PUBLIC

Code Assessment

of the Exchange V2 Smart Contracts

25 June, 2021

Produced for

Rarible Inc.

by

CHAINSECURITY

Contents

1 Executive Summary

Dear Sir or Madam,

First and foremost, we would like to thank Rarible Inc. for giving us the opportunity to assess the current state of their Exchange V2 system. This document outlines the findings, limitations, and methodology of our assessment.

The majority of raised issues were fixed by modifications of code or by clarified documentation. Two low severity and one medium severity issues were not fixed, as the risk was accepted by the Rarible Inc.. Even though they do not threaten the main functionality of the Exchange V2, we suggest revisiting these issues.

We hope that this assessment provides more insight into the current implementation and provides valuable findings. We are happy to receive questions and feedback to improve our service and are highly committed to further support your project.

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	1
• Code Corrected	1
Medium-Severity Findings	4
• Code Corrected	1
• Specification Changed	2
• Risk Accepted	1
Low-Severity Findings	5
• Code Corrected	2
• Specification Changed	1
• Risk Accepted	2

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 Rariable repository in `exchange-v2` folder, based on the documentation files. In addition contract `LibAsset` from the same repository `asset` folder is included in the scope. The table below indicates the code versions relevant to this report and when they were received.

V	Date	Commit Hash	Note
1	13 May 2021	ffb48d0c6c3593a8ea8e41669d370057315b8265	Initial Version
2	14 June 2021	f62e271ef3affb65f37d47723f6060f434f29152	Version with fixes
3	18 June 2021	1ea1ac9f965553cf749a646e65124e1e23966681	Version with fixes

For the solidity smart contracts, the compiler version `0.7.6` was chosen.

2.1.1 Excluded from scope

Contracts from test subfolders inside the folders mentioned in the Scope. External library dependencies and any UI components that might interact with the contracts.

2.2 System Overview

This system overview describes the 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.

Exchange V2 implements two main functionalities: order matching (`matchOrder`) and order cancellation (`cancelOrder`).

When a pair of valid orders is matched, at least one of the orders gets fully filled. Then, the fees and royalties are paid to the corresponding parties. The filling of the order is measured by the received take asset of the order. Due to flooring of the estimation of the remaining make amount from remaining take amount, some leftover make assets can be unsellable. Depending on the arrangement of arguments, the two orders of the pair are named Left and Right. An order is valid if its signature is valid or the invoker of the `matchOrder` is also the maker of the order. A pair of orders is matchable if:

1. For both orders, the receiver (`taker`) of the order, if defined, is the same as the offerer (`maker`) of the other one. If no taker is defined any offerer can match.
2. The asset types used in the orders match, meaning the make asset type of one order should match the take asset type of other order.
3. The make/take ratios of orders allow them to be filled. In other words, the seller and buyer can agree on the price. In case of matchable but different prices the left order dictates the price of the exchange. Because of the uint arithmetics, prices are estimated by uint and checks prevent price slippage with 0.1% accuracy.

The orders can be separated into two categories:

1. **Salted orders**: In these orders a `salt` (a random number) is defined. The status of these orders is stored on the blockchain. An order can be partially filled. These orders can be canceled.
2. **Ad-hoc orders**: they have `salt` set to 0 and need to be sent to contract directly by the maker. Filling degree tracking is off for such orders, while only the maker can resubmit the order.

A normal order is canceled by setting its filling degree to the maximum possible value. Cancellation of the order is possible only by the maker of the order.

2.2.1 Fees

Depending on the asset types in the orders, one of them can be chosen as an asset for fee payment. In the current setup, fees can only be paid if ERC20, ERC1155 or native ether are among the order asset types.

If both make and take assets of the order can pay fees, the left order make asset pays the fees.

Maker of the fee paying asset needs to provide fees on top of the order maker value. Taker of the fee paying asset will receive amount reduced by the fees.

2.2.2 Trust Model

We assume that the users are fully aware of the what orders they sign. An order formed by a malicious party can lead the seller to send the amounts to addresses that they do not control.

Owner of the contract is considered to be trusted party, that will not set up malicious contract parameters such as platform fees or transfer proxies.

