PUBLIC

Code Assessment

of the Carbon Smart Contracts

April 10, 2023

Produced for

by

Contents

1 Executive Summary

Dear Bancor Team,

Thank you for trusting us to help Bancor with this security audit. Our executive summary provides an overview of subjects covered in our audit of the latest reviewed contracts of Carbon according to Scope to support you in forming an opinion on their security risks.

Bancor implements an AMM with asymmetric liquidity on which each user's liquidity position is represented by two independent curves for buying and selling a token respectively. Trades can be matched against these positions using Bancor's decentralized SDK, or in whichever manner the user desires.

As has been communicated by the Bancor team at the start of the audit, a precision error could lead to some losses for customers due to unexpected pricing: Price Precision Very Low for Some Tokens. This issue has been mitigated by an encoding format that increases the amount of bits that can effectively be used.

The most critical subjects covered in our audit are functional correctness, precision of arithmetic operations and front-running. Security regarding all the aforementioned subjects is high.

In summary, we find that the codebase provides a high level of security.

It is important to note that security audits are time-boxed and cannot uncover all vulnerabilities. They complement but don't replace other vital measures to secure a project.

The following sections will give an overview of the system, our methodology, the issues uncovered and how they have been addressed. We are happy to receive questions and feedback to improve our service.

Sincerely yours,

ChainSecurity

1.1 Overview of the Findings

Below we provide a brief numerical overview of the findings and how they have been addressed.

Critical-Severity Findings	0
High-Severity Findings	2
• Code Corrected	2
Medium-Severity Findings	0
Low-Severity Findings	5
• Code Corrected	4
• Risk Accepted	1

2 Assessment Overview

In this section, we briefly describe the overall structure and scope of the engagement, including the code commit which is referenced throughout this report.

2.1 Scope

The assessment was performed on the source code files inside the Carbon repository based on the documentation files. The table below indicates the code versions relevant to this report and when they were received.

carbon-contracts

V	Date	Commit Hash	Note
1	5 February 2023	ca112969b9a5ddc7be71c867492c9b31777ddb0a	Initial Version
2	3 April 2023	fad7097d9d443dfa8b645e72c4102640168c8749	Second Version
3	10 April 2023	e0a96ceb0bce606a07b3c65cc8e84b21b990654a	Third Version

carbon-sdk

V	Date	Commit Hash	Note
1	6 February 2023	862fde1989e7f9b32138c749171c10e21c8473bf	Initial Version
2	4 April 2023	e23dab02a9e83fae82157e032d4766fe513af108	Second Version

From the `carbon-contracts` repository, all files in the `contracts` folder, with exception of the `helpers` and `fees` folders, are in scope of this audit.

For the solidity smart contracts, the compiler version `0.8.19` was chosen.

2.1.1 Excluded from scope

The `carbon-sdk` repository was also provided for context purposes, but is out of scope.

2.2 System Overview

This system overview describes the initially received version (Version 1) of the contracts as defined in the Assessment Overview.

Furthermore, in the findings section, we have added a version icon to each of the findings to increase the readability of the report.

Bancor offers an Automated Market Making protocol with concentrated liquidity. It features an invariant function that allows users to determine custom price intervals and price acceleration. Furthermore, it allows for the creation of concentrated liquidity that is traded in only one direction. Liquidity set in a specific price range (or a singular price) can be traded when the price, e.g., increases, but can not be traded back in the other direction. Each user can create such positions by publishing two orders on-chain: One order in buy direction and one order in sell direction. Because such orders can be published with 0 liquidity and completely custom pricing parameters, users are effectively able to create one-sided orders.

Trades happen on one or multiple of these so-called Strategies. A trader has to call the contract with a given set of Strategies. The set can be determined using Bancor's decentralized SDK, which heuristically calculates the Strategies for best price execution, or in whichever manner the user desires. In each Strategy, a trade removes liquidity from a one-sided order and adds liquidity to the opposite order. After that, anyone could theoretically execute a trade on this opposite order. The orders can therefore be only set in a "buy low, sell high" fashion so that it becomes economically pointless that such orders are executed: A "stop loss" order, for example, would be instantly executed by another user because it would sell the given asset below market price.

