Money on Chain
Decentralized Exchange Audit

October 2020

By CoinFabrik

Introduction

CoinFabrik was asked to audit the contracts for the Money on Chain Decentralized Exchange project. First we will provide a summary of our discoveries and then we will show the details of our findings.

Summary

The audited contracts are located in a private repository that belongs to Money on Chain. Contracts were tagged with version 1.3.0.

In this audit we reviewed the changes introduced in the contracts to support Market Orders since the previous audit performed in December 2019.

Additionally, we reviewed the correct usage of the Governance contracts.

Contracts

The audited contracts are:

- libs/MoCExchangeLib.sol: Library that manages the orderbook.
- libs/SafeTransfer.sol: Library to safely wrap token transfers.
- libs/TickState.sol: Library to manage tick state.
- token/BProToken.sol: BPro token implementation.
- token/DocToken.sol: Dollar on Chain token implementation.
- token/OwnerBurnableToken.sol: Token with burn functionality.
- token/WRBTC.sol: Token Wrapper for RBTC.
- CommissionManager.sol: Manages commission charges.
- ConfigurableTick.sol: Manages tick configuration settings.
- MoCDecentralizedExchange.sol: Decentralized Exchange main contract.
- OrderIdGenerator.sol: Manages the ID used for orders.
- OrderListing.sol: Manages orders creation, modification and removal.
- RestrictiveOrderListing.sol: Wraps OrderListing with some validations.
- TokenPairConverter.sol: Allows token prices conversion to a common token.
- TokenPairListing.sol: Manages token pairs creation in the exchange.

The following analysis were performed

- Misuse of the different call methods
- Integer overflow errors
- Division by zero errors
- Outdated version of Solidity compiler
- Front running attacks
- Reentrancy attacks
- Misuse of block timestamps
- Softlock denial of service attacks
- Missing or misused function qualifiers
- Insufficient validation of the input parameters
- Check for differences with provided specification documentation

Detailed findings

Severity Classification

Security risks are classified as follows:

- **Critical:** These are issues that we manage to exploit. They compromise the system seriously. They must be fixed **immediately**.
- **Medium:** These are potentially exploitable issues. Even though we did not manage to exploit them or their impact is not clear, they might represent a security risk in the near future. We suggest fixing them **as soon as possible**.
- **Minor:** These issues represent problems that are relatively small or difficult to take advantage of but can be exploited in combination with other issues. These kinds of issues do not block deployments in production environments. They should be taken into account and be fixed **when possible**.

- **Enhancement:** These kinds of findings do not represent a security risk. They are best practices that we suggest to implement.

This classification is summarized in the following table:

SEVERITY	EXPLOITABLE	ROADBLOCK	TO BE FIXED
Critical	Yes	Yes	Immediately
Medium	In the near future	Yes	As soon as possible
Minor	Unlikely	No	Eventually
Enhancement	No	No	Eventually

Issues Found by Severity

Critical severity

No issues of critical severity were found.

Medium severity

No issues of medium severity were found.

Minor severity

No issues of minor severity were found.

Observations

OrderListing is not properly initialized

The contract *OrderListing* inherits from *ReentrancyGuard*. The function *initialize* from ReentrancyGuard has to be called to properly initialize *OrderListing*.

Our analysis shows that this is not necessary in the ReentrancyGuard version used (openzeppelin-eth v2.2.0). Considering that in the new version v3.0.0 the implementation has changed and it is required to call initialize we propose to add the correct initialization so the contracts will not have problems in the future, and it will save some gas the first time ReentrancyGuard is used since it will already be initialized.

```
  function initialize(
      address _commonBaseTokenAddress,
      CommissionManager _commissionManager,
      uint64 _expectedOrdersForTick,
      uint64 _maxBlocksForTick,
      uint64 _minBlocksForTick,
      address _governor,
      address _stopper
  ) internal initializer {
      TokenPairConverter.initialize(_commonBaseTokenAddress,
_expectedOrdersForTick, _maxBlocksForTick, _minBlocksForTick,
_governor);
      OrderIdGenerator.initialize(0);
      commissionManager = _commissionManager;
      Stoppable.initialize(_stopper, _governor);

      // -- CoinFabrik:
      // -- Initialize ReentrancyGuard
      ReentrancyGuard.initialize();
  }
```

Using a boolean parameter can be confusing

A boolean parameter is used in several functions to specify if an order is either to buy or to sell tokens. Using boolean parameters can be confusing as it assumes that the values true and false means buy and sell, and without context it could mislead a user.

We suggest using an enum instead to avoid ambiguity. Example modifications required for function *doInsertLimitOrder* in MoCExchangeLib.sol:

```
  // -- CoinFabrik:
  // -- Available actions for an order
  enum OrderAction { SELL_ORDER, BUY_ORDER }

  function doInsertLimitOrder(
      Pair storage _self,
      uint256 _id,
      uint256 _exchangeableAmount,
      uint256 _reservedCommission,
      uint256 _price,
      uint64 _lifespan,
      uint256 _previousOrderIdHint,
      address _sender,
      address _receiver,
      OrderAction _action        // -- CoinFabrik: order action instead
of bool parameter
  ) public returns (uint256) {
    ..
    // -- CoinFabrik: Test action against OrderAction
    Token storage token = _action == OrderAction.BUY_ORDER ?
_self.baseToken : _self.secondaryToken;
    ..
  }
```

Internal functions are distinguishable from public functions

Often it is desirable to distinguish internal and private functions from public functions since they require different security approaches.

It is a common practice to prefix internal and private functions with an underscore so it is easy to differentiate them while reading the code without having to read the function signature.

Market order feature was not documented

In the provided documentation market orders were not specified so we were not able to check if the implementation follows the specification.

Other specific analyses performed

- We checked for spamming and tested for attacks involving filling the orderbook with expired or soon to be expired orders, and concluded that the matching execution is not affected.
- We tried to manipulate the price by waiting to be the only buyer/seller of a token-pair and then self-matching high price orders, with the purpose of being able to insert, less than allowed amount orders, after that.
- We also tried to prevent users from canceling their orders, by removing the order which id is prevHintId, from the orderbook, with the intention to revert the user transaction. Although this may happen, the user will just need to find the new *prevHintId* and resend the *cancelOrder* request, until it succeeds.
- Similarly we verified that *ProccesExpired* Orders can always be executed correctly even if the immediate previous order is canceled because *prevHintId* can refer to any previous order.

Conclusion

We found the contracts to have accurate documentation. The changes introduced to support Market Orders were simple and maintain the same structure and comments format. The functions are self explanatory making the complex mechanisms easy to understand.

The contracts have a good architecture, an appropriate library organization and a well chosen pattern design.

Disclaimer: This audit report is not a security warranty, investment advice, or an approval of the Money on Chain Decentralized exchange project. Moreover, it does not provide a smart contract code faultlessness guarantee.