Money on Chain
Decentralized Exchange Audit

December 16th, 2019

By CoinFabrik

Introduction

CoinFabrik was asked to audit the contracts for the MoC Dex project. Firstly, we will provide a summary of our discoveries and secondly, we will show the details of our findings.

Summary

The files for the contracts audited were provided to us in a compressed file.

The audited files are:

- libs/Orderbook.sol: Library with functions for handling the OrderBook
- libs/SafeTransfer.sol: Library with a wrapping function for safely transferring arbitrary tokens.
- libs/TickState.sol: Functions for handling Ticks and calculating their frequency.
- token/BProToken.sol: BPro Token implementation
- token/DocToken.sol: Dollar on Chain Token implementation
- token/OwnerBurnableToken.sol: Owner privileged token burning functionality.
- token/WRBTC.sol: Wrapper Token for RSK Bitcoin
- CommissionManager.sol: Handles commissions values, can be arbitrarily configured by authorized actors.
- ConfigurableTick.sol: Handles Tick global parameters, can be arbitrarily configured by authorized actors.
- MoCDecentralizedExchange.sol: The main contract for the exchange.
- OrderIdGenerator.sol: Generates sequential order ids with a counter.
- OrderListing.sol: Public functions for inserting and canceling orders.
- RestrictiveOrderListing.sol: Implements OrderListing and adds restrictions like minimal order amounts and order lifespans.
- TokenPairConverter.sol: Converts a token pair to Dollar on Chain value.
- TokenPairListing.sol: Handles addition and removal of Token Pairs by authorized actors.

Analyses

For the contract's code the following analyses were performed:

- Misuse of the different call methods: call.value(), send() and transfer().
- Integer rounding errors, overflow, underflow and related usage of SafeMath functions.
- Old compiler version pragmas.
- Race conditions such as reentrancy attacks or front running.
- Misuse of block timestamps, assuming anything other than them being strictly increasing.
- Contract softlocking attacks (DoS).
- Potential gas cost of functions being over the gas limit.
- Missing function qualifiers and their misuse.
- Fallback functions with a higher gas cost than the one that a transfer or send call allows.
- Fraudulent or erroneous code.
- Code and contract interaction complexity.
- Wrong or missing error handling.
- Overuse of transfers in a single transaction instead of using withdrawal patterns.
- Insufficient analysis of the function input requirements.

Contract Operation

MoC Dex as the name implies intends to be a decentralized exchange. In order to achieve decentralization it creates a concept called Ticks. Ticks are independent operations for each token pair that wants to be exchanged, and they are in charge of matching and fulfilling the orders of the pair that get added by users. Specifically, Ticks are atomic operations that are allowed to happen only after a variable amount of time (blocks) passed. The operation is atomic with respect to orders being introduced, so in a sense, the matching orders do not change while the entire Tick is executed.
Once allowed, a Tick can be started by any actor, and the total operation of the Tick can be split between multiple transactions by different actors by calling the same function in the contract. Internally, this is accomplished by a paging mechanism

called PartialExecution, which allows to split computations with elevated cost of gas in steps, which can be done in separate transactions using an internal state machine. Once all stages are completed the Tick finishes, another Tick can be started once a certain amount of time passed based on how many orders were fulfilled.

A Tick will execute these stages sequentially:

1. Simulation: It will simulate the matching of the orders without executing them. This allows to retrieve the final price that will be used to compute the orders in the next stage, and the last matching order as a stop condition. Expired orders are ignored.
2. Matching: In this stage, the orders will be executed based on the information retrieved in the previous stage. Any expired orders, that could be match, will also be removed.
3. Move Pending Orders: Orders that are introduced by users while the tick is running are moved to the order book. This mechanism allows orders to accumulate while the Tick is running. New orders should not interfere once the simulation and matching mechanism begin as these stages were designed to be executed atomically with respect to orders.