The SDK determines the Strategies to trade by simulating a trade on each Strategy with the maximum available liquidity and ordering them based on effective rate. This heuristic approach is necessary due to the complexity introduced by each Strategy having a different curve. Best execution can only be achieved by intensive computations that are not feasible for every-day usage (as Bancor claims).

2.2.1 Invariant function

The invariant function used in Carbon is a generalization of constant-product and constant-sum invariant functions. It features three parameters that allow the curve to cover a single price (equivalent to constant-sum with `A=0`), a fixed range of prices or (almost) infinite price space (~equivalent to constant-product, bound by the maximum value of `A`):

- `A`: The curvature of the curve, defined by the difference of the square roots of the maximum and minimum price of the given price range.
- `B`: The "slope" of the curve, defined by the square root of the minimum price of the given price range.
- `z`: The capacity of the curve, defined by the interception point of the curve on the y-axis.

All parameters have positive values. Negative values allow for even greater flexibility which is not desirable for Carbon though. Furthermore, other incarnations of the invariant with different parameters have been proposed by Bancor but are outside of the scope of this work.

2.2.2 Asymmetric liquidity

Liquidity in Carbon is provided uni-directionally. This means, for a given pair of tokens, a liquidity provider can create two different curves for buying and selling one of these tokens for the other. These curves can have different price ranges and different liquidity. Trades on one of these curves result in liquidity shifting to the opposite curve and vice versa: The trader receives some tokens that are removed from the curve's liquidity in exchange for the counterpart token which is added to the liquidity of the opposite curve. These counterpart tokens are then tradeable again, but only in the price range set by the curve they were added to.

2.2.3 Fees

In opposite to other AMM projects, Carbon does not accrue yield for liquidity providers. Thus, liquidity provision can be seen as a more traditional order placement. Since each Strategy has 2 different curves for buying and selling and Strategies can also be created with liquidity on only one side, Impermanent Loss is not inherent to Carbon.

Carbon only accumulates fees for the Bancor DAO. This is set as a percentage of the traded tokens. Users trading by "source amount" are paying fees in the target token, while users trading by "target amount" are paying fees in the source token.

2.2.4 Contracts

CarbonController

`CarbonController` is the main contract that is used for liquidity provision and trading. Users can call `createStrategy` to create a new strategy with custom parameters. They can also call `deleteStrategy` to remove the strategy and withdraw all provided tokens or call `updateStrategy` to

change the order parameters, which can potentially lead to token transfers between the contract and the user, depending on whether the two orders' liquidity has been increased or decreased. `tradeBySourceAmount` and `tradeByTargetAmount` can be used to trade a specific pair of tokens on one or multiple Strategies but in only one direction. The given amount for each provided Strategy is executed, no matter the divergence between the Strategy's current marginal price and the actual market price of the asset.

Voucher

Strategies are assigned to transferrable NFTs. `Voucher` is the ERC-721 contract that handles these NFTs.

MasterVault

The `MasterVault` is a separate contract that stores all the liquidity tokens of `CarbonController`.

2.2.5 Roles & Trust Model

The `CarbonController` contract uses an Access Control List and defines two different roles:

- `ROLE_ADMIN` is assigned to the deployer and allows for the assignment of other roles and the change of trading fees.
- `ROLE_EMERGENCY_STOPPER` allows the contract to pause most state-changing functions.

The `Voucher` contract has a single, transferrable owner. The owner can change the associated URI and the `CarbonController` address, which is the address allowed to `mint` and `burn` tokens.

The `MasterVault` uses an Access Control List and defines two different roles:

- `ROLE_ADMIN` is assigned to the deployer and allows for the assignment of other roles.
- `ROLE_ASSET_MANAGER` allows for the withdrawal of funds from the vault. It has to be assigned to the associated `CarbonController` by the admin.

`CarbonController` and `MasterVault` are set up using a customized version of the Transparent Upgradeable Proxy pattern. The proxies feature an immutable admin address that points to a `ProxyAdmin` contract. This contract is able to upgrade the implementations for both proxies.

Bancor maintains a DAO contract which is believed to be used to deploy the mentioned contracts and therefore be in control of the admin addresses.

