project Information

Protocol Name: CoW
Repository: https://github.com/cowprotocol/euler-integration-contracts
Commit Hash: beb87e836e3876f4b6cfc47176f14d86278f7263

2.2 Audit Team

HHK, adriro

2.3 Audit Timeline

The audit was conducted from March 10 to 16, 2026.

2.4 Audit Resources

Code repositories, documentation and previous audit report by Cantina

Category	Mark	Description
Access Control	Good	Proper access control is in place. Wrappers are intended to be executed by authorized solvers.
Mathematics	Good	Correct use of mathematical operations.
Complexity	Good	The contracts are well-architected given the complex integration with CoW, EVC, and EVK.
Libraries	Good	The codebase is self-contained with minimal use of external dependencies.
Decentralization	Average	Solvers are trusted to behave correctly when performing on-chain actions. However, this trust assumption is acknowledged in the protocol's threat model, which includes penalties for misbehavior.
Code Stability	Good	The codebase remained stable during the review.
Documentation	Good	The contracts are well-documented, and detailed high-level documentation was provided to the auditors.
Monitoring	Good	Monitoring mechanisms are in place to track key events and changes within the system.
Testing and verification	Good	The codebase features a rich test suite with unit and end-to-end tests.

Table 2: Code Evaluation Matrix

2.5 Critical Findings

None.

2.6 High Findings

None.

2.7 Medium Findings

None.

2.8 Low Findings

2.8.1 Non-conformant EIP-712 struct hashing in `Inbox.isValidSignature()`

The `isValidSignature()` function in `Inbox` hashes the CoW `Order` struct by treating all 12 fields as raw 32-byte words. However, the `Order` type contains three `string` fields (`kind`, `sellTokenBalance`, `buyTokenBalance`) which, per EIP-712, must each be individually `keccak256`-hashed before being included in the struct hash.

Technical Details

The assembly block in `isValidSignature()` computes `structHash` by hashing 416 contiguous bytes (type hash + 12 × 32-byte words) starting from `orderData`. This treats every field uniformly as a fixed-size value.
EIP-712 specifies that dynamic types like `string` must be encoded as `keccak256(value)` within the struct hash. The CoW `Order` type has `kind`, `sellTokenBalance`, and `buyTokenBalance` declared as `string`. Skipping this step produces a struct hash that does not conform to the EIP-712 specification.
In practice, the CoW settlement contract itself uses the same non-standard encoding for these fields (treating them as pre-hashed `bytes32` values), so the computed hash will match the settlement contract's expected digest. The risk is limited to interoperability: any standard EIP-712 tooling or future contract that strictly follows the specification would compute a different hash for the same order, potentially causing signature verification failures.

Impact

Low. The hashing deviates from the EIP-712 standard for `string`-typed fields, but mirrors the CoW settlement contract's own encoding, so it functions correctly in the current integration. The impact is limited to potential interoperability issues with strict EIP-712 implementations.

Recommendation

Add a comment documenting that the encoding intentionally mirrors the CoW settlement contract's non-standard treatment of the `string` fields as pre-hashed `bytes32` values, and that this is a known deviation from EIP-712.

Developer Response

Fixed in cc95202. Added a comment.

2.8.2 Wrappers lack sufficient post-settlement validation

All three wrappers lack adequate verification that the CoW settlement actually executed the user's trade. A compromised or malicious solver can manipulate the settlement to bypass existing checks, consuming wrapper-level authorization and degrading the user's position.