Contract Exploration

Aside from the general analyses mentioned previously, we specifically made the following checks to these contracts:

- Checked for DoS attacks in functions that had looping constructs. We'll mention them here and an explanation of why they are not considered vulnerable.
    - `findPreviousOrder: Is used for removing an order (when expire or cancel) to find the previous order and relink it to the next keeping the list ordered. This will fail if there are many orders starting from the first, to solve that startfromid parameter can be used to indicate where to start to look for the previous order.`
    - `findPreviousOrderToPrice: Is used to find where to add new orders, based on price, if there are too many orders this loops will fail due to gas limit in such case order insertion with hintid must be used.`
    - `getEmergentPrice: It's view, therefore it is expected to be only called locally without a transaction in the blockchain. It does not imply a gas problem.`
    - `processExpired: This function is expected to be called externally. It's not view, and it's used to process the expired orders in case of accumulation. The loop can be parametrized.`

- Checked for DoS Attacks in functions that had recursion constructs. We'll also mention these and an explanation of why they are not considered vulnerable.
    - `getNextValidOrder: It's executed in the Simulation phase. Find the next order that is not expired. Expired orders can be removed using processExpired.`
    - `getNextValidOrderForMatching: Same as the previous one, but also forcefully process expired orders.`

- Checked for DoS attacks by spamming the orderbook with irrational prices that will never be matched. In general, orders with irrational prices will never be iterated, as buy/sell orders are both sorted for best offer. The matching process will stop once matches are not found, and this coincides with the sorting order of both lists.
- Checked for price manipulation attacks in token pairs with little activity.
- Checked for inconsistencies between stages of a tick that resulted in contract softlocking.

Severity Classification

The security risk findings are evaluated according to the following classification:

- **Critical:** These are issues that we managed to exploit. They compromise the system seriously. We suggest to fix them **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 to fix them **as soon as possible**.
- **Minor:** These issues represent problems that are relatively small or difficult to exploit but can be used 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	Production roadblock	We suggest to fix it...
Critical	Yes	Yes	Immediately
Medium	Potentially	Yes	As soon as possible
Minor	Unlikely	No	When possible
Enhancement	No	No	Optionally

Detailed findings

Critical severity

None Found

Medium severity

Pending orders may be used to execute DoS under certain conditions

This attack relies on the fact that orders can still be inserted while a tokenpair is executing a Tick and the fact that to move the pending orders a hintid must be calculated off-chain and used correctly the fact that orders can be inserted with "0" expiration the affected Tick will have many trash pending orders to move, which the attacker can make very difficult to move by tampering with the orderid and keep pushing more trash orders.

The attack is described as follows:

S1. An attacker needs to wait for an orderbook to be empty and Tick to be runnable.

S2. By using a contract, insert one sell order and one buy order for a very high price and start the Tick. All three things would be in the same transaction preventing someone else to match the order.

S3. On the last step of the Simulation stage, the pair.lastClosingPrice will be set to a value calculated by the attacker's matching orders. This is important because pair.lastClosingPrice is used to convert the token to DoCs which is then used to check against the minimum order amount.

S4. The attacker can then insert many trash orders with almost zero REAL valuation, not according to the exchange, expending only the necessary gas to execute the function. These orders will be pending orders since the tick is still running.

S5. By finishing the matching stage the tick will be left in the last stage, requiring someone to move all the trash orders from pending into the orderbook. It can be very difficult to move these orders if their expiration is “0”. Because in this case the attacker can make them expire from the orderbook during the moving phase. This would invalidate the hint id needed to move the orders, and force a recalculation.

It’s important to mention that for a given token pair X/Y the price used to determine the minimum order amount for the token X is retrieved from the pair DoC/X. If said pair does not exist then the price is calculated for Y in the pair DoC/Y and then converted for the relation between X/Y. So even if the pair X/Y orderbook IS NOT empty at any moment but the pair X/DoC does not exist the attack can be performed anyway to the Y/DoC pair and the affected token pair would be X/Y without it being empty at any moment

1. The simplest but most impactful solution is to not have pending orders during a tick.

2. Another possible solution is to not depend on a single tick to calculate lastClosingPrice, which is then used to check if the amount is high enough to not be considered spam.

