V2 Matching (LiquidateModule and RFQModule) Smart Contract Audit

Lyra, 29 January 2024

Confidential
17 Tyrwhitt Avenue, Johannesburg, 2196, South Africa
hello@iosiro.com

Contents

1. Introduction 2
2. Executive summary 3
3. Audit details 4
3.1 Scope 4
3.1.1 Smart contracts 4
3.2 Methodology 4
3.2.1 Code review 4
3.2.2 Dynamic analysis 5
3.2.3 Automated analysis 6
3.3 Risk ratings 6
4. Design specification 7
5. Detailed findings 9
5.1 High risk 9
5.2 Medium risk 9
5.3 Low risk 9
5.4 Informational 9
5.4.1 Manager mismatch 9
5.4.2 Design comments 11
5.5 Closed 12
5.5.1 Incorrect calculation of price delta (high-risk) 12

1. Introduction

iosiro was commissioned by Lyra to conduct a smart contract audit of their V2 matching system's `RFQModule` and `LiquidateModule`. The audit was performed by 1 auditor between 17 January 2024 and 24 January 2024, using 5 resource days.

This report is organized into the following sections.

- Section 2 - Executive summary: A high-level description of the findings of the audit.
- Section 3 - Audit details: A description of the scope and methodology of the audit.
- Section 4 - Design specification: An outline of the intended functionality of the smart contracts.
- Section 5 - Detailed findings: Detailed descriptions of the findings of the audit.

The information in this report should be used to understand the smart contracts' risk exposure better and as a guide to improving the security posture of the smart contracts by remediating issues identified. The results of this audit reflect the in-scope source code reviewed at the time of the audit.

The purpose of this audit was to achieve the following:

- Identify potential security flaws.
- Ensure that the smart contracts function according to the documentation provided.

Assessing the off-chain functionality associated with the contracts, for example, backend web application code, was outside of the scope of this audit.

Due to the unregulated nature and ease of transfer of cryptocurrencies, operations that store or interact with these assets are considered high risk from cyber attacks. As such, the highest level of security should be observed when interacting with these assets. This requires a forward-thinking approach, which takes into account the new and experimental nature of blockchain technologies. Strategies that should be used to encourage secure code development include:

- Security should be integrated into the development lifecycle, and the level of perceived security should not be limited to a single code audit.
- Defensive programming should be employed to account for unforeseen circumstances.
- Current best practices should be followed where possible.

2. Executive summary

This report presents the findings of an audit performed by iosiro on the `LiquidateModule` and `RFQModule` that are extensions to the existing Lyra V2 matching system.

Audit findings

iosiro noted one high-risk issue related to incorrect calculation of the cash value of Perpetual Assets when settling RFQ orders. A number of security in-depth and code quality recommendations were also provided.

The high-risk issue was resolved during the audit, and the informational recommendations were discussed with and acknowledged by the development team.

Recommendations

At a high level, the security posture of the smart contracts in-scope was found to be of a high standard; however, it is recommended that additional audits are performed at regular intervals, as security best practices, tools, and knowledge change over time.

3. Audit details

3.1 Scope

The source code considered in-scope for the assessment is described below. Code from all other files was considered to be out-of-scope. Out-of-scope code that interacts with in-scope code was assumed to function as intended and not introduce any functional or security vulnerabilities for the purposes of this audit.

3.1.1 Smart contracts

Project name: v2-matching
Commits: 5519dbe, d8120a4
Files: RfqModule.sol, LiquidateModule.sol

3.2 Methodology

The audit was conducted using a variety of techniques described below.

3.2.1 Code review

The source code was manually inspected to identify potential security flaws. Code review is a useful approach for detecting security flaws, discrepancies between the specification and implementation, design improvements, and high-risk areas of the system.

3.2.2 Dynamic analysis

The contracts were compiled, deployed, and tested in a test environment, both manually and through the test suite provided. Manual analysis was used to confirm that the code was functional and discover security issues that could be exploited. The coverage report of the provided tests as on the final day of the audit is given below.

File	% Lines	% Statements	% Branches	% Funcs
src/ActionVerifier.sol	100.00% (18/18)	100.00% (31/31)	100.00 % (12/12)	100.00% (8/8)
src/Matching.sol	100.00% (27/27)	94.44% (51/54)	83.33% (15/18)	100.00% (5/5)
src/SubAccountsManager.sol	100.00% (19/19)	100.00% (24/24)	100.00 % (8/8)	100.00% (5/5)
src/modules/BaseModule.sol	100.00% (10/10)	100.00% (18/18)	100.00 % (6/6)	100.00% (3/3)
src/modules/DepositModule.sol	100.00% (17/17)	100.00% (20/20)	75.00% (3/4)	100.00% (1/1)
src/modules/LiquidateModule.sol	100.00% (26/26)	94.59% (35/37)	66.67% (4/6)	100.00% (2/2)
src/modules/RfqModule.sol	97.73% (43/44)	96.83% (61/63)	100.00 % (12/12)	80.00% (4/5)
src/modules/TradeModule.sol	98.46% (64/65)	92.71% (89/96)	87.50% (35/40)	100.00% (10/10)
src/modules/TransferModule.sol	100.00% (22/22)	100.00% (30/30)	90.00% (9/10)	100.00% (1/1)

