CoW-AMM Audit

This document presents the finding of a smart contract audit conducted by Côme du Crest for Gnosis.

Scope

The scope includes all contracts within CoW-amm as of commit 145384c. The audit was initially started as of commit d673529 and revealed three informational issues. Further work from the development team added a security feature to prevent multiple AMM orders to be executed in a single CoW protocol batch. The `BalancerWeightedPoolPriceOracle` contract was also added which lead to the final commit for this audit.

The project is tightly linked to composable-cow, safe-contracts/SignatureVerifierMuxer.sol, and GPv2.

Context

The AMM takes the form of a Safe contract that uses the `SignatureVerifierMuxer` to verify signatures on GPv2 orders. The Muxer calls `ComposableCow.isValidSafeSignature()` which in turns calls `ConstantProduct.verify()` where the logic of the AMM resides.

Status

The audit has been sent to the developer team.

Issues

▼ [Low] DOS orders on empty commitments by sending tokens

Summary

If CoW Protocol solvers do not commit on a hash and the commitment in the AMM is equal to `bytes32(0)`, the complete order parameters will be checked against the AMM's returned order via `_getTradeableOrder()` which relies on the internal balance of the AMM for computing the order. Anyone can make the signature verification fail by sending a few wei of tokens to the AMM in a front-running transaction.

Vulnerability Detail

The signature verification process starts with verifying :

```
function _verify(address owner, bytes32 orderHash, bytes calldata staticInput, GPv2Order.D
        internal
        view
    {
        requireMatchingCommitment(owner, orderHash, staticInput, order);
        ...
    }
```

`requireMatchingCommitment()` checks that either the committed hash matches the order hash or if the committed hash is `bytes32(0)` then it checks that the order matches what is returned by `_getTradeableOrder()`:

```
    function requireMatchingCommitment(
        address owner,
        bytes32 orderHash,
        bytes calldata staticInput,
        GPv2Order.Data calldata order
    ) internal view {
        bytes32 committedOrderHash = commitment[owner];
        if (orderHash != committedOrderHash) {
            if (committedOrderHash != EMPTY_COMMITMENT) {
                revert OrderDoesNotMatchCommitmentHash();
            }
            GPv2Order.Data memory computedOrder = _getTradeableOrder(owner, staticInput);
```

```
            if (!matchFreeOrderParams(order, computedOrder)) {
                revert OrderDoesNotMatchDefaultTradeableOrder();
            }
        }
    }

    function matchFreeOrderParams(GPv2Order.Data calldata lhs, GPv2Order.Data memory rhs)
        internal
        pure
        returns (bool)
    {
        bool sameSellToken = lhs.sellToken == rhs.sellToken;
        bool sameBuyToken = lhs.buyToken == rhs.buyToken;
        bool sameSellAmount = lhs.sellAmount == rhs.sellAmount;
        bool sameBuyAmount = lhs.buyAmount == rhs.buyAmount;
        bool sameValidTo = lhs.validTo == rhs.validTo;
        bool sameKind = lhs.kind == rhs.kind;
        bool samePartiallyFillable = lhs.partiallyFillable == rhs.partiallyFillable;

        // The following parameters are untested:
        // - receiver
        // - appData
        // - feeAmount
        // - sellTokenBalance
        // - buyTokenBalance

        return sameSellToken && sameBuyToken && sameSellAmount && sameBuyAmount && sameVal
            && samePartiallyFillable;
    }
```

However the exact values for sell and buy amounts can easily be influenced by sending a few wei of the sell or buy tokens to the AMM which will result in the verification failing with `OrderDoesNotMatchDefaultTradeableOrder`.

Impact

Anyone can DOS an order dependant on an empty commitment by sending a few wei of tokens to the AMM. If CoW protocol solvers rely on this behaviour, they can invalidate each other's batches with front-running transactions.

Code Snippets

https://github.com/cowprotocol/cow-amm/blob/145384cca64c44dd36bf68ae874f172b302eba10/src/ConstantProduct.sol#L245-L249

https://github.com/cowprotocol/cow-amm/blob/145384cca64c44dd36bf68ae874f172b302eba10/src/ConstantProduct.sol#L344-L392

https://github.com/cowprotocol/cow-amm/blob/145384cca64c44dd36bf68ae874f172b302eba10/src/ConstantProduct.sol#L174-L180

Recommendation

Solvers should not rely on the empty commitment behaviour and should commit to the order they execute at the beginning of a batch. As far as I understood from the protocol team, this is the behaviour they expect solvers to apply in the long run. The empty commitment feature is here to help solvers discover the use of the AMM before they adapt to it.

If that is indeed the case, no fixing is required.

