CoW Protocol

ComposableCoW &
ExtensibleFallbackHandler

by Ackee Blockchain

4.8.2023

Contents

1. Document Revisions

1.0	Final report	28.7.2023
1.1	Fix review	3.8.2023
1.2	Fix review	4.8.2023

2. Overview

This document presents our findings in reviewed contracts.

2.1. Ackee Blockchain

Ackee Blockchain is an auditing company based in Prague, Czech Republic, specializing in audits and security assessments. Our mission is to build a stronger blockchain community by sharing knowledge – we run free certification courses School of Solana, Summer School of Solidity and teach at the Czech Technical University in Prague. Ackee Blockchain is backed by the largest VC fund focused on blockchain and DeFi in Europe, RockawayX.

2.2. Audit Methodology

1. **Technical specification/documentation** - a brief overview of the system is requested from the client and the scope of the audit is defined.
2. **Tool-based analysis** - deep check with automated Solidity analysis tools and Woke is performed.
3. **Manual code review** - the code is checked line by line for common vulnerabilities, code duplication, best practices and the code architecture is reviewed.
4. **Local deployment + hacking** - the contracts are deployed locally and we try to attack the system and break it.
5. **Unit and fuzz testing** - run unit tests to ensure that the system works as expected, potentially write missing unit or fuzz tests.

2.3. Finding classification

A *Severity* rating of each finding is determined as a synthesis of two sub-ratings: *Impact* and *Likelihood*. It ranges from *Informational* to *Critical*.

If we have found a scenario in which an issue is exploitable, it will be assigned an impact rating of *High*, *Medium*, or *Low*, based on the direness of the consequences it has on the system. If we haven't found a way, or the issue is only exploitable given a change in configuration (such as deployment scripts, compiler configuration, use of multi-signature wallets for owners, etc.) or given a change in the codebase, then it will be assigned an impact rating of *Warning* or *Info*.

Low to *High* impact issues also have a *Likelihood*, which measures the probability of exploitability during runtime.

The full definitions are as follows:

Severity

		Likelihood			
		High	**Medium**	**Low**	**-**
Impact	**High**	Critical	High	Medium	-
	Medium	High	Medium	Low	-
	Low	Medium	Low	Low	-
	Warning	-	-	-	Warning
	Info	-	-	-	Info

Table 1. Severity of findings

Impact

- **High** - Code that activates the issue will lead to undefined or catastrophic consequences for the system.
- **Medium** - Code that activates the issue will result in consequences of serious substance.
- **Low** - Code that activates the issue will have outcomes on the system that are either recoverable or don't jeopardize its regular functioning.
- **Warning** - The issue cannot be exploited given the current code and/or configuration (such as deployment scripts, compiler configuration, use of multi-signature wallets for owners, etc.), but could be a security vulnerability if these were to change slightly. If we haven't found a way to exploit the issue given the time constraints, it might be marked as a "Warning" or higher, based on our best estimate of whether it is currently exploitable.
- **Info** - The issue is on the borderline between code quality and security. Examples include insufficient logging for critical operations. Another example is that the issue would be security-related if code or configuration (see above) was to change.

Likelihood

- **High** - The issue is exploitable by virtually anyone under virtually any circumstance.
- **Medium** - Exploiting the issue currently requires non-trivial preconditions.
- **Low** - Exploiting the issue requires strict preconditions.

2.4. Review team

Member’s Name	Position
Štěpán Šonský	Lead Auditor
Jan Kalivoda	Auditor
Josef Gattermayer, Ph.D.	Audit Supervisor

2.5. Disclaimer

We’ve put our best effort to find all vulnerabilities in the system, however our findings shouldn’t be considered as a complete list of all existing issues. The statements made in this document should not be interpreted as investment or legal advice, nor should its authors be held accountable for decisions made based on them.

3. Executive Summary

Revision 1.0

CoW Protocol engaged Ackee Blockchain to perform a security review of the ComposableCoW & ExtensibleFallbackHandler with a total time donation of 8 engineering days in a period between July 18 and July 28, 2023 and the lead auditor was Štěpán Šonský.

The audit has been performed on the following scope:

- https://github.com/rndlabs/composable-cow
    - Commit `cd893fa`
    - All contracts
- https://github.com/rndlabs/safe-contracts
    - Commit `e53ffea`
    - `contracts/handler/ExtensibleFallbackHandler.sol`
    - All contracts in `contracts/handler/extensible/`

We began our review by using static analysis tools, namely Woke. We then took a deep dive into the logic of the contracts. For testing, we have involved Woke testing framework (see Appendix C for outputs). During the review, we paid special attention to:

- replay attacks,
- signature validation,
- payload manipulation,
- detecting possible reentrancies,
- ensuring the arithmetic of the system is correct,

- the correctness of encoding/decoding data,
- ERC-1271 compliance,
- looking for common issues such as data validation.