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 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 associated with the items defined in the engagement letter on whether it is used in accordance with its specifications by the user meeting the criteria predefined in the business specification. 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 the facts that over the life cycle of any software product, changes to the product itself or to its environment, in which it is operated, can have an impact leading to operational behaviours other than 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 severities. 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. All of 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	1

- Cancel Order Authorization Differs From Match Risk Accepted

Low-Severity Findings	2

- Dependency on EIP712Upgradeable Risk Accepted
- Missing Indexes In Events Risk Accepted

5.1 Cancel Order Authorization Differs From Match

Design Medium Version 1 Risk Accepted

Function `validate` of `OrderValidator` contract permits matches in cases when the message sender is not the order maker. This can be done when the order maker is an ERC1271 implementation or when the sender provides a valid signature. During the cancellation the only check that is done is:

```
require(_msgSender() == order.maker, "not a maker");
```

This check is more strict than the `matchOrders` authorization rules and limits the possible pool of parties that can use this entry-point, for example, ERC1271 contracts cannot cancel their orders.

5.2 Dependency on `EIP712Upgradeable`

Security Low Version 1 Risk Accepted

Contract `OrderValidator` uses `EIP712Upgradeable` contract from openzeppelin library, which is currently in a draft stage. That increases the risk of bugs and errors in all contracts that use this dependency. In addition draft library contracts tend to be inefficient. For example in current version, every call to _hashTypedDataV4 triggers two storage lookups (_EIP712NameHash(), _EIP712VersionHash()) which together cost 4200. That is fairly unnecessary.

```
import "@openzeppelin/contracts-upgradeable/drafts/EIP712Upgradeable.sol";
```

5.3 Missing Indexes In Events

Design Low Version 1 Risk Accepted

In `ExchangeV2Core` the events `Cancel` and `Match` contain no indexed fields. Indexing order hashes will help to avoid performance issues on node clients.

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	1
Medium-Severity Findings	3
Low-Severity Findings	3

- doTransfers Does Not Hanlde LibFeeSide.FeeSide.NONE Code Corrected
- Function safeGetPartialAmountFloor Precision Problems Specification Changed
- Order Salt Problems Specification Changed
- Orders With Salt 0 Can Be Canceled Code Corrected
- Compiler Version Not Fixed Code Corrected
- Contracts Can Be Order Makers Specification Changed
- Precision Check in calculateRemaining Problem Code Corrected

6.1 `doTransfers` Does Not Hanlde `LibFeeSide.FeeSide.NONE`

Design High Version 1 Code Corrected

`doTransfers` performs the transfer of assets after choosing which is the feeable side. However, `getFeeSide` can return the value `LibFeeSide.FeeSide.NONE` in the case none of the assets are `ETH` or `ERC20` or `ERC1155`. This value is not handled by the function `doTransfers` which results to the transfer not being performed.

Code corrected:

`doTransfers` was changed to handle `LibFeeSide.FeeSide.NONE`.

6.2 Function `safeGetPartialAmountFloor` Precision Problems

Correctness Medium Version 1 Specification Changed

The function `safeGetPartialAmountFloor( uint256 numerator, uint256 denominator, uint256 target)` defined in `LibMath` contract effectively computes the `numerator * target / denominator` and reverts on too much divergence from the correct value. Due to the different nature of tokens (ETH, ERC20, ERC721, etc.) and different decimals on them, the actual values sent to this function can be of different orders. In cases when the denominator is greater

than the `numerator * target` the 0 will be returned. This can lead to situations when the orders cannot be matched. For example order "Buy 30 for 600X" cannot be matched with order "Sell X for 10", because the `fillRight` function that relies on `safeGetPartialAmountFloor` will return (10, 0) value that later will fail the check in `matchAndTransfer` function.

The `safeGetPartialAmountFloor` function is used in following places:

- Function `fillLeft` in `LibFill` contract.
- Function `fillRight` in `LibFill` contract.
- Function `calculateRemaining` in `LibOrder` contract. In this case, big, close to filling values may fail.

Specification corrected:

Now the specification correctly communicates the behavior of the contract.