Technical Details

Since `settleData` is fully solver-controlled, a solver can call `settle()` with empty `trades` (or trades that exclude the user's order), or use the settlement's `interactions` to send trivial token amounts that bypass existing checks.
Close position: The existing `NoSwapOutput` check (CowEvcClosePositionWrapper.sol#L222-L224) is trivially bypassed — a solver can send 1 wei of the borrow asset to the Inbox via settlement `interactions`. The check passes, the wrapper repays 1 wei of debt, and the unused collateral vault tokens are returned to the user's account (L248-252). Since the collateral vault remains enabled, the health check passes with the position essentially unchanged, but the wrapper authorization is consumed.
Open position: No post-settlement check exists in `_evcInternalSettle`. The batch items before settlement execute (enableCollateral, enableController, deposit collateral, borrow tokens). Without the swap converting borrowed tokens to additional collateral, the EVC health check will likely revert the batch due to undercollateralization. However, if the user's initial `collateralAmount` is large relative to `borrowAmount`, the health check could pass — leaving the user with an open debt position and unswapped borrowed tokens. The wrapper's pre-approved hash or permit nonce is consumed.
Collateral swap: No post-settlement check exists in `_evcInternalSettle`. When `owner == account` and `disableSourceCollateral == false`, the only pre-settlement action is `enableCollateral(toVault)`. A no-op settlement means nothing changes, the health check passes trivially, and the wrapper-level authorization is consumed for nothing.
In all cases, the CoW order itself is not invalidated on the settlement contract (`filledAmount` is not updated), but the wrapper-level authorization is burned, forcing the user to sign a new permit or submit a new pre-approved hash.

Impact

Low. Requires a compromised or malicious solver (bonded actors subject to slashing). No direct fund loss — users retain their assets in all scenarios (close position collateral is recoverable from the Inbox). The impact is limited to griefing: consuming wrapper-level authorizations, forcing users to re-authorize on-chain, and potentially degrading health factors.

Recommendation

Add a user-specified minimum output field to each wrapper's params struct. Since params are included in both the pre-approved hash and the EVC permit signature, the minimum is automatically authenticated without any changes to the authorization flows.

Close position — add `minDebtAssetOut` to `ClosePositionParams` and replace the existing `NoSwapOutput` check in `_evcInternalSettle`:

```
struct ClosePositionParams {
    // ... existing fields ...
    uint256 minDebtAssetOut; // minimum borrow asset received from swap
}

// In _evcInternalSettle, replace the NoSwapOutput check:
uint256 swapOutput = swapResultBalance - swapBeforeResultBalance;
require(
    swapOutput >= params.minDebtAssetOut,
    InsufficientSwapOutput(swapOutput, params.minDebtAssetOut)
);
```

Open position — add `minCollateral` to `OpenPositionParams` and check in `_evcInternalSettle`:

```
struct OpenPositionParams {
    // ... existing fields ...
    uint256 minCollateral; // minimum collateral vault token balance after settlement
}

// In _evcInternalSettle:
_next(settleData, remainingWrapperData);
require(
    IERC20(collateralVault).balanceOf(params.account) >= params.minCollateral,
    InsufficientCollateral(params.account)
);
```

Collateral swap — add `minCollateral` to `CollateralSwapParams` and check in `_evcInternalSettle`:

```
struct CollateralSwapParams {
    // ... existing fields ...
    uint256 minCollateral; // minimum destination vault token balance after settlement
}

// In _evcInternalSettle:
_next(settleData, remainingWrapperData);
require(
    IERC20(params.toVault).balanceOf(params.account) >= params.minCollateral,
    InsufficientCollateral(params.account)
);
```

This provides explicit wrapper-level slippage protection, prevents no-op and trivial-amount settlement griefing, and is automatically covered by both authorization flows since the new fields are part of the hashed/signed params struct.

Developer Response

Acknowledged.

2.9 Gas Savings Findings

None.

2.10 Informational Findings

2.10.1 Close position wrapper repay amount uses existing balance instead of delta

The comment at CowEvcClosePositionWrapper.sol#L226 states the repay amount equals "however much we get from swapping", but the code uses the Inbox's total borrow asset balance rather than the swap output delta.

Technical Details

In CowEvcClosePositionWrapper.sol#L226-L227:

```
// the amount we will *actually* repay is the same as however much we get from swapping
uint256 repayAmount = swapResultBalance;
```

`swapResultBalance` is the Inbox's total borrow asset balance, not the delta (`swapResultBalance - swapBeforeResultBalance`). If the Inbox holds pre-existing borrow asset tokens, those are silently consumed for additional debt repayment or returned to the owner as excess. The behavior is beneficial to the Inbox owner but contradicts the comment.