Version 2 changes:

- The `MasterVault` smart contract does not exist anymore and liquidity is instead kept directly in the `CarbonController`.
- A new `ROLE_FEES_MANAGER` role on the `CarbonController` allows the withdrawal of accumulated fees.
- Order's rates are now encoded differently and are more precise.
- Pools have been renamed to pairs.
- Strategy IDs are now composed of both an ID for the strategy and an ID for the corresponding pair.

3 Limitations and use of report

Security assessments cannot uncover all existing vulnerabilities; even an assessment in which no vulnerabilities are found is not a guarantee of a secure system. However, code assessments enable the discovery of vulnerabilities that were overlooked during development and areas where additional security measures are necessary. In most cases, applications are either fully protected against a certain type of attack, or they are completely unprotected against it. Some of the issues may affect the entire application, while some lack protection only in certain areas. This is why we carry out a source code assessment aimed at determining all locations that need to be fixed. Within the customer-determined time frame, ChainSecurity has performed an assessment in order to discover as many vulnerabilities as possible.

The focus of our assessment was limited to the code parts defined in the engagement letter. We assessed whether the project follows the provided specifications. These assessments are based on the provided threat model and trust assumptions. We draw attention to the fact that due to inherent limitations in any software development process and software product, an inherent risk exists that even major failures or malfunctions can remain undetected. Further uncertainties exist in any software product or application used during the development, which itself cannot be free from any error or failures. These preconditions can have an impact on the system's code and/or functions and/or operation. We did not assess the underlying third-party infrastructure which adds further inherent risks as we rely on the correct execution of the included third-party technology stack itself. Report readers should also take into account that over the life cycle of any software, changes to the product itself or to the environment in which it is operated can have an impact leading to operational behaviors other than those initially determined in the business specification.

4 Terminology

For the purpose of this assessment, we adopt the following terminology. To classify the severity of our findings, we determine the likelihood and impact (according to the CVSS risk rating methodology).

- *Likelihood* represents the likelihood of a finding to be triggered or exploited in practice
- *Impact* specifies the technical and business-related consequences of a finding
- *Severity* is derived based on the likelihood and the impact

We categorize the findings into four distinct categories, depending on their severity. These severities are derived from the likelihood and the impact using the following table, following a standard risk assessment procedure.

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

As seen in the table above, findings that have both a high likelihood and a high impact are classified as critical. Intuitively, such findings are likely to be triggered and cause significant disruption. Overall, the severity correlates with the associated risk. However, every finding's risk should always be closely checked, regardless of severity.

5 Findings

In this section, we describe any open findings. Findings that have been resolved have been moved to the Resolved Findings section. The findings are split into these different categories:

- Security: Related to vulnerabilities that could be exploited by malicious actors
- Design: Architectural shortcomings and design inefficiencies
- Correctness: Mismatches between specification and implementation

Below we provide a numerical overview of the identified findings, split up by their severity.

Critical-Severity Findings	0
High-Severity Findings	0
Medium-Severity Findings	0
Low-Severity Findings	1

- Trade Function Discrepancy Risk Accepted

5.1 Trade Function Discrepancy

Correctness Low Version 1 Risk Accepted

Users can call the trade functions `tradeBySourceAmount` and `tradeByTargetAmount` with `tradeActions` that contain the same strategy multiple times. During the execution of the trade functions, the corresponding strategies are updated after each trade action has been executed.

This is, however, not true for the view functions `tradeSourceAmount` and `tradeTargetAmount`: `y` and `z` values of the associated strategies are not updated during the execution. If the same strategy is traded against multiple times in one call, the trade result might therefore diverge from the result of the corresponding trade functions in the same state.

Risk accepted:

Bancor accepts the risk with the following statement:

> This is a known issue and is considered an edge case with minimal risk, as matching is done by the SDK that prevents this case. The alternative would be to check for duplicate strategies during the trade which would increase the gas costs for all trades, so we decided against adding such a check

6 Resolved Findings

Here, we list findings that have been resolved during the course of the engagement. Their categories are explained in the Findings section.

Below we provide a numerical overview of the identified findings, split up by their severity.