src/modules/WithdrawalModule.sol	100.00% (7/7)	100.00% (10/10)	100.00 % (4/4)	100.00% (1/1)
src/periphery/LyraAuctionUtils.sol	95.83% (23/24)	96.67% (29/30)	62.50% (5/8)	100.00% (5/5)
src/periphery/LyraSettlementUtils.sol	75.00% (12/16)	75.00% (24/32)	100.00 % (4/4)	80.00% (4/5)
src/periphery/SubAccountCreator.sol	100.00% (7/7)	100.00% (8/8)	50.00% (1/2)	100.00% (1/1)

3.2.3 Automated analysis

Tools were used to automatically detect the presence of several types of security vulnerabilities, including reentrancy, timestamp dependency bugs, and transaction-ordering dependency bugs. Static analysis results were reviewed manually and any false positives were removed. Any true positive results are included in this report.

Static analysis tools commonly used include Slither, Securify, and MythX. Tools such as the Remix IDE, compilation output, and linters could also be used to identify potential areas of concern.

3.3 Risk ratings

Each issue identified during the audit has been assigned a risk rating. The rating is determined based on the criteria outlined below.

- **High risk** – The issue could result in a loss of funds for the contract owner or system users.
- **Medium risk** – The issue resulted in the code specification being implemented incorrectly.
- **Low risk** – A best practice or design issue that could affect the security of the contract.
- **Informational** – A lapse in best practice or a suboptimal design pattern that has a minimal risk of affecting the security of the contract.
- **Closed** – The issue was identified during the audit and has since been satisfactorily addressed, removing the risk it posed.

4. Design specification

The following section outlines the intended functionality of the system at a high level. This specification is based on the implementation in the codebase. Any perceived points of conflict should be highlighted with the auditing team to determine the source of the discrepancy.

Lyra introduced two new modules to their v2 matching system to allow execution of trade bundles and submission of bids to liquidation auctions. The primary functions of these modules are only executable via the matching contract, which performs signature verification and validation of the signed orders submitted off-chain by traders and liquidators.

RFQ module

The module enables bundles of matched orders to be relayed and executed atomically. Orders are signed and submitted off-chain to the matching system. The module requires takers to match the order of makers explicitly.

The total cash transfer between traders is calculated using the amounts and prices specified by the maker. For perpetual asset trades, the cash transfer is based on the difference between the current perpetual asset price and the specified price, enabling traders to reference the base asset's price.

Upon executing orders, the module automatically introduces three trades: the initial cash transfer and two transactions for fee deductions from both the maker and the taker.

Liquidate module

The Liquidate module provides functionality that allows traders to perform partial liquidations of accounts atomically. A limitation of the core auction bidding mechanism is that the liquidator's account may only contain cash. Any attempt to bid with an account containing any other asset will revert.

The module creates a new sub-account for the liquidator, deposits the necessary cash, and executes the bid. This enhances the user experience and leverages the gasless relay functionality of the matching system.

Traders have the option to either retain the new account created with its asset holdings or merge these assets into their primary account.

Risk management

All asset transfers are done through the SubAccounts contract using the submitTransfers function, triggering the asset and manager hooks. This ensures that liquidator or trader accounts remain collateralized in accordance with the specific rules set by their respective risk managers.

5. Detailed findings

The following section details the findings of the audit.

5.1 High risk

No identified high-risk issues were open at the conclusion of the review.

5.2 Medium risk

No identified medium-risk issues were open at the conclusion of the review.

5.3 Low risk

No identified low-risk issues were open at the conclusion of the review.

5.4 Informational

5.4.1 Manager mismatch

LiquidateModule.sol#L55

Description

In `LiquidateModule::executeAction()` the `liquidatorAcc` is created with the same manager as the liquidatedAccountId. If the trader's subAccount has a different manager and `mergeAccount` account is specified, an unexpected revert could be encountered due to different margin requirements or supported assets.