Our review resulted in 13 findings, ranging from Informational to Medium severity. The most severe one M1: Oracle data validation reveals missing data validations from oracles. Other issues are low-severity data validations, warnings and informational findings, which are recommendations rather than issues. The overall code quality and architecture are professional. The whole project is well documented and contains in-code NatSpec documentation and detailed comments.

Ackee Blockchain recommends CoW Protocol:

- add oracle data validations,
- be aware of zero-address validations,
- unify syntax and naming,
- address all reported issues.

See Revision 1.0 for the system overview of the codebase.

Revision 1.1

The review was done on the given commits:

- `27ec79b` for ComposableCow
- `11273c1` for ExtensibleFallbackHandler

Nearly all issues and recommendations have been addressed, L1: Constructor data validation is marked as acknowledged, and W1: GPv2Order data tampering is invalidated. However, we identified a new critical severity issue C1: StopLoss arithmetic mismatches which occurred during the fixing of M1:

Oracle data validation, and can lead to loss of users' funds due to unrealized stop-loss orders in special cases. Also, the current implementation disables stop-loss orders to be used with tokens with similar exchange rates due to precision loss.

Ackee Blockchain recommends CoW Protocol:

- fix the critical issue,
- be extra careful about decimal arithmetics,
- introduce scaling for `strike` calculations.

See Revision 1.1 for the review of the updated codebase and additional information we consider essential for the current scope.

Revision 1.2

The review was done on the ComposableCow commit `bd2634d`, the ExtensibleFallbackHandler commit was not changed since revision 1.1.

The critical issue C1: StopLoss arithmetic mismatches has been fixed according to our recommendations, and the decimals handling in the M1: Oracle data validation issue is now implemented properly.

See Revision 1.2 for the review of the updated codebase and additional information we consider essential for the current scope.

4. Summary of Findings

The following table summarizes the findings we identified during our review. Unless overridden for purposes of readability, each finding contains:

- a *Description*,
- an *Exploit scenario*,
- a *Recommendation* and if applicable
- a *Fix*.

There might often be multiple ways to solve or alleviate the issue, with varying requirements regarding the necessary changes to the codebase. In that case, we will try to enumerate them all, clarifying which solves the underlying issue better (albeit possibly only with architectural changes) than others.

	Severity	Reported	Status
M1: Oracle data validation	Medium	1.0	Fixed
L1: Constructor data validation	Low	1.0	Acknowledged
W1: GPv2Order data tampering	Warning	1.0	Invalid
W2: Revert conditions inconsistency	Warning	1.0	Fixed
W3: Vulnerable MerkleProof library	Warning	1.0	Fixed
W4: GoodAfterTime order is missing the receiver address	Warning	1.0	Fixed
I1: Unnecessary SafeMath	Info	1.0	Fixed

	Severity	Reported	Status
I2: Missing cabinet cleanup	Info	1.0	Fixed
I3: Errors in the documentation	Info	1.0	Fixed
I4: TradeAboveThreshold order receiver naming	Info	1.0	Fixed
I5: Inconsistent error	Info	1.0	Fixed
I6: Commented-out code	Info	1.0	Fixed
I7: Inconsistent naming	Info	1.0	Fixed
C1: StopLoss arithmetic mismatches	Critical	1.1	Fixed

Table 2. Table of Findings

5. Report revision 1.0

5.1. System Overview

This section contains an outline of the audited contracts. Note that this is meant for understandability purposes and does not replace project documentation.

Contracts (ComposableCoW)

Contracts we find important in ComposableCoW for better understanding are described in the following section.

ComposableCow.sol

The core contract for creating and authorizing conditional orders. It holds the owner's Merkle roots in the `roots` mapping for authorizing multiple conditional orders and the `singleOrders` mapping is used to authorize single orders based on `msg.sender` and order hash. Orders can be additionally protected using swap guards, which are stored in the `swapGuards` mapping.

BaseConditionalOrder.sol

Parent contract for all types of conditional orders described below. `BaseConditionalOrder` inherits from the `IConditionalOrderGenerator` interface, which defines two important functions `verify` and `getTradeableOrder`. The function `verify` checks if the `_hash` passed by the parameter equals the hash of order data returned from the `getTradeableOrder` function, which is `virtual` in `BaseConditionalOrder` and every type of conditional order implements its own logic.

TWAP.sol

TWAP order splits the order into multiple discrete orders which are executed in the `frequency` interval (1 second to 1 year) and these partial orders sells always the same amount of tokens. Max price is limited by `minPartLimit`.

TWAPOrder.sol

Library for validating TWAP order data and generating individual order parts. It uses `TWAPOrderMathLib` to calculate `validTo` timestamp.

TWAPOrderMathLib.sol

Contains only the function `calculateValidTo`, which calculates `validTo` timestamp for the single order. The function contains `unchecked` block, but overflow/underflow is properly handled using smaller `uint` types and assertions.

GoodAfterTime.sol

Conditional order, which is valid between `startTime` and `endTime`. If the `data` contains `priceCheckerPayload`, there is a logic, which uses a price checker to avoid placing orders with `buyAmount` less than minimum output (expected output from price checker minus allowed slippage).