6.3 Order Salt Problems

Design Medium Version 1 Specification Changed

The salt is effectively a field of an order that allows different orders of the same asset types from the same maker to be distinguishable from each other. This field is also part of the `hashKey` of the order that is used to track the filling of the order. However, due to the lack of Asset values in the `hashKey`, the same value for salt can be resubmitted with higher-order take value, and thus lead to multiple full filling of the same order. For example, an order that makes 20 take X after filling can be resubmitted with the same salt and higher take limit: make 30 take 2X. Note that after cancellation the salt becomes unusable for the maker. From a specification point of view, it the order with same `hashKey` shouldn't be fully filled multiple times.

```
function hashKey(Order memory order) internal pure returns (bytes32) {
    return keccak256(abi.encode(
            order.maker,
            LibAsset.hash(order.makeAsset.assetType),
            LibAsset.hash(order.takeAsset.assetType),
            order.salt
        ));
}
```

Specification corrected:

The behavior was documented and properly described in `exchange-v2/readme.md`.

6.4 Orders With Salt 0 Can Be Canceled

Design Medium Version 1 Code Corrected

The filling degree of orders with salt 0 is not tracked in the `matchOrders` function. But the `calculateRemaining` function will use the value from `fills` map to compute the remaining value that needs to be filled. The `cancel` function effectively sets the `fills` map value to the `UINT256_MAX` value. Users can also cancel orders with salt 0, effectively making the asset pair not longer usable with salt 0.

Code corrected:

A check that prevents 0 salt order cancellation was added.

6.5 Compiler Version Not Fixed

Design | Low | Version 1 | Code Corrected

The solidity compiler is not fixed in the code. In addition, different files define different pragmas. The version, however, is defined in the `truffle-config.js` to be `0.7.6`. In the code the following pragma directives are used:

```
pragma solidity >=0.6.2 <0.8.0;
pragma solidity >=0.6.9 <0.8.0;
```

Code corrected:

The pragma was fixed to 0.7.6 for all contracts.

6.6 Contracts Can Be Order Makers

Design | Low | Version 1 | Specification Changed

Maker and Taker of orders can be contracts with the help of the ERC1271 standard. In addition, fee receiving parties can be contracts too. If native ether is used as an asset during the match, the transfers can fail if the contracts do not implement a payable fallback function. The system specification should clearly communicate this requirement to the users.

Specification corrected:

The expectations from contracts were documented and described in `exchange-v2/readme.md`.

6.7 Precision Check in calculateRemaining Problem

Design | Low | Version 1 | Code Corrected

Due to a precision check in function `calculateRemaining` orders with different magnitudes of take and make values can become unfillable even with a small filling degree. For example, Order with make 10 take 100 cannot be filled if fill amount of take is 15. In `calculateRemaining` the remaining make value for that order will be approximated with value 8. Because the true value of 8.5 cannot be expressed with integer numbers, the error of 0.5 will exceed 0.1% limit that is built-in in `calculateRemaining` due to utilization of `LibMath.safeGetPartialAmountFloor` function.

Code corrected:

The precision check in `calculateRemaining` function was removed in commit 839710b1bd7ed11fc22fa2093f408934b92ccf35. This fix prevents premature order freeze. With the fix,

only the last make item of the order can be unsellable. For example, an order with make 10 take 100 cannot be fully filled if the fill amount of take is 95, as the 0.5 make value will be estimated by `calculateRemaining` function as 0. With help of order extension functionality, such orders can be fixed via signature resubmission with greater values. The precision check for price computation is still used.

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 Accumulation of Rounding Errors

Note Version 1

The fullfilment of an order is tracked by the `fills` mapping. The remaining part comes from the subtraction of the value the `fills` mapping holds for a particular order from the total take value of the order.

```
if (orderLeft.salt != 0) {
     fills[leftOrderKeyHash] = leftOrderFill.add(newFill.takeValue);
}
if (orderRight.salt != 0) {
    fills[rightOrderKeyHash] = rightOrderFill.add(newFill.makeValue);
}
```