3. A solution which would not require contract code modification at all and has almost no gas cost is to implement an off-chain script that monitors for the required conditions that make the attack possible, empty orderbooks, and insert at least one valid order making S1 imposible.

Minor severity

Array length not checked on AddTokenPairChanger constructor

```
constructor  (
   MoCDecentralizedExchange _dex,
   address[] memory _baseTokens,
   address[] memory _secondaryTokens,
   uint256[] memory _precisions,
   uint256[] memory _prices)
 public {
   require(
     _baseTokens.length == _secondaryTokens.length &&
     _baseTokens.length == _precisions.length,
     "All three arrays must have the same length");
```

The require is missing the comparison between _prices.length and _baseTokens.lenght allowing a bad deploy to occur where the price array is not set or full making the post execution fail.

Observations

Recursion

Usage of recursion for iteration can result in running out of stack much before running out of gas. Moreover when compared with loops, it is less clear that there is an iteration happening that may result in DoS attacks.
In this project both recursive functions don't have this problem because expired orders are intended to be deleted previously.

Token SafeTransfer

The project uses a mechanism to transfer ERC20 tokens called SafeTransfer. Instead of making a normal call, it makes a low level call and informs if the transfer was successful or not, without propagating the exception. Since the exception is not propagated the rest of the logic is still executed and the tokens will end up trapped inside the contracts instead of being sent to the recipient. This is needed, as matching orders always have to be executed to have a fair exchange, regardless of what is happening to the underlying token. In the end, tokens should be audited to not have unexpected reverting behavior before being added to the exchange.

Non compliant ERC20 tokens can make SafeTrasnfer fail

SafeTransfer retrieves the boolean return value of the transfer call by using abi.decode:

```
    // transfer completed successfully (did not revert)
    bool callResult = abi.decode(returnData, (bool));
```

Since the return value is not part of the signature of a function, some old non compliant tokens do not return a boolean value. In that case abi.decode will revert, likely softlocking the token pair in which is included. We do not consider this a major issue as it only affects the functionality related to the token itself and as we said before, tokens should be audited to ensure they can work with the exchange before being added.

Decentralization considerations

Since the most important project pillar is being decentralized, we decided to add this section to document behaviors in the contracts that may impact or be against decentralization. We understand that these privileged mechanics are bound to be connected to a decentralized gobernanza solution, but we are assuming a worst case scenario where manipulation is possible. In general, users may appreciate and trust more a pure decentralized solution.

Token pair disable

There are two functions in the contract TokenPairListing.sol that allows an authorized actor to enable or disable any token pair. Disabling a token prohibits users from inserting new orders into the token pair orderlists. This is obviously not decentralized as a privileged author shouldn't be allowed to discriminate when an order can be inserted on a given token pair.

Exchange pausing

The exchange contract is Stoppable and most functionality can be disabled by pausing. This includes inserting orders, canceling orders, and matching orders. This is global and affects every token pair available. While the contract is paused, every fund residing inside the contract will be frozen, as the functionality to either execute or refund them will be disabled.

Commissions changes

There are functions that allow an authorized actor to change commissions. Users may have agreed to insert an order at a given commision but the commision is paid at the end of the match so it can change during this time even to a %100 making the user lose all his funds.

Min blocks for tick

The are some changer contracts to set how many blocks are needed to run the next tick. An authorized changer can set a high number of blocks preventing the tick to run when users agreed to in the moment of placing the orders.

Conclusion

We have looked into the contracts and the solution is solid. The potential issues are mostly related to the overall throughput of the exchange, and not on the security of the orders themselves. Even though some functionalities are centralized, we understand that it will be controlled by a gobernanza which will mitigate this centralization
Overall a lot of thought was put into these contracts. They expose a lot of understanding of how the blockchain works in general.