PerpetualStableSwap.sol

This conditional order swaps one token to another in 1:1 ratio. The swap order depends on the user's balance of tokens. Order always sells the token, which the user has more of.

StopLoss.sol

Stop loss conditional order, triggers when the `sellToken` price is lower than the `strike` price. Uses two Chainlink oracles (`sellTokenPriceOracle` and `buyTokenPriceOracle`). Requires both oracles to be nominated in the same currency. The Stop loss order can be buy/sell type and also, can be partially fillable.

TradeAboveThreshold.sol

Conditional order, which executes when the user's balance of `sellToken` is above the `threshold`.

ERC1271Forwarder.sol

The converter of standard `ERC1271.isValidSignature` function to `ComposableCoW.isValidSafeSignature`. Decodes the `order` data and `payload` from the `signature`, checks the hashes and passes the params to the `ComposableCoW`.

Contracts (ExtensibleFallbackHandler)

Contracts we find important in ExtensibleFallbackHandler for better understanding are described in the following section.

ExtensibleFallbackHandler.sol

`ExtensibleHFallbackHandler` inherits from `FallbackHandler`, `SignatureVerifierMuxer`, `TokenCallbacks` and `ERC165Handler` and overrides only the `_supportsInterface` function.

Base.sol

Base.sol includes `ExtensibleBase` abstract contract, which inherits from Safe `HandlerContext` and provides management of `safeMethods` mapping. It contains the mapping of function selectors to the specific handler and `internal` function `_setSafeMethod` for setting values into the mapping.

FallbackHandler.sol

`FallbackHandler` abstract contract inherits from the `ExtensibleBase` contract and implements `IFallbackHandler` interface. The `setSafeMethod` function calls the `ExtensibleBase._setSafeMethod` function implementation with the last 20 bytes of the calldata as the Safe address (`_msgSender` function). `FallbackHandler` also defines the `fallback` behavior, which validates the `calldata` and calls `handle` function on the `handler`. If the function `isStatic`, then `IStaticFallbackMethod` interface is called with `handle view` function.

MarshalLib.sol

Library for encoding and decoding handler data (`handler` address, function `selector` and boolean flag whether the function `isStatic`). Data is `bytes32`, the first byte contains `isStatic` flag and the last 20 bytes are the handler address.

SignatureVerifierMuxer.sol

Contract for advanced signature verifying, which uses multiplexing to allow different verifiers to be used in different domains. Inherits from `ExtensibleBase` and implements `ERC1271` and `ISignatureVerifierMuxer` interfaces. Contains mapping of Safe to `domainSeparators` to `ISafeSignatureVerifier` and function `setDomainVerifier` for modifying this mapping. Implementation of ERC-1271's `isValidSignature` contains logic for delegating signature validation to the respective `ISafeSignatureVerifier`.

Actors

This part describes actors of the system, their roles, and permissions.

Deployer

The deployer account has no additional privileges in the system after the deployment.

User

User role means any EOA or contract, which can interact with the protocol, using any external/public functions. In `ComposableCow` contract can change the state using `setRoot`, `setRootWithContext`, `create`, `createWithContext`, `remove` and `setSwapGuard` functions.

Cow Swap

CoW Swap is the settlement layer for Composable CoW conditional orders and from the perspective of the current scope is considered as a black box.

5.2. Trust Model

All contracts have no privileged access controls like owner/admin role. From this point of view, Composable CoW can be marked as trustless. However, users need to trust the CoW Protocol and its off-chain infrastructure in terms of transaction settlement.

M1: Oracle data validation

Medium severity issue

Impact:	High	Likelihood:	Low
Target:	StopLoss.sol	Type:	Data validation

Description

The price from the oracle is not validated. This can cause incorrect prices and order executions.

Listing 1. Excerpt from StopLoss.getTradeableOrder

```
        (, int256 latestSellPrice, , , ) =
data.sellTokenPriceOracle.latestRoundData();
        (, int256 latestBuyPrice, , , ) =
data.buyTokenPriceOracle.latestRoundData();
```

Exploit scenario

1. Bob sets the `StopLoss` order selling Token A and buying Token B. However the corresponding pairs in the oracle have different decimals. As a result, the order executes in different conditions.
2. Bob sets the `StopLoss` order and due to stale/incorrect oracle prices the order executes in different conditions.

Recommendation

Add proper validation using retrieved values from the `latestRoundData` call.

- Check decimals of the answers by using the `decimals()` function from the `IAggregatorV3Interface` interface. The answers have usually 8 or 18 decimals and should be unified for the `strike` calculation.

- Check positive price:

```
require(answer > 0, "Negative returned price");
```

- Check stale prices (if the round is not too old):

```
require(updatedAt >= block.timestamp - HEARTBEAT_TIME , "Stale price
feed"); ①
```

① Where `HEARTBEAT_TIME` is a constant that is set to a maximum desired freshness (should be higher than the oracle heartbeat time).