▼ [Low] Solvers can DOS orders by committing to different hash

Summary

CoW Protocol solvers that rely on a specific committed to hash before a batch execution to get their order validated can be denied by other solvers committing to any other hash in a front-running transaction. This is specifically the case for solvers relying on the empty commitment behaviour.

Vulnerability Detail

The signature verification process starts with verifying :

```
function _verify(address owner, bytes32 orderHash, bytes calldata staticInput, GPv2Order.D
        internal
        view
    {
        requireMatchingCommitment(owner, orderHash, staticInput, order);
        ...
    }
```

`requireMatchingCommitment()` checks that either the committed hash matches the order hash or if the committed hash is `bytes32(0)` then it checks that the order matches what is returned by `_getTradeableOrder()`:

```
    function requireMatchingCommitment(
        address owner,
        bytes32 orderHash,
        bytes calldata staticInput,
        GPv2Order.Data calldata order
    ) internal view {
        bytes32 committedOrderHash = commitment[owner];
        if (orderHash != committedOrderHash) {
            if (committedOrderHash != EMPTY_COMMITMENT) {
                revert OrderDoesNotMatchCommitmentHash();
            }
            GPv2Order.Data memory computedOrder = _getTradeableOrder(owner, staticInput);
            if (!matchFreeOrderParams(order, computedOrder)) {
                revert OrderDoesNotMatchDefaultTradeableOrder();
            }
        }
    }
```

If any adversary CoW Protocol solvers commits to a hash before submission of a batch relying on an empty commitment or specific hash, then the signature will not be verified and the order invalid.

Impact

Any solver can front-run a batch execution relying on a committed hash and not committing in the same transaction. Due to the order hash not being cleared after signature verification (because signature verification legitimately uses a view function), any solver committing to any value will permanently deny other solvers relying on the empty commitment feature until they re-apply a `bytes32(0)` commitment.

Code Snippets

https://github.com/cowprotocol/cow-amm/blob/145384cca64c44dd36bf68ae874f172b302eba10/src/ConstantProduct.sol#L245-L249

https://github.com/cowprotocol/cow-amm/blob/145384cca64c44dd36bf68ae874f172b302eba10/src/ConstantProduct.sol#L344-L360

Recommendation

The protocol team already expressed interests to use `TSTORE` and `TLOAD` in the future to alleviate the issue. In the meantime, the committed hash could be bound to each independent solver by using a mapping `commitment[owner][solver] => hash` instead of `commitment[owner] => hash`. This would require to find out the solver's address in the signature verification process either via additional static input data, via the use of `tx.origin`, or by providing it in `payload.offchainInput`.

The protocol team may acknowledge this issue.

▼ [Low] Balancer's oracle vulnerable to manipulation

Summary

Balancer's functions `vault.getPoolTokens(poolId)` is vulnerable to a read-only re-entrancy attack that allows manipulating the returned values of token balances. This means that the price returned by the Balancer oracle can be arbitrarily manipulated.

See: Balancer's forum and provided library solution.

Vulnerability Detail

The `getPrice()` function of the `BalancerWeightedPoolPriceOracle` uses `vault.getPoolTokens(poolId)` to get the balances of each token in the pool and relies on these balances to compute the price:

```
    function getPrice(address token0, address token1, bytes calldata data) external view r
        IWeightedPool pool;
        IERC20[] memory tokens;
        uint256[] memory balances;
        uint256[] memory weights;
        {
            // Note: function calls in this scope aren't affected by the paused
            // state
            bytes32 poolId = abi.decode(data, (Data)).poolId;
            // If the pool isn't registered, then the next call reverts
            try vault.getPool(poolId) returns (address a, IVault.PoolSpecialization) {
                pool = IWeightedPool(a);
            } catch (bytes memory) {
                revert IConditionalOrder.OrderNotValid("invalid pool id");
            }

            (tokens, balances,) = vault.getPoolTokens(poolId);

            ...
        }
        ...

        uint256 priceNumerator = balanceToken0 * weightToken1;
        uint256 priceDenominator = balanceToken1 * weightToken0;

        ...
    }
```

The value of balances can be manipulated via read-only re-entrancy impacting the computed price.

Impact

The manipulated price impacts the order computed from `getTradeableOrder()` by the AMM. This in turns impact signature verification that relies on the empty commitment feature for matching the executed order to the computed order by the AMM.

However, it does not impact the logic of the AMM for verifying that it should accept an order based on its underlying curve as defined in `_verify()`.

No loss of funds should be caused by this issue.