Impact

Informational. No security impact — pre-existing tokens in the Inbox conceptually belong to the owner (who can recover them via `callTransfer`). The extra repayment benefits the user.

Recommendation

Either use the swap delta to match the comment, or update the comment to reflect the actual behavior.

Developer Response

Fixed in 539f99b.

2.10.2 Misleading comments about `remainingWrapperData` in `_evcInternalSettle()`

In both `CowEvcClosePositionWrapper._evcInternalSettle()` and `CowEvcCollateralSwapWrapper._evcInternalSettle()`, the inline comment states that `wrapperData` is "empty since we've already processed it in `_wrap`". This is misleading as `remainingWrapperData` can be empty (when this is the last wrapper in the chain) or non-empty (when additional wrappers are chained after this one).

Technical Details

The comment in both wrappers reads: "Use CowWrapper's `_next` to call the settlement contract / wrapperData is empty since we've already processed it in `_wrap`". The `_next()` function's behavior depends on `remainingWrapperData`: if it is empty, `_next()` calls the settlement contract directly; if non-empty, it extracts the next wrapper address and continues the chain. The comment incorrectly implies `remainingWrapperData` is always empty, which obscures the chaining capability of the wrapper architecture.

Impact

Informational.

Recommendation

Update the comments in both `CowEvcClosePositionWrapper._evcInternalSettle()` and `CowEvcCollateralSwapWrapper._evcInternalSettle()` to accurately describe that `remainingWrapperData` may contain additional chained wrapper data, and that `_next()` handles both cases.

Developer Response

Fixed in 40e396c.

2.10.3 `evcInternalSettle()` caller check could additionally verify `onBehalfOfAccount`

The `evcInternalSettle()` function in `CowEvcBaseWrapper` only checks that `msg.sender == address(EVC)`. Since anyone can initiate an `EVC.batch()` call, and the EVC forwards calls on behalf of the specified `onBehalfOfAccount`, the check could be strengthened by also verifying that `EVC.getCurrentOnBehalfOfAccount() == address(this)`.

Technical Details

The batch item that triggers `evcInternalSettle()` is constructed with `onBehalfOfAccount: address(this)`, meaning the EVC should be executing the callback on behalf of the wrapper contract itself. However, the current check only validates the caller is the EVC — it does not confirm that the EVC is acting on behalf of `address(this)`. The `expectedEvcInternalSettleCallHash` transient storage check already ensures that the callback data matches what the wrapper expected, which provides strong protection against unauthorized invocations. Adding the `onBehalfOfAccount` check would provide defense-in-depth by also validating the execution context matches expectations, consistent with how the batch item is configured.

Impact

Informational.

Recommendation

Add `require(EVC.getCurrentOnBehalfOfAccount(address(0)) == address(this))` alongside the existing `msg.sender == address(EVC)` check in `evcInternalSettle()`.

Developer Response

Fixed in 5e41bb0.

2.11 Final Remarks

The CoW-Euler integration enables users to atomically manage leveraged positions by combining CoW Protocol settlement with Euler's EVC and EVK flows for opening, closing, and collateral swaps. Although the system coordinates several moving parts and depends on trusted solver execution within its stated threat model, the reviewed codebase is well-architected for that complexity. The contracts are clearly documented, the structure is consistent across wrappers, and the available unit and end-to-end tests contributed positively to the review process.
No critical, high, or medium severity issues were identified during this engagement. The most notable concern was a low-severity gap in post-settlement validation across the wrappers, where a malicious or compromised solver could grief users by consuming wrapper-level authorizations without meaningfully executing the intended trade. The remaining findings were informational and mostly centered on clarifying code behavior, comments, and defense-in-depth checks, and these were promptly addressed by the team. The CoW team was responsive throughout the review, fixing the informational issues quickly and acknowledging the remaining low-severity recommendation.