PRIVATE

Code Assessment

# of the On-Chain Orders Smart Contracts

Dec 17, 2021

Produced for

by

CHAINSECURITY

Contents

1 Executive Summary

Dear Alexei and Eugene,

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

Rarible, in the current implementation, extends the functionality of the ExchangeV2 so that it can store orders on chain. Users are allowed to update or cancel these orders. Otherwise, integrate as normal orders with the rest of the system. Users should lock ETH on the Exchange contract for orders that offer/make ETH.

The most critical subjects covered in our audit are functional correctness, access control and precision of arithmetic operations. Security regarding the functionality is low. During this assessment we found numerous critical vulnerabilities that allow an attacker to drain the locked Ether from the Exchange contract. Security with regards to the access control and arithmetic operations precision suffices.

The general subjects covered are documentation, gas efficiency and error-handling. Regarding the aforementioned areas, the security is satisfactory.

In summary, we find that the codebase at last version commit in Scope provides provides a low level of security. The current state of code does not take into account the reentrant behavior of the EVM smart contract calls. The codebase requires alteration and cannot be used in production at current state. Bugs listed in the Findings section need to be fixed before the Exchange contract upgrade.

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	3
• No Response	3
High-Severity Findings	1
• No Response	1
Medium-Severity Findings	3
• No Response	3
Low-Severity Findings	0

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 On-Chain Orders repository based on the `exchange-v2/readme.md` documentation file. The table below indicates the code versions relevant to this report and when they were received. It is important to note that the current assessment reviews only the changes of a code base which has already been audited in a previous assessment.

V	Date	Commit Hash	Note
1	5 Dec 2021	319c4c5916ac9f1b573cfc2a2630eca44460258d	Initial Version

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

The following files are considered in scope:

- exchange-v2/contracts/ExchangeMetaV2.sol
- exchange-v2/contracts/ExchangeV2.sol
- exchange-v2/contracts/ExchangeV2Core.sol
- exchange-v2/contracts/LibOrder.sol
- exchange-v2/contracts/RaribleTransferManager.sol
- exchange-v2/contracts/ITransferManager.sol
- exchange-v2/contracts/LibFill.sol
- exchange-v2/contracts/LibOrderData.sol
- exchange-v2/contracts/LibOrderDataV2.sol
- exchange-v2/contracts/OrderValidator.sol
- exchange-v2/contracts/TransferExecutor.sol
- exchange-v2/contracts/interfaces/IERC1271.sol

2.1.1 Excluded from scope

From the scope are excluded:

- All other repository files that are not mentioned in the scope.

2.2 System Overview

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

This assessment covers the extension of the Rarible Exchange V2 system functionality. Review report of the core functionality can be found here: ChainSecurity Exchange V2 audit.

The newest extension to the Exchange V2 system functionality allows for the creation of onchain stored orders via `upsertOrder` function. The orders that are created this way, are stored in the storage of the contract. The on-chain order can only be created if the order maker is the message sender. The on-chain order that is argument of the `matchOrders` function, doesn't require maker's signature argument. The onchain orders can be canceled using `cancel` function. Protocol fee changes don't affect the fee that the on-chain orders pay, as it is queried and stored onchain during the order creation. However, when an order is updated the protocol fees are updated with the newest value defined by the system.

Another change to the functionality includes the `isMakeFill` order data flag, that specifies, which order asset (take or make) the filling should track.

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

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 our findings. 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

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

Critical-Severity Findings	3

- Function ExchangeV2Core.cancel Reentrancy
- Function upsertOrder Reentrancy
- Onchain Orders With 0 Salt

High-Severity Findings	1

- On-chain Order Filling Inconsistency

Medium-Severity Findings	3

- Function deleteFilledOrder Fill Measurment
- On-chain Order Locked Ether
- Solidity 0.7.6 Compiler Bug

Low-Severity Findings	0

5.1 Function `ExchangeV2Core.cancel` Reentrancy

Security Critical Version 1

The `cancel` function relies on `transferLockedAsset` for ether transfer. However, this function effectively allows reentrancy, if the order maker is a contract implementing a fallback function. The `onChainOrders` entry is deleted only after the ether transfer. Thus, the attacker can create an on-chain order and lock the ether as the make asset. Then he can cancel the order multiple times with help of reentancy. After each reentrancy cycle, the attacker will get full initial lockup amount. Since the `ExchangeV2Core` contract all the locked ether for on-chain orders, attacker can fully drain the balance of the exchange contract.

Example of call trace:

```
cancel
    transferLockedAsset
        cancel
            transferLockedAsset
            delete onChainOrders[orderKeyHash];
    delete onChainOrders[orderKeyHash];
```

5.2 Function `upsertOrder` Reentrancy

Security Critical Version 1

The `ExchangeV2Core.upsertOrder` function returns the locked ether back to the order maker, if the new order make asset value is smaller then the old order one. The ether transfer is done via `LibTransfer.transferEth` function. This function sends all available gas using the `call` primitive. This effectively allows a reentancy attack.