Critical-Severity Findings	0
High-Severity Findings	2

- Overflow Might DOS the App Code Corrected
- Price Precision Very Low for Some Tokens Code Corrected

Medium-Severity Findings	0
Low-Severity Findings	4

- Missing Events Code Corrected
- No Function for Fee Withdrawal Code Corrected
- Read-only Reentrancy Code Corrected
- Zero Amount ERC20 Transfers Code Corrected

6.1 Overflow Might DOS the App

Design High Version 1 Code Corrected

The `_tradeTargetAmount` and `_tradeSourceAmount` functions compute the input/output needed to perform a trade in a Strategy. However, Strategy inputs are only constrained by the `z` variable which should be greater or equal to `y`. This implies that `A` or `B` can be set arbitrarily by a user and `z` has no no limit to the upside.

Let's take the example of `_tradeTargetAmount`:

```
function _tradeTargetAmount(uint256 x, Order memory order) private pure returns (uint128) {
    uint256 y = uint256(order.y);
    uint256 z = uint256(order.z);
    uint256 A = uint256(order.A);
    uint256 B = uint256(order.B);

    if (A == 0) {
        return MathEx.mulDivF(x, B * B, ONE * ONE).toUint128();
    }

    uint256 temp1 = y * A + z * B;
    uint256 temp2 = (temp1 * x) / ONE;
    uint256 temp3 = temp2 * A + z * z * ONE;
    return MathEx.mulDivF(temp1, temp2, temp3).toUint128();
}
```

Here, `z` could potentially be `type(uint128).max`, which would imply that the `temp3` computation overflows, and the transaction reverts.

An attacker could create a simple Strategy with all parameter values (except the liquidity) maxed out. This Strategy will yield great results in the SDK: It returns extremely favorable rates for any trade. And as the computation in the SDK is not bound by 256 bit limits, these rates will always be sorted to the top of

each `getTradeData` call without errors. This results in all users relying on the SDK now trading against a Strategy that reverts on-chain.

Note that it might also happen when the `mulDiv` function's result ends up greater than `type(uint128).max` due to the safe downcast to 128 bits.

Code corrected:

The SDK now incorporates checks to verify that each calculation (e.g., `mulDivC`) does not exceed 256 bits and does not go below 0. Additionally, the functions `_tradeSourceAmount` and `_tradeTargetAmount` now calculate factors that are used to scale down intermediate numbers that are too big to fit in 256 bits.

6.2 Price Precision Very Low for Some Tokens

Correctness High Version 1 Code Corrected

Pairs containing a very low-value token and a low decimal token create some precision issues. Assume the existence of a token named TOK with 18 decimals and another token named USD with 6 decimals precision. Let's say the price of TOK is 0.00001 USD per TOK.

If a user wants to buy some TOK at constant price P = 0.00001 USD per TOK, then they need to compute the B parameter this way:

```
B = 2^32 * sqrt(0.00001 * 1e6 / 1e18)
B = 13.581879131294592
```

However, B must be an integer, meaning it will be rounded up or down (depending on the frontend implementation). In any case, the price will differ greatly from the intended price resulting in possible loss or no execution for the user.

Note: This issue was already disclosed by Bancor at the beginning of the audit.

Code corrected:

Bancor added precision by increasing the multiplying factor to 2^48 instead of 2^32, and implemented an encoding logic to be able to stretch the range of possible rates. Values greater than 2^48 now can have a (negligible) precision loss.

6.3 Missing Events

Design Low Version 1 Code Corrected

Some state-changing actions are missing an event emission. For example, most setters in the `Voucher` contract are not emitting events.

Code corrected:

Missing events have been added to all state-changing functions.

6.4 No Function for Fee Withdrawal

Design Low Version 1 Code Corrected

Each trade accumulates fees in `Strategies._accumulatedFees`. There is, however, no function to withdraw these fees. Withdrawal is still possible by assigning the `ROLE_ASSET_MANAGER` role to the owner of the contract and withdrawing funds via `MasterVault.withdrawFunds`. This is, however,

error-prone as more than the actual amount of fees could be withdrawn. The past has shown that faulty governance proposals can be executed. This should be avoided by restricting the fee withdrawal directly in the contract.

Code corrected:

A `ROLE_FEES_MANAGER` role was added along with a function to withdraw a specific amount of fees of a particular token.