And if there is a possibility of using old version of OffchainAggregator called FluxAggregator, then also check the following parameters, since rounds can be calculated between more rounds.

- Check stale price:

```
require(answeredInRound >= roundId, "Price is stale");
```

- Check incomplete round:

```
require(updatedAt > 0, "Round is incomplete");
```

Solution (Revision 1.1)

The issue is partially fixed. The new code base contains proper checks for invalid and stale prices. However, the attempt to fix decimals introduced the critical severity issue C1: StopLoss arithmetic mismatches.

Solution (Revision 1.2)

Fixed, decimals are now normalized to 18.

Go back to Findings Summary

L1: Constructor data validation

Low severity issue

Impact:	Low	Likelihood:	Low
Target:	ComposableCoW.sol, ERC1271Forwarder.sol, TWAP.sol	Type:	Data validation

Description

Contracts are missing zero-address validations in the constructor. Namely `ComposableCoW`, `ERC1271Forwarder` and `TWAP`.

Listing 2. Excerpt from ComposableCoW.constructor

```
    constructor(address _settlement) {
        domainSeparator = CoWSettlement(_settlement).domainSeparator();
    }
```

Listing 3. Excerpt from TWAP.constructor

```
    constructor(ComposableCoW _composableCow) {
        composableCow = _composableCow;
    }
```

Listing 4. Excerpt from ERC1271Forwarder.constructor

```
    constructor(ComposableCoW _composableCoW) {
        composableCoW = _composableCoW;
    }
```

Exploit scenario

The contract is deployed with zero-address parameters and there is no way to set them later. Therefore the contract becomes unusable and needs to be re-deployed.

Recommendation

Add data validations to the constructors, e.g.:

```
constructor(address _settlement) {
    require(_settlement != address(0), "Zero-address _settlement");
    domainSeparator = CoWSettlement(_settlement).domainSeparator();
}
```

Solution (Revision 1.1)

Acknowledged. No zero-address checks have been introduced.

Client's response: "Given the importance of the constructor items, notably for order types that refer to critical state in the cabinet that may influence their logic, an additional step has been taken to make sure that all variables instantiated by a constructor are made public. This ensures that contracts/users interacting with these contracts can easily check all their assumptions."

Go back to Findings Summary

W1: GPv2Order data tampering

Impact:	Warning	Likelihood:	N/A
Target:	ComposableCow.sol, BaseConditionalOrder.sol	Type:	Payload manipulation

Description

In the `BaseConditionalOrder` contract, there is the `verify` function with the following parameters:

Listing 5. Excerpt from BaseConditionalOrder.constructor

```
    function verify(
        address owner,
        address sender,
        bytes32 _hash,
        bytes32 domainSeparator,
        bytes32 ctx,
        bytes calldata staticInput,
        bytes calldata offchainInput,
        GPv2Order.Data calldata
    ) external view override {
```

It is visible that GPv2Order data is omitted. For the hash verification is used:

- the hash passed as a parameter in the `isValidSafeSignature` call,
- the hash calculated with GPv2Order hash function based on the data from `getTradeableOrder` (that used static input from the passed payload).

Therefore, GPv2Order data itself doesn't figure in the `verify` function. The `isValidSafeSignature` is using it only for the `_guardCheck` call. And that means the GPv2Order data can be tampered with while it can affect the guard check but not the `verify` function result.

Recommendation

Ensure this behavior is not a problem or adjust the verification process to disallow any tampering with the GPv2Order data.

Solution (Revision 1.1)

The issue is invalidated by the following client's response: "The data tampering that has been presented (from the context of calling directly to `isValidSafeSignature`) is known. The test case presented assumes calling directly, and not from the context of a Safe with `ExtensibleFallbackHandler` as the fallback handler with `ComposableCoW` set as a `domainVerifier` for the `GPv2Settlement` domain. The reason that the `GPv2Order.Data` isn't verified subsequently is it has already been verified in the `SignatureVerifierMuxer`."

Go back to Findings Summary

W2: Revert conditions inconsistency

Impact:	Warning	Likelihood:	N/A
Target:	StopLoss.sol	Type:	Code maturity

Description

Most of the contracts use negations in revert conditions, e.g.:

Listing 6. Excerpt from GoodAfterTime.getTradeableOrder

```
        if (!(data.sellToken.balanceOf(owner) >= data.minSellBalance)) {
            revert IConditionalOrder.OrderNotValid();
        }
```

But in `StopLoss` contract, the condition is used without negation. This inconsistency decreases the readability of the code and can introduce potential human errors during future development.

Listing 7. Excerpt from StopLoss.getTradeableOrder

```
        if (latestSellPrice/latestBuyPrice > data.strike) {
            revert IConditionalOrder.OrderNotValid();
        }
```

Recommendation

Unify revert conditions syntax across the whole project.

```
if (!(latestSellPrice/latestBuyPrice <= data.strike)) {
     revert IConditionalOrder.OrderNotValid();
 }
```

Solution (Revision 1.1)

Fixed. Client's response: "Recommendation as per audit finding adopted uniformly across the project. Also searched in context of the `ComposableCoW` repository and found another instance in `BaseConditionalOrder.sol` that lacked this consistency around reversions."

