CoW Protocol – Technical Security Review

October 29, 2025
Mourning Dove

This report represents an independent, educational security analysis performed under the public Immunefi bounty scope for CoW Protocol. No private or undisclosed data was accessed during this work. We would like to thank the CoW Protocol security and development team for their willingness to review this document. Their transparency and responsiveness made this analysis both possible and rewarding.

The analysis was conducted between September 9^{th}, 2025 and October 22^{nd}, 2025, targeting the bounty posted on Immunefi. No vulnerabilities were found during these efforts. This document discusses only publicly available information — including on-chain data and open-source off-chain CoW Protocol components. We provide an overview of the relationship between the off-chain and on-chain services. Section 3 presents a detailed outline of the security checks performed using the CoW Protocol SDK. To strengthen our analysis, we replicated real transactions using Foundry tests on a forked instance of the Ethereum mainnet, providing further assurance of the protocol's robustness. The tests are designed to be simple, extensible, and efficient, allowing other security researchers to expand upon this work and validate additional business logic within the protocol. Forking the mainnet enables these tests to run with minimal additional contract deployments, providing a safe, local, and reproducible environment for further security analysis.

Contents

1 Scope

Immunefi's posting of the bounty states that only smart contracts are in scope of the reward. Three core smart contracts are analyzed for in scope vulnerabilities. We target the GPv2Settlement smart contract in our test suite. The bounty posting has a clear description in Immunefi which we provide in Table 2.

Table 1: CoW Protocol Core Contracts

GPv2Settlement
GPv2VaultRelayer
GPv2AllowListAuthentication

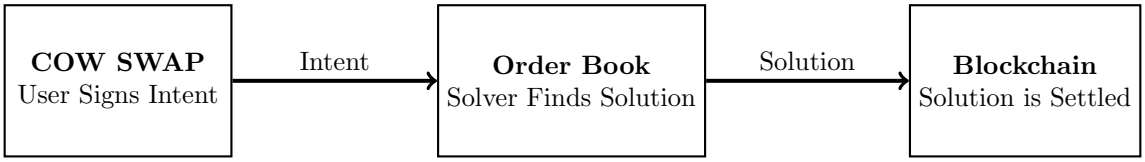
Table 2: Posted Scope On Immunefi

Severity	**Description**
Critical	Unauthorized control of authentication contract and solvers
Critical	Unauthorized execution of funded trades
Critical	Unauthorized settlement execution
Critical	Misuse of solver privileges for trades
Critical	Unauthorized token transfers
Critical	Unauthorized access to user funds
High	Manipulation of settlement interactions
High	Unauthorized solver removal
High	Denial of service for solvers
Medium	Unauthorized storage manipulation
Medium	Unauthorized order invalidation

2 CoW Protocol

If you are already familiar with the protocol, we recommend skipping to security analysis in Section 3. The following subsections explore the flow of an order in the protocol. CoW Protocol, formerly known as Gnosis Protocol v2, allows users to trade crypto assets using intents and fair combinatorial batch auctions. A user creates a signed order request and submits the request to an off-chain order book service. This order book service provides a platform for solvers to compete for a solution derived from a pool of intents. Solvers verify the intents and optimize the exchange by finding a coincidence of wants. The solver settles the solution containing the user's order in an on-chain transaction. Since the user signed the order, the exchange is executed on their behalf.

Figure 1: Cow Protocol Flow



2.1 Vault Relayer Approval

User funds on the CoW Protocol are managed through a contract known as the Vault Relayer (GPv2VaultRelayer). The Vault Relayer only accepts calldata originating from the settlement contract. To execute a trade for the first time, a user must complete an on-chain approval transaction, granting the Vault Relayer permission to spend tokens on their behalf. Once approval is granted, the user can specify the assets to be traded and sign an intent off-chain to execute the trade.

2.2 Intent

An intent is the encapsulation of a user signature and order information such as buy/sell token, receiver of the funds, and whether an order is partially fillable. Specifically, intents contain the fields required by the GPv2Trade contract data. GPv2Trade data contains the executed amount, all the fields required to rebuild GPv2Order data, and the user's signature of that data. Signing the order data allows solvers to make on-chain requests on the user's behalf. Figure 2 compares the two data structs.