However, the value added to the `fills` mapping is a result of a division occuring in `LibOrder.calculateRemaining`. Division might introduce some rounding erros which gradually accumulate if an order is partially filled multiple times. Notice that the implemenation tolerates a 0.1% rounding error.

```
function calculateRemaining(Order memory order, uint fill) internal pure returns (uint makeValue, uint takeValue) {
    takeValue = order.takeAsset.value.sub(fill);
    makeValue = LibMath.safeGetPartialAmountFloor(order.makeAsset.value, order.takeAsset.value, takeValue);
}

function safeGetPartialAmountFloor(
    uint256 numerator,
    uint256 denominator,
    uint256 target
) internal pure returns (uint256 partialAmount) {
    if (isRoundingErrorFloor(numerator, denominator, target)) {
        revert("rounding error");
    }
    partialAmount = numerator.mul(target).div(denominator);
}
```

7.2 AssetMatcher Gas Efficiency

Note Version 1

The `matchAssetOneSide` function in AssetMatcher contract effectively decides if two assets types can be matched. It also contains logic for matching assets that are not yet known to the systems:

```
if (classLeft == classRight) {
    bytes32 leftHash = keccak256(leftAssetType.data);
    bytes32 rightHash = keccak256(rightAssetType.data);
    if (leftHash == rightHash) {
        return leftAssetType;
    }
}
```

This piece of code works for all known asset types as well. In addition, it is more efficient than the current `matchAssetOneSide` for most of the known asset types.

7.3 Incentives for Front-Running

Note **Version 1**

In case when 2 assets that can pay fees are exchanged, the order of arguments in `matchOrders` function might matter. Moreover it determines the price and thus the amounts exchanged between the two parties. There might be third parties that are incentivised to front-run the transactions in order to determine the position of the orders for their own interest. The users should be aware of such events. In addition, once the transactions are visible in the mining pool, any other parties can try to frontrun the match, to profit from matching with lower fees or good price.

Illustration of order importance:

Let A and B be an ERC20 and ERC1157 token respectively. Accoring to the contract logic currently implemented, the feeable token is A. Assume two orders O1:(10A, 20B) and O2:(50B,11A) Executing `matchOrder(O1, O2)` yields `fillResult(10A, 20B)` (`fillLeft` will be called). On the other hand, executing `matchOrder(O2, O1)` yields `fillResult(20B, 220/50A)` (`fillRight` will be called). Assuming a fee of 10% then in the first case we have 0.1 * 10A and the second 0.1* 220/50 A

7.4 Orders Can Pay No Fees

Note **Version 1**

Before transfering the assets to the corresponding parties the fee side is chosen. The side is chosen to be the one that offers `ETH` or `ERC20` or `ERC1155`. If there is no such types in make and take assets of the order, the fees won't be deducted.

7.5 Reentrancy Risk

Note **Version 1**

In the `matchOrders` can occur calls to other contacts and addresses. For example, during the native ether transfer or during the transfer of tokens that allow user hooks e.g. ERC777 (extension of ERC20). While we haven't identified a direct way, how this can be abused. But risk of reentrancy is nullified when a non-reentrant lock is used, for a price of small gas cost increase.

In addition, following transfers of ether will send all the gas to the callee, allowing it to execute any other contract with no restrains.

```
(bool success,) = to.call{ value: value }("");
```

7.6 The Order of Orders Determines The Price

Note **Version 1**

In centralized order book-based exchanges, the price of matchable orders with different prices is usually determined by the order with the earliest submission time. In the current implementation, the price is determined by the left order. While the centralized method is not applicable to this system, the current behavior should be documented in specification, as the users should be aware of the price formation.

7.7 Use of `SafeMath`

Note Version 1

There are many instance where the `SafeMath` is not used. Such calculations can lead to overflows and, thus, unexpected behavior. No dangerous overflows have been found during the overflow, however, the use of `SafeMath` is recommended. For example such calculations happen in `transferPayouts` function on `sumBps` accumulator.

7.8 Validate Gas Efficiency

Note Version 1

Function `validate` in OrderValidator contract can be restructured for a lower gas cost. The `isContract` check is performed in all cases when the message sender is not the maker. Assuming that the most popular cases are when the maker is not a contract, the signature check can be performed first, before the `isContract` check.