Go back to Findings Summary

W3: Vulnerable MerkleProof library

Impact:	Warning	Likelihood:	N/A
Target:	ComposableCoW.sol	Type:	Dependencies

Description

The codebase is using OpenZeppelin `MerkleProof` library `v4.8.0`, which contains a vulnerability in multi-proofs. The contract is not exploitable since it is not using any multi-proofs but could be a potential problem in future development.

Recommendation

Update to `v4.9.2` or higher where is this issue patched or stay on the current version if you are not going to use multi-proofs.

Solution (Revision 1.1)

Fixed. Client's response: "Recommendation as per audit finding adopted. openzepplin library updated to v4.9.3."

Go back to Findings Summary

W4: GoodAfterTime order is missing the receiver address

Impact:	Warning	Likelihood:	N/A
Target:	GoodAfterTime.sol	Type:	Logic error

Description

The `GoodAfterTime` order is the only one whose data doesn’t contain the receiver address and there is passed a zero-address instead of it:

Listing 8. Excerpt from GoodAfterTime.getTradeableOrder

```
80          order = GPv2Order.Data(
81              data.sellToken,
82              data.buyToken,
83              address(0),
84              data.sellAmount,
85              buyAmount,
86              data.endTime.toUint32(),
87              keccak256("GoodAfterTime"),
88              0, // use zero fee for limit orders
89              GPv2Order.KIND_SELL,
90              data.allowPartialFill,
91              GPv2Order.BALANCE_ERC20,
92              GPv2Order.BALANCE_ERC20
93          );
```

L83: The receiver address

This can result in loss of funds, which will happen if no other handling prevents this order from executing. However, the severity is set to the warning because there is on-chain handling that replaces the zero-address receiver by the owner in `GPv2Settlement`.

Recommendation

Include the `receiver` into the `GoodAfterTime` order data for consistency.

Solution (Revision 1.1)

Fixed. Client's response: "To ensure consistency, `receiver` is now specified in the `Data` struct to avoid use of `address(0)` and maintain maximum flexibility for the order type."

Go back to Findings Summary

I1: Unnecessary SafeMath

Impact:	Info	Likelihood:	N/A
Target:	PerpetualStableSwap.sol	Type:	Best practices

Description

`PerpetualStableSwap` uses `SafeMath` for `uint256` and `uint8` even with Solidity `>=0.8.0 <0.9.0`. This is not necessary, since the `>=0.8` contains implicit overflow/underflow handling. All the other contracts use native Solidity math operators, which is inconsistent.

Listing 9. Excerpt from [PerpetualStableSwap.](PerpetualStableSwap)

```
    using SafeMath for uint256;
    using SafeMath for uint8;
```

Recommendation

Unify the mathematical syntax and replace `SafeMath` calls with standard math operators.

Solution (Revision 1.1)

Fixed, `SafeMath` is removed.

Go back to Findings Summary

I2: Missing cabinet cleanup

Impact:	Info	Likelihood:	N/A
Target:	ComposableCoW.sol	Type:	Best practices

Description

The function `remove` in the `ComposableCoW` contract removes the order flag from the `singleOrders` mapping but keeps data in the `cabinet` mapping. This causes data leftovers in the storage.

Listing 10. Excerpt from ComposableCoW.remove

```
    function remove(bytes32 singleOrderHash) external {
        singleOrders[msg.sender][singleOrderHash] = false;
    }
```

Recommendation

Add a data removal also for the cabinet to keep storage as clean as possible.

```
function remove(bytes32 singleOrderHash) external {
     singleOrders[msg.sender][singleOrderHash] = false;
     cabinet[msg.sender][singleOrderHash] = bytes32(0);
 }
```

Solution (Revision 1.1)

Fixed, `cabinet` cleanup is added to the `remove` function.

Go back to Findings Summary

I3: Errors in the documentation

Impact:	Info	Likelihood:	N/A
Target:	GoodAfterTime.sol, MarshalLib.sol, SignatureVerifierMuxer.sol	Type:	Documentation

Description

This informational issue summarizes inconsistencies and typos in the documentation or comments.

In the `GoodAfterTime` contract, there should be `buyAmount` instead of `sellAmount`.

Listing 11. Excerpt from GoodAfterTime.getTradeableOrder

```
            // Don't allow the order to be placed if the sellAmount is
less than the minimum out.
            if (!(buyAmount >= (_expectedOut * (MAX_BPS -
p.allowedSlippage)) / MAX_BPS)) {
                revert IConditionalOrder.OrderNotValid();
            }
```

In the `MarshalLib` library, in `decode` and `decodeWithSelector` functions should be "is 0x00" instead of "is not 0x00".