Code Snippets

https://github.com/cowprotocol/cow-amm/blob/145384cca64c44dd36bf68ae874f172b302eba10/src/oracles/BalancerWeightedPoolPriceOracle.sol#L53-L117

Recommendation

It is not possible to use Balancer's recommanded library to alleviate the read-only re-entrancy because it is not a view function. Ultimately, I understand the oracle is here to help CoW Protocol solvers to produce valid orders rather than enforce order validity.

Since no loss of funds is at stake, the protocol team may acknowledge the issue.

▼ [Info] Unenforced minimum trade amount

Summary

The code to generate the trade order attempts to enforce a minimum trade amount, however this amount is not enforced during signature verification and orders below minimum trade amount will still be accepted. Additionally, orders can be partially fillable in which case the minimum trade amount will not be respected anyway.

Vulnerability Detail

The function to get the GPv2 order from the AMM reverts when trade amount in token0 is below a minimum trade amount:

```
    function _getTradeableOrder(address owner, bytes calldata staticInput)
        internal
        view
        returns (GPv2Order.Data memory order)
    {
                ...

                if (tradedAmountToken0 < data.minTradedToken0) {
            revert IWatchtowerCustomErrors.PollTryAtEpoch(
                Utils.validToBucket(MAX_ORDER_DURATION) + 1, "traded amount too small"
            );
          }

                ...
        }
```

The value of `data.minTradedToken0` is user-provided anyway and there is no verification on it (unless called through `ComposableCoW.getTradeableOrderWithSignature()`).

No rules enforce this is the case during order verification.

Impact

The function to get the tradeable order is not meant to be called on-chain and is just a helper function, I understand there is a trust assumption that most of the time the orders executed from the AMM will be computed from `getTradeableOrder()`. However, developers should be aware that this value is not strictly enforced and any forged order could execute below this.

Code Snippets

https://github.com/cowprotocol/cow-amm/blob/d673529e3d5df028aca0eab23db90b5c8790e0ff/src/ConstantProduct.sol#L131-L135

Recommendation

Acknowledge that this is intended behaviour.

▼ [Info] Unenforced order expiry time

Summary

The `_verify()` function attempts to enforce that order's validity in terms of max duration but the `order.validTo` field is user-provided and does not prevent old orders from executing.

Vulnerability Detail

The `_verify()` function reverts when `order.validTo` is too far in the future:

```
// We add a maximum duration to avoid spamming the orderbook and force
// an order refresh if the order is old.
if (order.validTo > block.timestamp + MAX_ORDER_DURATION) {
    revert IConditionalOrder.OrderNotValid("validity too far in the future");
}
```

However, the `order.validTo` can be a forged value by the order provider. If not a forged value, then this condition can never be met as `_getTradeableOrder()` computes the `order.validTo` value as `Utils.validToBucket(MAX_ORDER_DURATION)`.

Impact

This check does not seem to force a refresh of old orders or prevent `order.validTo` manipulation.

Code Snippets

https://github.com/cowprotocol/cow-amm/blob/d673529e3d5df028aca0eab23db90b5c8790e0ff/src/ConstantProduct.sol#L202-L206

https://github.com/cowprotocol/cow-amm/blob/d673529e3d5df028aca0eab23db90b5c8790e0ff/src/ConstantProduct.sol#L143

Recommendation

Delete the check.

▼ [Info] Incorrect rounding in _getTradeableOrder()

Summary

The documentation states that they want to round down the sell amount and round up the buy amount, which makes sens to guarantee the interests of the AMM. However, the rounding of the buyAmount may result in a rounding down from the exact calculation.

Vulnerability Detail

The `sellAmount` is properly rounded down from the calculation:

```
sellAmount = selfReserve0 / 2 - Math.ceilDiv(selfReserve1TimesPriceNumerator, 2 * priceDen
```

However, the `buyAmount` calculation re-uses the `sellAmount` value which has been rounded down and rounds up the final result:

```
buyAmount = Math.mulDiv(
    sellAmount,
    selfReserve1TimesPriceNumerator + (priceDenominator * sellAmount),
    priceNumerator * selfReserve0,
    Math.Rounding.Up
);
```

There is no guarantee that when we express `buyAmount` in terms of reserves and oracle price the returned value is a rounded up value of the calculation since an intermediate rounded down value is used.

Impact

If we consider that the buyAmount shall be expressed in terms of buyAmount and not reserve and oracle price, then the rounding issue is irrelevant. Otherwise, since the calculation should not give the best available price for the AMM, the impact should be insignificant.

Code Snippets

https://github.com/cowprotocol/cow-amm/blob/d673529e3d5df028aca0eab23db90b5c8790e0ff/src/ConstantProduct.sol#L107-L129

Recommendation

Acknowledge the issue.

Conclusion

No important security issues have been found during the audit.