Figure 2: GPv2Order Data (Left) and GPv2Trade Data (Right)

```
struct Data {
    IERC20 sellToken;
    IERC20 buyToken;
    address receiver;
    uint256 sellAmount;
    uint256 buyAmount;
    uint32 validTo;
    bytes32 appData;
    uint256 feeAmount;
    bytes32 kind;
    bool partiallyFillable;
    bytes32 sellTokenBalance;
    bytes32 buyTokenBalance;
}
```

```
struct Data {
    uint256 sellTokenIndex;
    uint256 buyTokenIndex;
    address receiver;
    uint256 sellAmount;
    uint256 buyAmount;
    uint32 validTo;
    bytes32 appData;
    uint256 feeAmount;
    uint256 flags;
    uint256 executedAmount;
    bytes signature;
}
```

Most of the fields in `GPv2Order.Data` seem standard besides the appData hash. This field is a **keccak256** hash of off-chain metadata, such as the data in Figure 3. Acceptable hashes for `app-data` can be found in the migrations for the off-chain services. The order book service requires a pre-image of this hash to be present in its database before processing the order. While we may explore validation of appData in future work, such findings are likely out of scope for the Immunefi bounty. Further details about the `appData` hash are given within the codebase.

Figure 3: App Data of Transaction

```
{
    "appCode": "CoW Swap",
    "environment": "production",
    "metadata": {
      "orderClass": {"orderClass": "limit"},
      "quote": {"slippageBips": 0},
      "utm": {
        "utmContent": "menubar-dao-nav-cowswap",
        "utmMedium": "web",
        "utmSource": "swap.cow.fi"
      }
    },
    "version": "1.6.0"
}
```

2.3 CoW Swap

CoW Swap is a user interface that allows users to connect their wallets and exchange crypto assets on the CoW Protocol. The interface provides a convenient platform for creating signed intents and granting approval to the protocol's Vault Relayer. Signed intents are sent to an off-chain order book service to be executed as part of batch settlements.
Orders originating from CoW Swap appear to trigger a call to an intermediate contract, the CoW Swap Eth Flow (EthFlow), which performs token transfers and internal transactions directly to the GPv2Settlement contract. Transactions targeting the `createOrder` method in the EthFlow contract are executed by the order service after receiving order data from CoW Swap. The argument for this method, `EthFlowOrder`, corresponds exactly to the non-signature fields in the order creation request submitted via the UI. We omit a detailed analysis of the EthFlow contract, as it is out of scope for the current bounty.

2.4 Solvers and Interactions

Solvers verify intents in the order book and optimize the exchange by identifying a coincidence of wants. A solver's solution is then settled on-chain, finalizing the transaction. The solution provides the executed amount for each signed order, completing the `GPv2Trade.Data`.
Orders without a direct match are settled by the solver using on-chain liquidity sources via *interactions*. The SDK provides predefined interaction types that can be incorporated into our Foundry tests, simplifying the testing of these edge cases. Figure 4 illustrates the fields required for a sample interaction.

Figure 4: Interaction Type Example in SDK

```
const interactionIntra1: Interaction = {
      target: TOKEN_ADDRESS,
      value: 0,
      callData: CALL_DATA
    }
```

2.5 GPv2Settlement Contract

Solver solutions are settled through the smart contract GPv2Settlement via its `settle` method. As stated in the documentation, a settlement consists of a list of buy and sell tokens, a list of `GPv2Trade.Data`, and a list of interactions. Etherscan may parse the calldata and display the function as `MoooZ1089603480`. Calldata decoder tools reveal that the `settle` function parameters are specified in the transaction calldata. A closer inspection of the `settle` function code shows that the argument type is `GPv2Trade`, which corresponds to the function parameters.