Listing 12. Excerpt from MarshalLib.decode

```
            // set isStatic to true if the left-most byte of the data is
not 0x00
            isStatic := iszero(shr(248, data))
```

Listing 13. Excerpt from MarshalLib.decodeWithSelector

```
            // set isStatic to true if the left-most byte of the data is
```

```
    not 0x00
                isStatic := iszero(shr(248, data))
```

Typo in `SignatureVerifierMuxer` documentation - ”arbitray” instead of ”arbitrary”.

Listing 14. Excerpt from SignatureVerifierMuxer.

```
    ) external view returns (bytes4 magic);
}
```

Recommendation

Fix these errors and double-check the rest of the project documentation for potential other issues to ensure 100% documentation consistency.

Solution (Revision 1.1)

Fixed.

Go back to Findings Summary

I4: TradeAboveThreshold order receiver naming

Impact:	Info	Likelihood:	N/A
Target:	TradeAboveThreshold.sol	Type:	Code maturity

Description

The `TradeAboveThreshold` order has confusing receiver naming, called `target`. It is inconsistent and with the inline documentation can be misleading about its purpose.

Listing 15. Excerpt from TradeAboveThreshold.

```
// @title A smart contract that trades whenever its balance of a certain token exceeds a target threshold
contract TradeAboveThreshold is BaseConditionalOrder {
    using GPv2Order for GPv2Order.Data;

    struct Data {
        IERC20 sellToken;
        IERC20 buyToken;
        address target;
        uint256 threshold;
    }
```

Recommendation

Rename it to `receiver` as it is for different orders.

Solution (Revision 1.1)

Fixed.

Go back to Findings Summary

I5: Inconsistent error

Impact:	Info	Likelihood:	N/A
Target:	TradeAboveThreshold.sol	Type:	Code maturity

Description

The `TradeAboveThreshold` order is the only order that uses a different error message for wrong conditions.

Listing 16. Excerpt from TradeAboveThreshold.getTradeableOrder

```
        require(balance >= data.threshold, "Not enough balance");
```

Recommendation

Replace it with the custom error that is used across all other orders.

```
if (!(balance >= data.threshold)) {
    revert IConditionalOrder.OrderNotValid();
}
```

Solution (Revision 1.1)

Fixed.

Go back to Findings Summary

I6: Commented-out code

Impact:	Info	Likelihood:	N/A
Target:	PerpetualStableSwap.sol	Type:	Code maturity

Description

There is a commented-out code in the `PerpetualStableSwap` contract.

Listing 17. Excerpt from PerpetualStableSwap.getTradeableOrder

```
54        // (IERC20 sellToken, IERC20 buyToken, uint256 sellAmount,
   uint256 buyAmount) = side(owner, data);
```

Recommendation

Dead and commented-out code should not be in the production-ready codebase.

Solution (Revision 1.1)

Fixed. Client's comment: "Some comment blocks as well found that were inconsistent with project-wide styling that were fixed."

Go back to Findings Summary

I7: Inconsistent naming

Impact:	Info	Likelihood:	N/A
Target:	Base.sol	Type:	Code maturity

Description

The file `Base.sol` contains the contract named `ExtensibleBase`, which is confusing a decreases the code clarity.

Recommendation

Rename the file to `ExtensibleBase.sol`.

Solution (Revision 1.1)

Fixed.

Go back to Findings Summary

6. Report revision 1.1

6.1. System Overview

Updates and changes we find important for fix review.

Contracts (ComposableCoW)

Changes in the ComposableCoW contracts.

StopLoss.sol

Added function `scalePrice` for normalizing price decimals.

ConditionalOrdersUtilsLib.sol

The `ConditionalOrdersUtilsLib` is a new library, which contains only the `validToBucket` function for calculating the end timestamp of the bucket. The `validToBucket` function is used in `PerpetualStableSwap`, `StopLoss`, and `TradeAboveThreshold`.

Contracts (ExtensibleFallbackHandler)

Changes in the ExtensibleFallbackHandler contracts.

ExtensibleBase.sol

`ExtensibleBase.sol` is renamed `Base.sol` from the revision 1.0.

C1: StopLoss arithmetic mismatches

Critical severity issue

Impact:	High	Likelihood:	High
Target:	StopLoss.sol	Type:	Arithmetic

Description

The `StopLoss` contract includes various arithmetic errors in the following code.

Listing 18. Excerpt from [StopLoss.getTradeableOrder]()

```
            // Normalize the basePrice and quotePrice.
            basePrice = scalePrice(basePrice, oracleSellTokenDecimals, erc20SellTokenDecimals);
            quotePrice = scalePrice(quotePrice, oracleBuyTokenDecimals, erc20BuyTokenDecimals);

            if (!(basePrice / quotePrice <= data.strike)) {
                revert IConditionalOrder.OrderNotValid(STRIKE_NOT_REACHED);
            }
```

The `scalePrice` function is used to convert oracle prices' decimals to token decimals, which is a fatal mistake.