6.5 Read-only Reentrancy

Security | Low | Version 1 | Code Corrected

When updating a strategy, funds are first transferred or withdrawn (which might lead to a callback to the user, for example if the native token is used) before the strategy is updated in storage. Considering the possibility of an external protocol integrating with Carbon, it might be the case that the protocol wants to measure the value of a strategy by reading the order parameters in storage. In this case, the read value would not correspond to the real value of the strategy.

For example, an external integration might accept a Strategy NFT and give the user something in return based on the liquidity the NFT holds. Consider the following process:

- The user approves the NFT for the given contract.
- The user calls `updateStrategy` to reduce the liquidity of the Strategy associated with the NFT.
- In the callback, the user calls the contract's function that transfers the NFT.
- The contract transfers the NFT, checks the associated liquidity and gives the user a return.
- The liquidity of the Strategy is reduced and the tokens are sent to the user.

Code corrected:

Order data is now updated in storage before any possible calls to the user are performed.

6.6 Zero Amount ERC20 Transfers

Correctness | Low | Version 1 | Code Corrected

Some tokens revert on transfers of 0 amount. Calls to `CarbonController.createStrategy` potentially try to perform such 0-amount-transfers if one of the given orders does not contain liquidity. The call would revert in this case.

Code corrected:

The code now transfers tokens only if the amount is greater than 0.

6.7 Ambiguous Naming

Informational | Version 1 | Code Corrected

Some functions / error messages are named ambiguously in contrast to the naming of other entities:

- The error `GreaterThanMaxInput` is thrown in `CarbonController.tradeByTargetAmount` when a value is smaller than `maxInput`.
- `CarbonController.tradeSourceAmount` and `CarbonController.tradeTargetAmount` are easily confused with `tradeByTargetAmount` and `tradeBySourceAmount`, but the meaning of `target` and `source` is reversed.

- `Strategies._tradeSourceAmount` and `_tradeTargetAmount` are easily confused with `tradeByTargetAmount` and `tradeBySourceAmount`, but the meaning of `target` and `source` is reversed.

Code corrected:

Function names were changed and the error name was corrected.

6.8 Gas Inefficiencies

Informational **Version 1** **Code Corrected**

1. If a pool does not already exist when creating a Strategy, the `_createPool` function returns the created pool but it is not used and instead read from storage afterwards.
2. The `Pool` struct `id` field is redundant as the struct can only be accessed through a mapping with Pool ID as key.
3. The `StoredStrategy` struct `id` field is redundant as the struct can only be accessed through a mapping with Strategy ID as key.
4. In `_createPool`, the `_poolIds` mapping is set up with the tokens in sorted and reversed order. Instead, `_pool` could always sort the tokens before searching in the mapping with little overhead.
5. When updating a Strategy, the packed orders could be updated word by word, just like in the `_trade` function.
6. In `tradeBySourceAmount` / `tradeByTargetAmount`, `_validateTradeParams` loads all Strategies associated with the given trade actions from storage. `_trade` later loads the Strategies from storage again.
7. In `_trade`, `StoredStrategy` fields are accessed multiple times (in storage), but could have rather been saved to memory once.
8. In `StoredStrategy`, the fields `owner` and `token1` might be redundant as they are not used in the contracts (except in events). Off-chain applications could extract the data elsewhere.

Code corrected:

1. The returned pool from the `_createPool` function is now being used.
2. The `id` field was removed from struct.
3. The `id` field was removed from struct.
4. Sorting is now performed where needed, instead of the double sided storage.
5. Orders are now updated word by word.
6. `validateTradeParams` no longer loads the strategies.
7. Trade flow now loads values only once.
8. Both owner and the tokens were removed from the strategies.

6.9 Typographical Errors

Informational **Version 1** **Code Corrected**

Some comments / symbol names contain typographical errors. Some examples are:

- `PoolDoesNotExists` error thrown in `Pools._validatePoolExistance`.
- The comment "revert here if the minReturn/maxInput constrants is unmet" in `Strategies._trade`.
- The comment "the address of the admin can't be change, so [...]" in `TransparentUpgradeableProxyImmutable`.

Errors corrected:

Typographical errors in various sections of the code have been improved or removed.