Figure 5: Comparison of `settle` Parameters and `GPv2Trade.Data`

```
settle(
    address[],
    uint256[],
    (uint256, uint256,
        address,
        uint256,
        uint256,
        uint32,
        bytes32,
        uint256,
        uint256,
        uint256,
        bytes
    )[],
    (
        address,
        uint256,
        bytes
    )[][3]
)
```

```
struct Data {
    uint256 sellTokenIndex;
    uint256 buyTokenIndex;
    address receiver;
    uint256 sellAmount;
    uint256 buyAmount;
    uint32 validTo;
    bytes32 appData;
    uint256 feeAmount;
    uint256 flags;
    uint256 executedAmount;
    bytes signature;
}
```

Recall that a user signs the GPv2Order data, not the GPv2Trade data. The GPv2Settlement contract ensures that a GPv2Trade is valid by verifying it against a previously signed `GPv2Order.Data`. Within the contract's `settle` function, the exact EIP-712 digest of the original order is reconstructed from the trade parameters. The user's provided signature is then used to authenticate the order owner. Execution proceeds only if the trade's final price meets or exceeds the original limit price. This on-chain check enforces a bridge between off-chain authorization (`GPv2Order`) and on-chain settlement (`GPv2Trade`), preventing any manipulation of the order's core terms.

2.6 Example: User to Smart Contract Exchange

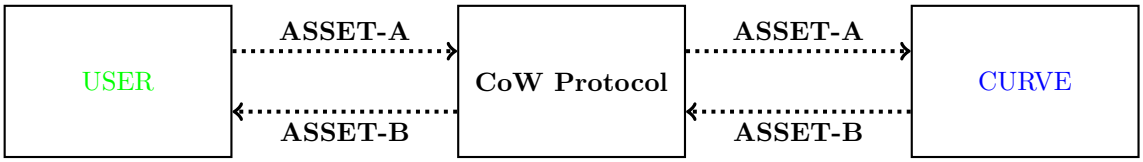
To consolidate the preceding subsections, we examine a real transaction invoking the `settle` function of the GPv2Settlement contract (SETTLE). Examination of the transaction in Etherscan reveals the transaction originator — the solver (SOLVER). Table 3 lists the complete set of event logs associated with this transaction.

Event Log	Contract Address	Sender	Receiver	Owner	Spender
Trade	SETTLE			USER	
Transfer	**ASSET-A**	USER	SETTLE		
Approval	**ASSET-A**			SETTLE	CURVE
Interaction	SETTLE				
Transfer	**ASSET-A**	SETTLE	CURVE		
Transfer	**ASSET-B**	CURVE	SETTLE		
TokenExchange	CurveStableSwapNG				
Interaction	SETTLE				
Transfer	**ASSET-B**	SETTLE	USER		
Settlement	SETTLE				

Table 3: Event Logs of Transaction

From the event log, we can extract an `orderUID`. Using this value (or the transaction hash), the corresponding order can be inspected in the CoW Explorer, revealing the exact intent originally signed by the user. In this example, the order is partially fillable, and the transaction under analysis represents one such partial fulfillment.
There are two user addresses in these transaction logs. One user (USER) is seeking to sell **ASSET-A** for **ASSET-B**. The other user (CURVE) is an exchange contract that handles the transaction of **ASSET-A** for **ASSET-B**. Recall, interactions can be used by solvers to complete the order when a coincidence of wants is not found. Figure 6 is a diagram outlining the transaction.

Figure 6: Transaction Overview



In the first transaction of Table 3 we see the `Owner` value is USER indicating USER signed the intent to trade **ASSET-A** for **ASSET-B**. This can be further verified this by considering the `receiver` field in the transaction's calldata. USER signed an intent to trade and SOLVER was the solver that submitted the solution to the settlement contract. Figure A.1 in the appendix has the solidity test verifying the balance change for the involved addresses.

