Security Assessment
Final Report

Balancer V1 Changes for CoW AMM

JUNE 2024

Prepared for BALANCER

certora Balancer

Table of content

Project Summary

Project Scope

Project Name	Repository (link)	Latest Commit Hash	Platform
Balancer	https://github.com/defi-wonderland/balancer-v1-amm/tree/dev/src/contracts	99f6910	EVM

Project Overview

This document describes the manual code review findings of the Balancer V1 project. The work was undertaken from 20/6/2024 to 2/7/2024.

The following contract list is included in our scope:

```
contracts/BCoWConst.sol
contracts/BCoWFactory.sol
contracts/BCoWPool.sol
contracts/BConst.sol
contracts/BFactory.sol
contracts/BMath.sol
contracts/BNum.sol
contracts/BPool.sol
contracts/BToken.sol
```

The team performed a manual audit of all the Solidity contracts, but the main focus was on the changes that were made from the previous version. During the manual audit, the Certora team discovered issues in the Solidity contracts code, as listed on the following page.

Protocol Overview

Balancer CoW AMM is an automated portfolio manager, liquidity provider, and price sensor, that allows swaps to be executed via the CoW Protocol.

Findings Summary

The table below summarizes the findings of the review, including type and severity details.

Severity	Discovered	Confirmed	Fixed
Critical	-	-	-
High	-	-	-
Medium	-	-	-
Low	-	-	-
Informational	5		
Gas	6		
Total	11		

Severity Matrix

Impact	High	Medium	High	Critical
	Medium	Low	Medium	High
	Low	Low	Low	Medium
		Low	Medium	High
		Likelihood		

Detailed Updates Review

As mentioned before, the main focus of this review was on the changes from the previous version of the Balancer V1 protocol. The following is a list of some of those changes, our analysis of them, and a description of some of the things we did:

1. Using a new version of the solidity compiler- As of version 8.0, the solidity compiler adds a built in underflow and overflow checks. While it should not cause any security-related problems, it might increase gas cost where underflow/overflow checks are not required. We searched for places in the code in which adding an unchecked block might be possible and might help reduce gas costs.

2. Adding unchecked blocks- As many of the functions in the BNum.sol already contained manual overflow/underflow checks, the overflow/underflow checks automatically added by the solidity compiler are redundant. Therefore, it made sense to have all these operations inside an unchecked block. We went over these changes and determined that they appear to be correct and that there is in fact a manual overflow/underflow check. Furthermore, any possible issue as a result of an overflow/underflow must have already been present in the previous version of the code.

3. Using the token balance directly instead of internal accounting- While some attacks might take advantage of the fact it is possible to “gift” the protocol tokens in order to somehow affect tokens calculations in favor of the attacker, we determined that this change is reasonable as any such attack would have been possible in the previous version of the code as well because the attacker would simply be able to call the gulp() function to force _records[token].balance to match the actual balance of the protocol. We also considered some possible attacks that “gift” the protocol tokens and determined that they are insignificant. For example, it is possible to front run a join request and cause that transaction to revert, or to front run a swap to worsen the price from the perspective of the swapper, but those possibilities are a part of the system’s design, and transactions will only execute if they satisfy the boundaries set by the user (such as min/max amount or max price).

4. Replacing require(condition) statements with if(!condition) revert()- We went over the changes and made sure that the translations of the logical conditions to the new form appear to be correct. For example, a relatively complex condition such as require (a==0 || c0 / a == b) was correctly changed to if (a != 0 && c0 / a != b) revert().

5. Changing the way bind() and unbind() work- We saw that no exitFee is taken when calling unbind(), and that rebind() was deprecated in favor of unbind() and bind(). Some of the logic that was previously performed in the rebind() function was moved to the bind(), and the bind() function now has the lock modifier instead of the deprecated rebind(). We concluded that this change appears to be fine.

6. Deprecating unused functions- Some functions were no longer needed and thus deleted. We determined that this change appears to be fine.

7. Adding new contracts- Some new contracts were added. Namely BCoWConst.sol, BCoWFactory.sol (which inherits BFactory.sol logic) and BCoWPool.sol (which inherits BPool.sol logic). We went over these contracts, and determined that the code seems to match the description of these contracts. Specifically in the BCoWPool.sol contract, we saw that the isValidSignature() function validates that the order's hash matches the commitment and then calls the verify() function. We also noted that as described, the reentrancy lock is saved in the same transient storage slot as the commitment, but once it is set it cannot be reset in the same transaction and thus the commit() function cannot be used by the SOLUTION_SETTLER to override the reentrancy lock. We also went over the implementation of the _afterFinalize() function, and saw that it grants approval to the VAULT_RELAYER for all tokens for any amount, and that it calls the _factory contract in order for the factory to emit the event, and does it itself if it reverts.

Detailed Findings

Informational Severity Issues

I-01. Tokens that are not compliant with the ERC20 standard cannot be supported.

Description:
Some tokens do not implement the ERC20 standard properly but are still accepted by most code that accepts ERC20 tokens. For example Tether (USDT)'s transfer() and transferFrom() functions do not return booleans as the specification requires, and instead have no return value. When these sorts of tokens are cast to IERC20, their function signatures do not match and therefore the calls made, revert.

Recommendation:
Consider using OpenZeppelin's SafeERC20's safeTransfer()/safeTransferFrom() instead.

Customer's response:

Fix Review:

Fixed in a6d0e0e.

I-02. Unused constant.

Description:
The EMPTY_COMMITMENT constant was defined but never used.

Recommendation:
Consider deleting this constant or adding the logic that is supposed to use it.

Customer's response:

Fix Review:

Fixed in a6d0e0e.