Listing 19. Excerpt from [StopLoss.scalePrice]()

```
    function scalePrice(int256 oraclePrice, uint8 oracleDecimals, uint8 erc20Decimals) internal pure returns (int256) {
        if (oracleDecimals < erc20Decimals) {
            return oraclePrice * int256(10 ** uint256(erc20Decimals - oracleDecimals));
        } else if (oracleDecimals > erc20Decimals) {
            return oraclePrice / int256(10 ** uint256(oracleDecimals - erc20Decimals));
        }
        return oraclePrice;
    }
```

Exploit scenario

We identified 3 different possible scenarios which can happen due to bad arithmetics. All of them have a high impact and violate the expected system behavior.

Unrealized stop-loss order

This scenario has been introduced due to the new `scalePrice` function (see Listing 19) which normalizes prices from oracles to custom decimals. However, token decimals are used for this normalization. It causes miscalculations when `sellToken` and `buyToken` have different decimals, and can lead to unrealized users' stop-loss orders. Further explanation is in the following example.

Assumptions:

- Both oracle prices have 18 decimals,
- `sellToken` has 6 decimals,
- `buyToken` has 4 decimals,
- `sellToken` price (`basePrice`) = 1000e18,
- `buyToken` price (`quotePrice`) = 10e18.

Scenario:

1. Alice wants to set the stop-loss order to `basePrice = 900e18`, therefore `data.strike` has to be set to 90.
2. Price of the `sellToken` decreases to 900e18 and Alice expects the order to be fulfilled. But there is a problem due to the incorrect normalization of

decimals.

3. `sellToken` has 6 decimals, therefore the `basePrice` gets scaled to 900e6.
4. `buyToken` has 4 decimals, therefore the `quotePrice` gets scaled to 10e4.
5. These numbers go into the revert condition (see L95 in Listing 18).
6. The left side of the equation is 9000 and the right side is 90.
7. Therefore the transaction reverts and the stop-loss does not get realized at the intended price and not even at much lower prices.

Precision loss

The second exploit scenario is caused by precision loss during the division operation in the revert condition (see L95 in Listing 18).

Assumptions:

- Oracle prices, `sellToken` and `buyToken` all have 18 decimals.
- `basePrice`= 3000e18
- `quotePrice` = 1000e18

Scenario:

1. Alice sets stop-loss order to `basePrice` = 2000e18, which means `data.strike` = 2.
2. `basePrice` decreases from 3000e18 to 2900e18.
3. Division in the revert condition (see L95 in Listing 18) causes rounding and 2900e18/1000e18 = 2 instead of precise 2,9.
4. The revert condition is not met and the sell order is created at a much higher price than intended.

Tokens with similar exchange rates

The current design disallows stop-loss order to be used for tokens with similar exchange rates.

Assumptions: * Oracle prices, `sellToken` and `buyToken` all have 18 decimals. * `basePrice`= 1100e18 * `quotePrice` = 1000e18

Scenario: 1. Alice wants to set the stop-loss order to `basePrice` = 900e18, which means `data.strike` = 0,9. 2. The system does not allow to input decimal numbers as `data.strike`.

Recommendation

- Normalize the prices to 18 decimals to mitigate any decimals mismatch and improve the code clarity.

```
basePrice = scalePrice(basePrice, oracleSellTokenDecimals, 18);
quotePrice = scalePrice(quotePrice, oracleBuyTokenDecimals, 18);
```

- Denominate the `data.strike` also in 18 decimals to allow precise `data.strike` user input in case of similar token prices
- Upscale the left side of the condition also by 18 decimals.

```
if (!((basePrice * (10 ** 18)) / quotePrice <= data.strike)) {
    revert IConditionalOrder.OrderNotValid(STRIKE_NOT_REACHED);
}
```

Solution (Revision 1.2)

Fixed according to our recommendations.

Go back to Findings Summary

7. Report revision 1.2

7.1. System Overview

Updates and changes we find important for fix review.

Contracts (ComposableCoW)

Changes in ComposableCoW contracts.

StopLoss.sol

The function `scalePrice` for normalizing price decimals has been moved to `ConditionalOrdersUtilsLib`.

ConditionalOrdersUtilsLib.sol

Added function `scalePrice` for normalizing price decimals.

Appendix A: How to cite

Please cite this document as:

Ackee Blockchain, CoW Protocol: ComposableCoW & ExtensibleFallbackHandler, 4.8.2023.

Appendix B: Glossary of terms

The following terms might be used throughout the document:

Superclass/Ancestor of C

A contract that C inherits/derives from.

Subclass/Child of C

A contract that inherits/derives from C.

Syntactic contract

A Solidity contract. May have an inheritance chain, and may be deployed.

Deployed contract

An EVM account with non-zero code. If its source was written in Solidity, it was created through at least one syntactic contract. If that contract had superclasses (parents), it would be composed of multiple syntactic contracts.

Init/initialization function

A non-constructor function that serves as an initializer. Often used in upgradeable contracts.

External entrypoint

A `public` or `external` function.

Public/Publicly-accessible function/entrypoint

An `external` or `public` function that can be successfully executed by any network account.