6.10 Unused Code

Informational Version 1 Code Corrected

A few pieces of code seem to be nut in use in the project anymore. This list is non-exhaustive:

1. In `CarbonController`, the error `ZeroLiquidityProvided` is unused.
2. In the `MathEx` library, the `Math` library of OpenZeppelin is imported but unused.
3. In `Strategies`, the `StrategyUpdate` struct is unused.

Code corrected:

Unused code has been removed.

6.11 `_updateOrders` Error Message

Informational Version 1 Code Corrected

`Strategies._updateOrders` removes the target amount of a trade from the respective order's `y` value. If this amount is actually larger than `y`, the transaction reverts on underflow. As this is a common case (for example for trades that are executed after some other trades on the same Strategy), an actual error message might make sense.

Code corrected:

An error message has been created and is returned for this specific case.

7 Notes

We leverage this section to highlight further findings that are not necessarily issues. The mentioned topics serve to clarify or support the report, but do not require an immediate modification inside the project. Instead, they should raise awareness in order to improve the overall understanding.

7.1 No Automatic Order Execution

Note Version 1

Orders are not evaluated against the set of currently active orders upon creation. This means, that an order with a price outside of the spread between buy and sell orders (on the respective "other side") is not executed against other orders but just included into the set of active orders. If a user were to place an order with a price that is not intended (e.g., due to keyboard input error), this order will be executed at exactly that price (as soon as a trade occurs), even if there are other open orders the order could be executed against for a better price.

7.2 Order Prices

Note Version 1

Users are able to choose order prices arbitrarily, making it possible to create strategies that do not make sense market-wise but might happen by mistake for example.

One could create a Strategy with the buy order prices being greater than the sell order prices, basically giving out liquidity to other users.

Another possibility is a Strategy with partially overlapping prices. In this case there are conditions that could lead to the user also losing liquidity. Consider the following example of a Strategy with opposite orders in the same price range:

- Fees are 0% for simplification.
- The buy order buys A tokens for 100 B tokens with a price range of [1, 4].
- The sell order sells 50 A tokens for B tokens with a price range of [1, 4].
- A trader now buys 100 B tokens for a total of 50 A tokens on the buy curve.
- The 50 A tokens are added to the liquidity of the sell curve, resulting in 100 A tokens liquidity. The capacity of the sell curve is adjusted to 100, moving the curve.
- The trader now immediately buys ~67 A tokens on the sell curve for the 100 B tokens they received before.
- The trader made an instant profit of ~17 A tokens.

For these reasons, it is important to note that users should be fully aware of the consequences of their strategy parameters choice.

7.3 Token Incompatibilities

Note Version 1

Some tokens are not compatible with the system:

- Tokens with fees.
- Rebasing tokens.

7.4 Trade Frontrunning

Note Version 1

`CarbonController.tradeBySourceAmount` and `tradeByTargetAmount` implement slippage protection in order to mitigate risks of frontrunning. However, as Strategies can be updated by their owners for relatively cheap and with arbitrary values, frontrunning trades in Carbon is simpler/cheaper than on traditional AMMs. For this reason, users should keep their slippage protection tight and expect their minima/maxima to be hit regularly.

Consider the following example:

- User1 creates a strategy with:
    - A = 1
    - B = 1
    - y = 100
    - z = 100
- User2 creates a trade transaction based on this strategy and expects to receive exactly 100 tokens for the 50 tokens they send in (not considering fees). They set the minimum amount of tokens they want to receive to 80.
- User1 observes the mempool and sees the transaction User2 just created. They now create an `updateStrategy` transaction with the following changes to their strategy:
    - A = 0.5
- User1 pays a miner to include this transaction before the transaction of User2 (or any other trade transaction that is performed on the given strategy) in the next block.
- After User2's transaction is executed, they receive only ~82 tokens instead of the expected 100.

7.5 `updateStrategy` Denial of Service

Note Version 1

`CarbonController.updateStrategy` employs a mechanism that ensures that a strategy has not been altered between the creation of a transaction and the actual execution. An attacker could theoretically block certain users from ever updating their Strategies by frontrunning their calls to `updateStrategy` with miniscule trades.

Nevertheless, users still have the possibility to delete and re-create their Strategies if they were targeted in such an attack.