I-03. Missing checks for address(0) when assigning values to address state variables.

Description:
The accidental calling of the setController() (in Bpool.sol) and the setBLabs() (in BFactory.sol) functions or the constructor of the BCowPool.sol or BCowFactory.sol contracts with the 0 address as a parameter may be irreversible.

Recommendation:
Consider adding a check that the passed parameter is non-zero.

Customer's response:

Fix Review:

Partially fixed in 9f8015d. The checks were only added to the setController() and setBLabs() functions.

I-04. Incorrect revert messages.

Description:
The revert messages in the following places are incorrect:

BPool.sol:274:

```
if (spotPriceAfter > maxPrice) {
  revert BPool_SpotPriceAfterBelowMaxPrice();
}
```

BPool.sol:336:

```
if (spotPriceAfter > maxPrice) {
  revert BPool_SpotPriceAfterBelowMaxPrice();
}
```

Recommendation:
Replace BPool_SpotPriceAfterBelowMaxPrice(); with BPool_SpotPriceAfterAboveMaxPrice();

Customer's response:

Fix Review:

I-05. Max value not allowed.

Description:
The verify() function does not support orders which are valid exactly up until the maximal allowed time.

BCoWPool.sol:110:

```
if (order.validTo >= block.timestamp + MAX_ORDER_DURATION) {
  revert BCoWPool_OrderValidityTooLong();
}
```

Recommendation:
Allow order.validTo to be exactly equal to block.timestamp + MAX_ORDER_DURATION by changing >= to >.

Customer's response:

Fix Review:

Gas optimization

G-01. State variables only set in the constructor could be declared immutable.

Description:
The _factory variable (in Bpool.sol) is only set in the constructor and thus could potentially be declared immutable.
This would avoid the expensive storage-writing operation in the constructor (around 20000 gas per variable) and replace the expensive storage-reading operations (around 2100 gas per reading) to a less expensive value reading (3 gas).

Recommendation:
Declare the _factory variable as immutable.

Customer's response:

Fix Review:

Fixed in 79b8d58.

G-02. Array length could be cached outside of the loop.

Description:

In the following places:

BPool.sol:174: for (uint256 i = 0; i < _tokens.*length*; i++) {

BPool.sol:209: for (uint256 i = 0; i < _tokens.*length*; i++) {

_tokens.*length* is checked on each iteration of the loop.
For a storage array, this is an extra sload operation (100 additional extra gas for each iteration except for the first).

Recommendation:

Cache the length of the array outside of the loop.

Customer's response:

Fix Review:

G-03. Arithmetics operations that can't underflow/overflow could be unchecked.

Description:
Solidity version 0.8+ adds overflow and underflow checks on unsigned integers.
In the following cases, these checks might be redundant:

BPool.sol:153: uint256 last = _tokens.*length* - 1;

BToken.sol:35: _approve(msg.sender, dst, oldValue - amt);

Recommendation:
Consider wrapping with an unchecked block where it's certain that there cannot be an underflow.

Customer's response:

Fix Review:

G-04. ++i costs less gas compared to i++.

Description:

Although both operators increment i and are the same in the context of for loops, those operators have different gas costs.

In previous versions of the solidity compiler, the following was true:

- i+=1 is the most expensive form.
- i++ costs 6 gas less than i+=1.
- ++i costs 5 gas less than i++ (11 gas less than i += 1).

This might no longer be the case in the newest versions of the solidity compiler.

Recommendation:
Check whether replacing i++ with ++i inside for loops saves gas.

Customer's response:

Fix Review:

G-05. Increments inside for-loops could be unchecked.

Description:

Incrementing uint256 variables that were initialized to 0 could not possibly cause an overflow and thus could be unchecked.
This might be relevant in the following places:

BNum.sol:179: for (uint256 i = 1; term >= precision; i++) {

BPool.sol:174: for (uint256 i = 0; i < _tokens.*length*; i++) {

BPool.sol:209: for (uint256 i = 0; i < _tokens.*length*; i++) {

Recommendation:
Nothing, as this is already automatically done by the newest version of the solidity compiler.

Customer's response:

Fix Review:

G-06. Use shift right instead of division if possible.

Description:
While the DIV opcode uses 5 gas, the SHR opcode only uses 3 gas. Furthermore, beware that Solidity's division operation also includes a division-by-0 prevention which is bypassed using shifting. Eventually, overflow checks are never performed for shift operations as they are done for arithmetic operations. Instead, the result is always truncated, so the calculation can be unchecked in Solidity version 0.8+.
This might be relevant in following places:

BConst.sol:44: uint256 public constant MAX_IN_RATIO = BONE / 2;

BNum.sol:97: uint256 c1 = c0 + (BONE / 2);

BNum.sol:115: uint256 c1 = c0 + (b / 2);

Recommendation:
Use >> 1 instead of / 2.

Customer's response:

Fix Review:

Fixed in 79b8d58.

Disclaimer

Even though we hope this information is helpful, we provide no warranty of any kind, explicit or implied. The contents of this report should not be construed as a complete guarantee that the contract is secure in all dimensions. In no event shall Certora or any of its employees be liable for any claim, damages, or other liability, whether in an action of contract, tort, or otherwise, arising from, out of, or in connection with the results reported here.

About Certora

Certora is a Web3 security company that provides industry-leading formal verification tools and smart contract audits. Certora's flagship security product, Certora Prover, is a unique SaaS product that automatically locates even the most rare & hard-to-find bugs on your smart contracts or mathematically proves their absence. The Certora Prover plugs into your standard deployment pipeline. It is helpful for smart contract developers and security researchers during auditing and bug bounties.

Certora also provides services such as auditing, formal verification projects, and incident response.