Figure 7: Two Interactions. Approval (Left) and Transfer Amounts (Right)

```
[
  "Exchange Address",
  "ASSET-A Amount"
]
```

```
[
  "0",
  "1",
  "ASSET-A Amount",
  "ASSET-B Amount"
]
```

Figure 7 displays the two interactions in the transactions calldata. Both of these interactions are seen in our event logs in Table 3. The first interaction, in the order of event logs, is an approval

targeting the **ASSET-A** Token contract. SETTLE is granting CURVE permissions to spend the **ASSET-A** tokens on SETTLE's behalf.
In the second interaction we see the amounts of **ASSET-A** and **ASSET-B** to be traded on the Curve exchange. Since approval has been granted to CURVE to spend the specified **ASSET-A**, the exchange transfers **ASSET-A** from SETTLE to CURVE. CURVE then transfers the specified amount of **ASSET-B** to SETTLE, completing the interaction. Now with possession of **ASSET-B**, SETTLE transfers the **ASSET-B** to USER.

3 Security Review

All tests and files discussed in this section are available in the report's GitHub repository. Our testing environment uses the CoW Protocol SDK to generate settlement calldata and a Foundry-based Solidity test suite to execute that calldata on a local chain. Using Foundry's `ffi` cheatcode, we can directly call JavaScript scripts from within Solidity tests. With these integrated modules, orders can be signed and settled successfully on our local blockchain. A complete version of our implementation is shown in Item A.2 of the appendix. We then test the generated calldata on a local Anvil chain using Solidity, following the same approach as the first test in the appendix.

3.1 Vault Approval

A user must grant the Vault Relayer approval to spend tokens on the user's behalf. If this user has not granted the Vault Relayer approval, the revert will occur in the Vault Relayer contract as seen in Figure 8. In the example error, **0xC92E8bdf7...** is the address of the Vault Relayer.

Figure 8: On-Chain Approval Error

```
0xC92E8bdf79f0507f65a392b0ab4667716BFE0110::transferFromAccounts(
    [
        (
            0x70997970C51812dc3A010C7d01b50e0d17dc79C8,
            ASSET_A,
            AMOUNT,
            ...
            )
        ])
```

Our test user address, **0x7099797...**, is attempting to transfer **ASSET-A**. Foundry's `onPrank` cheatcode allows us to simulate transactions as if sent by a specific user. Sending an approval transaction to the Vault Relayer resolves the issue. We test that the transaction results in revert when the user has not provided approval to spend their tokens.

Table 4: Vault Relayer Approval Test Cases

Test Title	Description
Successful Approval	Including the approval leads to a successful transaction. Acts as control for other tests.
Reverts When No Approval	Reverts as expected when the user does not grant the Vault Relayer approval.

3.2 Executed Amount and Clearing Prices

Solvers control the `executedAmount` and `clearingPrices`, which determine the amount of tokens a user will sell and receive in the settlement. These values are especially important for partially fillable orders, where the user allows execution within a range between 0 and the total sell amount of the order.

Clearing prices are particularly interesting to test because they are determined after the user has signed their order. During intent creation, the user specifies the sell and buy amounts for the assets they wish to exchange. A solver must not choose an executed amount and clearing prices that result in a smaller return than the ratio implied by the user's sell and buy amounts.

For example, suppose a user creates an intent to sell 2 of **ASSET-A** for 6 of **ASSET-B**. If a solver attempts to set the executed amount to 1 of **ASSET-A** with a clearing price of only 2 of **ASSET-B**, the transaction reverts. The user expects at least 3 of **ASSET-B** for 1 **ASSET-A**, and any execution that yields less violates the price ratio guaranteed by the signed intent.

Table 5: Executed Value Tests

Test Title	Description
Executed Amount Resulting in Smaller than Expected Return	Ensures the transaction reverts when the solver provides a clearing price that yields less than the minimum return implied by the order's sell-to-buy ratio.

3.3 Signature Validation

Signing schemes are documented in the CoW documentation. To prevent replay attacks, the address of the smart contract that will process the GPv2Trade data must be included when computing the signature. For mainnet trades this is the GPv2Settlement contract. The domain separator shown in Figure 9 targets the settlement contract and uses `chainId = 1` for mainnet.