Mutating function

A non-`view` and non-`pure` function.

Appendix C: Woke outputs

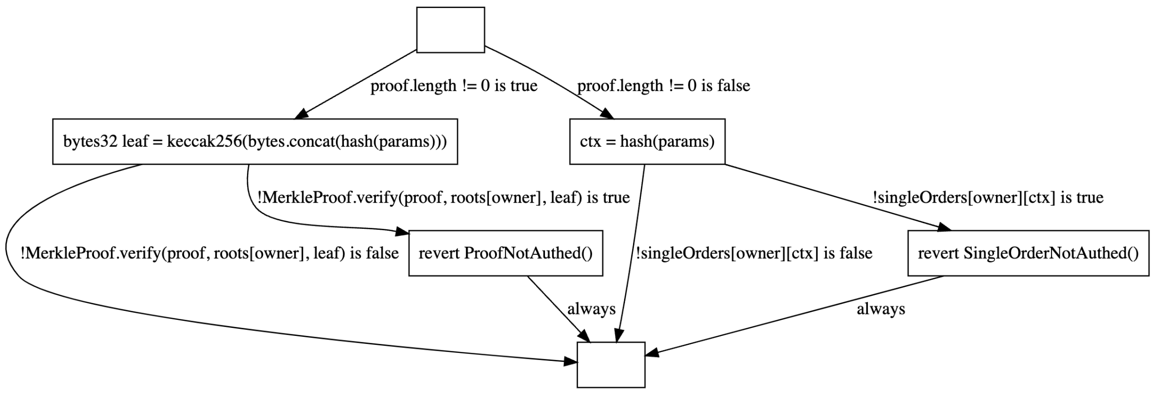
C.1. Detectors

The static analysis discovered that the `_auth` function in the `ComposableCoW` contract is missing return statements for some code paths but it is not considered as an issue in this case (see Graphs).

```
╭─ Not all code paths have return or revert statement and the return value─╮
│  288        * @param params that uniquely identify the order              │
│  289        * @param proof to assert that H(params) is in the merkle tree │
│  290        */                                                            │
│ ❱ 291      function _auth(address owner, IConditionalOrder.ConditionalOrd │
│  292            internal                                                  │
│  293            view                                                      │
│  294                                                                      │
╰─ src/ComposableCoW.sol ──────────────────────────────────────────────────╯
```

C.2. Graphs

During the audit were used control flow graphs to visualize the execution paths of the contracts. The following graph shows the `_auth` function in the `ComposableCoW` contract. The `ctx` return parameter is returned only for single orders otherwise the initial value (zero) is returned.


proof.length != 0 is true
proof.length != 0 is false
bytes32 leaf = keccak256(bytes.concat(hash(params)))
ctx = hash(params)
!MerkleProof.verify(proof, roots[owner], leaf) is true
!singleOrders[owner][ctx] is true
!MerkleProof.verify(proof, roots[owner], leaf) is false
revert ProofNotAuthed()
!singleOrders[owner][ctx] is false
revert SingleOrderNotAuthed()
always
always

C.3. Tests

The following part of test shows an example of tampering the data on GPv2Order data (see the issue for more information).

```
def validate_single_order(params, gpv2order_data, safe, ccow):
    # get payload
    payload = ComposableCoW.PayloadStruct(
        proof=[],
        params=params,
        offchainInput=Abi.encode(['uint256'], [10**18])
    )
    payload_values = [(payload.proof, (params.handler, params.salt,
params.staticInput), payload.offchainInput)]
    payload_types = ["(bytes32[],(address,bytes32,bytes),bytes)"]
    payload_encoded = Abi.encode(payload_types, payload_values)

    # it is possible to tamper the order data
    gpv2order_values = [(gpv2order_data.sellToken, gpv2order_data.buyToken,
gpv2order_data.receiver, gpv2order_data.sellAmount,
gpv2order_data.buyAmount + 10**30, gpv2order_data.validTo,
gpv2order_data.appData, gpv2order_data.feeAmount, gpv2order_data.kind,
gpv2order_data.partiallyFillable, gpv2order_data.sellTokenBalance,
gpv2order_data.buyTokenBalance)]
    gvp2order_types =
["(address,address,address,uint256,uint256,uint32,bytes32,uint256,bytes32,b
ool,bytes32,bytes32)"]
    gpv2order_encoded = Abi.encode(gvp2order_types, gpv2order_values)

    # do a proper hash
    gpv2hash = GPv2Hash.deploy()
    gpv2order_hash = gpv2hash.hash(gpv2order_data, ccow.domainSeparator())

    tx = ccow.isValidSafeSignature(safe, Address.ZERO, gpv2order_hash,
ccow.domainSeparator(), b'', gpv2order_encoded, payload_encoded,
request_type="tx")
    assert tx.return_value == ERC1271.isValidSignature.selector
```

Thank You

Ackee Blockchain a.s.

Prague, Czech Republic

hello@ackeeblockchain.com

https://twitter.com/AckeeBlockchain