MAY 2023

SECURITY AUDIT

Key Finance

Audited By :
HollaDieWaldfee

Audit Report - Key Finance

Audit Date 28/05/2023 - 31/05/2023
Auditor HollaDieWaldfee (@HollaWaldfee100)
Version 1 31/05/2023 Initial Report

Contents

Disclaimer

The following smart contract audit report is based on the information and code provided by the client, and any findings or recommendations are made solely on the basis of this information. While the Auditor has exercised due care and skill in conducting the audit, it cannot be guaranteed that all issues have been identified and that there are no undiscovered errors or vulnerabilities in the code.

Furthermore, this report is not an endorsement or certification of the smart contract, and the Auditor does not assume any responsibility for any losses or damages that may result from the use of the smart contracts, either in their current form or in any modified version thereof.

About HollaDieWaldfee

HollaDieWaldfee is a top ranked Smart Contract Auditor doing audits on code4rena (www.code4rena.com) and Sherlock (www.sherlock.xyz), having ranked 1st in multiple contests. On Sherlock he uses the handle “rogueereddwarf” to compete in contests. He can also be booked for conducting Private Audits.

Contact:

Twitter: @HollaWaldfee100

Scope

The audit has been conducted in the “key-for-gmx” repository which is private.

The commit hash at the start of the audit was:
2159411fe529540ad670c86b98b94fdc66c6916e

The final version of the contract has been pushed to the client’s public repository (https://github.com/KeyFinanceTeam/key-finance-contracts) at commit:
ced7bc388a37444c360a50629fe934542f0040ac

Files included in the audit:

/contracts/Market.sol

/contracts/SortedArrays.sol

/contracts/LinkedList.sol (added during the audit)

Severity Classification

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	HIGH	HIGH	MEDIUM
Likelihood: Medium	HIGH	MEDIUM	LOW
Likelihood: Low	MEDIUM	LOW	LOW

Impact - the technical, economic and reputation damage of a successful attack

Likelihood - the chance that a particular vulnerability is discovered and exploited

IMPROVEMENT: Findings in this category are recommended changes that are not related to security but can improve structure, usability and overall effectiveness of the protocol.

Summary

Severity	Total	Fixed	Acknowledged	Disputed	Reported
HIGH	0	0	0	0	0
MEDIUM	3	3	0	0	0
LOW	0	0	0	0	0
IMPROVEMENT	2	2	0	0	0

#	Title	Severity	Status
1	Cancellation of order can be front-run such that user would close wrong order	MEDIUM	FIXED
2	Orders at a given price are not processed in FIFO order	MEDIUM	FIXED
3	Constant market properties are not suited for all intended trading pairs	MEDIUM	FIXED
4	_getOrders function: restrict count to arr.length	IMPROVEMENT	FIXED
5	_executeOrder function: break out of loop if condition is false	IMPROVEMENT	FIXED

Findings

Medium Risk Findings (3)

1. Cancellation of order can be front-run such that user would close wrong order MEDIUM FIXED

Description: When an order is cancelled, the `_cancelOrder` function is called. A user needs to provide the `_price` of the order as well as the `_orderId` of the order when he calls the external `cancelBidOrder` or `cancelAskOrder` function.

The issue is that it is not guaranteed that the order that will be closed is the order with `_orderId`.

Due to other transactions being processed first, another order may end up in the position of the intended order with `_orderId`. Thereby the wrong order would be closed.

The root cause of this lies in the `_cancelOrder` function.

contracts/Market.sol#L136-L140

```
uint256 idx = orderIdToIndex[_orderId];
Order storage _order = orderMap[_price][idx];
require(_order.maker == msg.sender, 'Market: not order maker');
_cancelOrder(_order);
_removeOrder(orderMap[_price], orderIdToIndex, idx);
```

The `_orderId` is used to determine the index of the order to remove in the `orderMap[_price]` array.

It's possible that the order with `_orderId` is filled before it can be cancelled. Its index would not be deleted from `orderMap[_price]` and another order of the same `msg.sender` might now move to the same index.

Thereby `orderMap[_price][idx]` now contains the wrong order.

Impact: Due to front-running or just other transactions being processed first without evil intention, a user may close a wrong order.

Recommendation: Add a check in the `_cancelOrder` function that `_order.id == _orderId`.

Fix: In response to issue number 2, the order maps are now implemented as linked lists. As a result of the refactoring that has taken place, the issue now no longer exists. Either the intended order is cancelled or the transaction reverts.

2. Orders at a given price are not processed in FIFO order MEDIUM FIXED

Description: The orders at a given price are stored in an `Order[]` array and when an order is removed at any position in the array, the order at the last position is moved to the position that is now available:

contracts/Market.sol#L170-L174

```
Order memory _last = _orders[_orders.length - 1];
Order memory _removed = _orders[idx];
_orders[idx] = _last;
_orderIdToIndex[_last.id] = idx;
_orders.pop();
```

This means that orders are not processed in a FIFO fashion. Instead it is unpredictable in which sequence orders are processed.

Impact: The above described behavior breaks an important property of orderbook markets which is that orders that are created earlier are filled earlier.

If the Market reaches a certain amount of liquidity and trading volume it can become profitable to intentionally influence the sequence of orders to get one's order filled earlier.

This leads to a loss in the form of unfilled orders for less sophisticated users and in any case is an unreliable (i.e. unpredictable) order matching mechanism.

Recommendation: Move to a different data structure for managing orders that supports processing orders in a FIFO fashion.

Fix: The fix has been implemented over the range of multiple commits, starting at commit *266017b63dfab5315065ab154af8ac33a5112512* and ending at commit *f1b164e737a58a523d77e09af27c43896ec63ff4*.