Example of a possible attack:

- Initialy, the attacker calls `upsertOrder` with Make 200 ETH.
- The attacker is a contract with implemented fallback function.
- The attacker calls `upsertOrder` with Make 100 ETH with same hashKey as 1st order. During the ether transfer the maker contract fallback calls same `upsertOrder` with Make 100 ETH. This reentrancy loop can be repeated multiple times, since the `onChainOrders` entry is updated only after the call.
- The reentant `transferEth` call returns and the order is updated on chain:

  ```
  onChainOrders[orderKeyHash].fee = getProtocolFee();
  onChainOrders[orderKeyHash].order = order;
  ```

Example of call trace:

```
upsertOrder
    transferEth
        upsertOrder
            trasferEth
        onChainOrders[orderKeyHash].fee = getProtocolFee();
        onChainOrders[orderKeyHash].order = order;
    onChainOrders[orderKeyHash].fee = getProtocolFee();
    onChainOrders[orderKeyHash].order = order;
```

5.3 Onchain Orders With 0 Salt

Security Critical Version 1

The `upsertOrder` function allows the creation of on-chain orders with 0 salt. The fillings are not tracked for the 0 salt orders. Thus, any on-chain order with 0 salt can be matched multiple times. The `upsertOrder` function requires the order maker to lock the full amount of ETH to cover the full filling. This means that the ExchangeV2 contract holds all the locked funds. Any repetitive matches with 0 salt on-chain order will drain the balance of the ExchangeV2 contract.

Example of an attack:

Assume the attacker has an ERC-1155 Token (NFTs) in great amount. Attacker create both A and B orders himself.

1. The attacker upserts the order A:Make 100 ETH, Take 1 NFT, Salt 0. This step locks 100 ETH on the ExchangeV2 contract.
2. Matches A with order B: Make 1 NFT and take 100 ETH. This step transfers 100 ETH from ExchangeV2.
3. Repeat step 2 with B order salt change.

This way the attacker can transfer more then 100 ETH while locking only 100 ETH.

In addition, the requirement statement in `cancel` function does not allow cancellation of 0 salt onchain orders.

5.4 On-chain Order Filling Inconsistency

Design High Version 1

The `matchOrders` function can be called with orders that have same hash key as on-chain orders, while having some differences from on-chain stored order. For example, orders can have same asset types, but different values. In the `matchAndTransfer` function the fillings of orders is first adjusted and then the orders are deleted if they are filled. Because fills are tracked using the hash key of the order, the on-chain order filling value can be inconsistent.

For example assume two orders, A and B. A is stored on chain:

- It holds `hashKey(A) == hashKey(B)`.
- A: make 100, take 1.
- B: make 1000, take 1.
- Both have `isMakeFill == true`.
- If B is matched, it will set the filling of A to 1000.
- However, the A order will stay on chain and won't be deleted as a fulfilled order since order B is not the same as A (`isTheSameAsOnChain` returns false).

In addition, if make asset of A was Ether, the funds of A will stay locked on chain. The A order cannot be canceled due to safeMath underflow check trigger in `LibOrder.calculateRemaining` function. During the B match the assets need to be transferred again.

5.5 Function `deleteFilledOrder` Fill Measurment

Design Medium Version 1

The `ExchangeV2Core.deleteFilledOrder` function does not use the `isMakeFill` order flag, and instead always assumes that the order filling is tracked by the take asset.

5.6 On-chain Order Locked Ether

Design Medium Version 1

If some ether is sent along with the `matchOrder` function call, where one of the orders is onchain order with ether as a make asset, the `returnChange` won't return the leftover funds to the the msg.sender. This is done because the code assumes that all needed ether is already locked in the contract and thus `matchRequiresEth` will return false. However due to ther human mistakes, users still can send some ether to payable `matchOrder` function, when it is not needed. This ether will be locked on the contract.

5.7 Solidity 0.7.6 Compiler Bug

Security Medium Version 1

The project relies on the solidity compiler version 0.7.6. This version has a known bug: https://blog.soliditylang.org/2021/04/21/decoding-from-memory-bug/. The decoding function mentioned in the bug is used in the `LibOrderDataV2.decodeOrderDataV2` function. The `DataV2` struct contains the arrays that can potentially be used to trigger the compiler bug.

```
function decodeOrderDataV2(bytes memory data) internal pure returns (DataV2 memory orderData) {
    orderData = abi.decode(data, (DataV2));
}
```

Same bug can be also triggered in function mentioned in the bug is used in the `LibOrderDataV2.decodeOrderDataV2` function. While we believe the bug can be definitely triggered, the effect of it seems to be somewhat limited, due to the memory copying mechanics of the solidity. This bug only affects the user who signed weird `orderData`, and thus one can say, that this bug only can be triggered if user blindly signed the calldata.

6 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.

6.1 Function `ExchangeV2Core.returnChange` Reentrancy

Note **Version 1**

Function `returnChange` transfer the funds back to the `msg.sender` under specific conditions. This transfer sends along all the gas and thus external contract code can be executed freely. The field `deleteFilledOrder` is deleted after this call and thus this can be seen as a reentrancy bug. There is no lucrative attacks we have founds that malicious actors can perform, but nevertheless we suggest making no storage modifications after the external calls.