The illustrate the issue, the unit test below was developed:

```
JavaScript
function testFullSolventAuctionMockManager() public {
    {
                              MockLiquidatableManager    mockManager    =    new
MockLiquidatableManager(address(subAccounts));
                                  camAcc    =    subAccounts.createAccount(cam,
IManager(address(mockManager)));

        _depositCash(camAcc, 1000e18);

        vm.prank(cam);
        subAccounts.setApprovalForAll(address(this), true);
```

```
        _openCLOBAccount(cam, camAcc);
    }

    _bidOnAuction(1000e18, 1e18, 0, 0, true);

    assertEq((auction.getAuction(liqAcc)).ongoing, false);
    assertGt(subAccounts.getBalance(camAcc, mockPerp, 0), 0.2e18);
              assertEq(subAccounts.getBalance(camAcc,    mockPerp,    0)    +
subAccounts.getBalance(liqAcc, mockPerp, 0), 1e18);
}
```

The unit test performs a liquidation bid with a different manager than the target account.

Recommendation

Due to the time-sensitive nature of liquidations, any unexpected reverts should be avoided. This can be achieved by only merging accounts when the managers are the same.

Update

The Lyra development team acknowledged the recommendation but decided it does not warrant additional changes to the codebase.

5.4.2 Design comments

Actions to improve the functionality and readability of the codebase are outlined below.

Named Parameters

The function prototype for `DutchAuction::bid()` contains several parameters of the same type.

Named parameters should be used to reduce the likelihood of a parameter mismatch.

Update: The Lyra development team acknowledged the recommendation but decided it does not warrant additional changes to the codebase.

State duplication

`RfqModule::isPerpAsset()` duplicates state and will require all Perps to be set again if the module is replaced (non-upgradeable).

An alternative pattern would be to either query the manager (would require adding a `isPerpAsset/assetDetails` function to the `BaseManager`) or reuse the `try/catch` for `perpPrice` pattern from `LiquidateModule`.

Update: The Lyra development team acknowledged the recommendation but decided it does not warrant additional changes to the codebase.

5.5 Closed

5.5.1 Incorrect calculation of price delta (high-risk)

RfqModule.sol#L203

Description

When selling a short position, the price and amount of the trade are specified as negative values. This results in a positive cash amount when multiplied.

However, the cash value of a `PerpAsset` is calculated as the difference between the `marketPrice` specified (negative) and the current `perpPrice`. This calculation incorrectly assumes the `marketPrice` always to be positive.

The `_getPriceDelta` function is given below for reference:

```
JavaScript
/**
* @dev Get the difference between the perp price and the traded price
* If perp price is $2000, and the limit order matched is trading at $2005,
the delta is $5
* The bidder (long) needs to pay $5 per Perp contract traded
*/
function getPerpDelta(address perpAsset, int marketPrice) internal view
returns (int delta) {
    (uint perpPrice,) = IPerpAsset(perpAsset).getPerpPrice();
    return (marketPrice - perpPrice.toInt256());
}
```

Following the example given in the function documentation, when a `Maker` is selling a Perp short position, the marketPrice would be specified as `-$2005`. This would result in `(-$2005 - $2000) = -$4005` as the price delta instead of the expected difference of $5.

The following unit test was developed to illustrate the issue:

```
JavaScript
function testFillPerpRfqOrderReverse() public {
    mockPerp.setMockPerpPrice(1500e18, 1e18);
    IRfqModule.TradeData[] memory trades = new IRfqModule.TradeData;
```

```
      trades[0] = IRfqModule.TradeData({asset: address(mockPerp), subId: 0,
price: -1490e18, amount: -10e18});

    submitRfqTrade(trades);
    assertEq(subAccounts.getBalance(camAcc, mockPerp, 0), -10e18);
    assertEq(subAccounts.getBalance(dougAcc, mockPerp, 0), 10e18);

    // cam pays 10 per perp, as the mark price is 10 under the perp price
       assertEq(subAccounts.getBalance(camAcc, cash, 0), int(cashDeposit) -
100e18);
     // dougAcc receives 10 per perp, as the mark price is 10 under the perp
price
      assertEq(subAccounts.getBalance(dougAcc, cash, 0), int(cashDeposit) +
100e18);
}
```

This test switches the direction of the trade in the `providedtestFillPerpRfqOrder` unit test, which means that the outcome of the trade should be the same, with only the parties switched. However, the test fails due to the incorrectly calculated price delta and outsized cash transfer between the parties.

Recommendation

To resolve the issue, `_getPriceDelta` should be calculated using the absolute value of `marketPrice`.

JavaScript

```
function _getPerpDelta(address perpAsset, int marketPrice) internal view
returns (int delta) {
    (uint perpPrice,) = IPerpAsset(perpAsset).getPerpPrice();
    return (int(marketPrice.abs()) - perpPrice.toInt256());
}
```

Update

The `RFQModule` was updated in commit [d8120a4] to only support positive prices for trades, removing the ambiguity of trades with negative prices.