Figure 9: Domain Separator

```
{
  name: "Gnosis Protocol",
  version: "v2",
  chainId: 1,
  verifyingContract: "0x9008D..."
}
```

In the normal flow a user signs the order via the CoW Swap UI. A containerized playground for CoW Swap and other off-chain services is available in the repository playground. For testing, we bypass the off-chain services and create/sign orders programmatically using the CoW SDK. This SDK was essential — getting the signature bytes correct is critical for the protocol. By integrating the SDK with Foundry tests, we can sign orders and settle them on a local fork of mainnet.

Table 6: Signature Validation Checks

Test Title	Description
Invalid Domain Separator	Generate a signature using a modified `chainId` or `verifyingContract` and verify that on-chain validation rejects it.
Replay Attack Attempt	Reuse a previously valid signature with altered non-hashed parameters (for example, timestamp or nonce) to confirm replay protection is active.

A Code

A.1 Replay Transaction

```
pragma solidity ^0.8.20;

import {Test, console2} from "forge-std/Test.sol";
import "../src/PublicValues.sol";

interface IERC20 {
    function balanceOf(address account) external view returns (uint256);
}

contract Replay is Test, PublicValues {

    IERC20 internal sellToken;
    IERC20 internal buyToken;

    function setUp() public {
        uint256 forkId = vm.createFork(MAINNET_URL, T_BLOCK_NUMBER_MINUS_ONE);
        vm.selectFork(forkId);

        sellToken = IERC20(SELL_TOKEN_ADDRESS);
        buyToken = IERC20(BUY_TOKEN_ADDRESS);

    }

    function test_replayAndVerifyTrade() public {

        bool success;
        bytes memory returnData;

        uint256 initialSellBalance = sellToken.balanceOf(T_RECEIVER);
        uint256 initialBuyBalance = buyToken.balanceOf(T_RECEIVER);

        vm.startPrank(SOLVER);
        (success, returnData) = COW_SETTLEMENT.call{value: T_ETH_VALUE}(T_CALLDATA);
        vm.stopPrank();

        assertTrue(success);

        uint256 finalSellBalance = sellToken.balanceOf(T_RECEIVER);
        uint256 finalBuyBalance = buyToken.balanceOf(T_RECEIVER);

        assertTrue(finalSellBalance < initialSellBalance);
        assertTrue(finalBuyBalance > initialBuyBalance);
    }

}
```

A.2 Generate Calldata

```
import pkg from '@cowprotocol/contracts';
import pkg2 from "ethers"
const {
    Order,
    OrderBalance,
    OrderKind,
    SettlementEncoder,
    SigningScheme,
    TypedDataDomain
} = pkg;
const { ethers, Wallet } = pkg2

const rpcUrl = RPC_URL;
const provider = new ethers.providers.JsonRpcProvider(rpcUrl);
const testSecretKey = TEST_PRIVATE_KEY;
const walletSigner = new Wallet(testSecretKey, provider);
const order: Order = {
    sellToken: SELL_TOKEN_ADDRESS,
    buyToken: BUY_TOKEN_ADDRESS,
    receiver: RECEIVER_ADDRESS,
    sellAmount: SELL_AMOUNT,
    buyAmount: BUY_AMOUNT,
    validTo: VALID_TO,
    appData: APP_DATA,
    feeAmount: "0",
    kind: OrderKind.SELL,
    partiallyFillable: false,
    sellTokenBalance: OrderBalance.ERC20,
    buyTokenBalance: OrderBalance.ERC20,
};
const domainData: TypedDataDomain = {
    name: 'Gnosis Protocol',
    version: 'v2',
    chainId: 1,
    verifyingContract: '0x9008D19f58AAbD9eD0D60971565AA8510560ab41',
};
const settlementEncoder: SettlementEncoder = new SettlementEncoder(domainData)
await settlementEncoder.signEncodeTrade(order, walletSigner, SigningScheme.EIP712)
const encoded = settlementEncoder.encodedSettlement(
    {
        ASSET_A: CLEARING_AMOUNT_A,
        ASSET_B: CLEARING_AMOUNT_B
    }
)
console.log(encoded)
```