The `bidOrderMap` and `askOrderMap` are no longer implemented as an array but as a Linked List (the "LinkedList.List" data structure has been implemented in the `contracts/LinkedList.sol` file).

3. Constant market properties are not suited for all intended trading pairs MEDIUM FIXED

Description: The `Market` is supposed to work with GMX/GMXKey, GMX/esGMXKey and GMX/MPKey trading pairs and it defines the following constants:

contracts/Market.sol#L9-L15

```
uint256 public constant TICK_TO_UNIT = 100; // tick size: 0.01
uint256 public constant FEE = 100; // 1%
uint256 public constant FEE_BASE = 10000;
uint256 public constant MAX_PRICE_COUNT = 200;
uint256 public constant MAX_ORDER_COUNT = 100;
uint256 public constant MAX_PRICE = 200; // 2.00
uint256 public constant MIN_AMOUNT = 1e18;
```

These constants are not suited for all trading pairs. They only work for the GMX/GMXKey trading pair.

Assume the trading pair is GMX/MPKey. The price of MPKey is currently ~$6 (https://gmxkey.com/) and the price of GMX is ~$55 (https://coinmarketcap.com/currencies/gmx/).

Therefore the fair price of 1 MPKey should be around ~0.1 GMX.

With the above parameters there would only be 10 prices to trade at in between zero and the fair market price (0.01, 0.02, ..., 0.10).

Clearly this does not allow for a liquid and efficient market to arise.

Impact: The constants defined in the `Market` do not support all three intended trading pairs.

Recommendation: I recommend to set the constants in the constructor of the contract such that they can be adapted to the specific pair that is traded in the `Market`. Care must be taken such as to not introduce additional vulnerabilities by setting these parameters.

Fix: The recommended fix has been implemented in commit *e6b92725a1b6d8e4502318ac6d6d50b96b4a6344*. The `tick_size` and `max_price` variables can now be configured per instance of the `Market`.

Improvement Findings (2)

4. _getOrders function: restrict count to arr.length IMPROVEMENT FIXED

Description: The `_getOrders` view function can be improved.

If the `count` parameter is greater than `arr.length`, there would be empty elements in the returned `orderBookInfo` array.

contracts/Market.sol#L196-L212

```
function _getOrders(bool _bidAsk, uint256 count) private view returns
(OrderBookInfo[] memory orderBookInfo) {
    SortedArrays.SortedArray storage priceSorted = _bidAsk ? bidPriceSorted :
askPriceSorted;
    mapping(uint256 => Order[]) storage orderMap = _bidAsk ? bidOrderMap :
askOrderMap;
    uint256[] memory arr = priceSorted.array;

    orderBookInfo = new OrderBookInfo[](count);
    for (uint256 i = 0; i < count; i++) {
        uint256 price = arr[i];
        Order[] memory _orders = orderMap[price];
        uint256 amount = 0;
        for (uint256 j = 0; j < _orders.length; j++) {
            if (!_orders[j].cancelled) amount +=
_getRemainingAmount(_orders[j]);
        }
        orderBookInfo[i] = OrderBookInfo(price, amount, _bidAsk);
        if (i == arr.length - 1) break;
    }
}
```

I recommend to set `count = arr.length` if `count > arr.length`.

Fix: The recommendation has been implemented in commit *334c292de0e4f632d8dd543b86bf33278943dc43*. Now the length of the returned `orderBookInfo` array is capped at `count`. Also the function has been renamed to `_getOrderBookInfo`.

5. _executeOrder function: break out of loop if condition is false IMPROVEMENT FIXED

Description: The `_executeOrder` function iterates over all prices in the `arr` array, which is not necessary since the prices are ordered.

contracts/Market.sol#L86-L103

```
function _executeOrder(uint256 _price, uint256 _amount, uint256 _loop, bool
_bidAsk) private {
    require(_amount >= MIN_AMOUNT, 'Market: order with too small amount');

    SortedArrays.SortedArray storage priceSorted = _bidAsk ? askPriceSorted :
bidPriceSorted;
    mapping(uint256 => Order[]) storage orderMap = _bidAsk ? askOrderMap :
bidOrderMap;
    uint256[] memory arr = priceSorted.array;

    for (uint256 i = 0; i < arr.length; i++) {
        if ((_bidAsk && arr[i] <= _price) || (!_bidAsk && arr[i] >= _price))
{
            Order[] storage _orders = orderMap[arr[i]];
            (_amount, _loop) = _matchOrders(_orders, _amount, _loop,
_bidAsk);
            if (_loop == 0) return;
            if (_amount == 0) return;
        }
    }

    _makeOrder(_price, _amount, _bidAsk);
}
```

For the "bid" side the prices are ordered descending and for the "ask" side they are ordered ascending. Therefore we know that if the `((_bidAsk && arr[i] <= _price) || (!_bidAsk && arr[i] >= _price))` condition fails we can safely break out of the for loop because there will be no other prices to match orders at.

Fix: The recommendation has been implemented in commit *45d0934c010c33425814bc650f9920139b08f85a*. Now there is an "else" case which breaks out of the loop.

```
for (uint256 i = 0; i < arr.length; i++) {
    if ((_bidAsk && arr[i] <= _price) || (!_bidAsk && arr[i] >= _price)) {
        Order[] storage _orders = orderMap[arr[i]];
        (_amount, _loop) = _matchOrders(_orders, _price, _amount, _loop,
_bidAsk);
        if (_amount == 0) return;
        if (_loop == 0) return;
    } else {
        break;
```

```
    }
}
```