SECURITY AUDIT OF

HOLDSTATION DEX

Draft Report

May 10, 2023

Verichains Lab

info@verichains.io

https://www.verichains.io

Driving Technology > Forward

ABBREVIATIONS

Name	Description
Ethereum	An open source platform based on blockchain technology to create and distribute smart contracts and decentralized applications.
Ether (ETH)	A cryptocurrency whose blockchain is generated by the Ethereum platform. Ether is used for payment of transactions and computing services in the Ethereum network.
Smart contract	A computer protocol intended to digitally facilitate, verify or enforce the negotiation or performance of a contract.
Solidity	A contract-oriented, high-level language for implementing smart contracts for the Ethereum platform.
Solc	A compiler for Solidity.
ERC20	ERC20 (BEP20 in Binance Smart Chain or *x*RP20 in other chains) tokens are blockchain-based assets that have value and can be sent and received. The primary difference with the primary coin is that instead of running on their own blockchain, ERC20 tokens are issued on a network that supports smart contracts such as Ethereum or Binance Smart Chain.
BSC	Binance Smart Chain or BSC is an innovative solution for introducing interoperability and programmability on Binance Chain.

EXECUTIVE SUMMARY

This Security Audit Report was prepared by Verichains Lab on May 10, 2023. We would like to thank the Holdstation for trusting Verichains Lab in auditing smart contracts. Delivering high-quality audits is always our top priority.

This audit focused on identifying security flaws in code and the design of the Holdstation Dex. The scope of the audit is limited to the source code files provided to Verichains. Verichains Lab completed the assessment using manual, static, and dynamic analysis techniques.

During the audit process, the audit team found some vulnerabilities in the application.

TABLE OF CONTENTS

1. MANAGEMENT SUMMARY

1.1. About Holdstation Dex

Holdstation Dex is a decentralized leveraged trading platform that combines liquidity efficiency, robustness, and user-friendliness.

1.2. Audit scope

This audit focused on identifying security flaws in code and the design of the smart contracts of Holdstation Dex.

It was conducted on commit `6fb1151dd05395a94a00bd73fb8c6095d5cd055d` from git repository link: *https://gitlab.com/hspublic/contract-holdstation-dex*.

1.3. Audit methodology

Our security audit process for smart contract includes two steps:

- Smart contract codes are scanned/tested for commonly known and more specific vulnerabilities using public and RK87, our in-house smart contract security analysis tool.
- Manual audit of the codes for security issues. The contracts are manually analyzed to look for any potential problems.

Following is the list of commonly known vulnerabilities that were considered during the audit of the smart contract:

- Integer Overflow and Underflow
- Timestamp Dependence
- Race Conditions
- Transaction-Ordering Dependence
- DoS with (Unexpected) revert
- DoS with Block Gas Limit
- Gas Usage, Gas Limit and Loops
- Redundant fallback function
- Unsafe type Inference
- Reentrancy
- Explicit visibility of functions state variables (external, internal, private and public)
- Logic Flaws

For vulnerabilities, we categorize the findings into categories as listed in table below, depending on their severity level:

SEVERITY LEVEL	DESCRIPTION
CRITICAL	A vulnerability that can disrupt the contract functioning; creates a critical risk to the contract; required to be fixed immediately.
HIGH	A vulnerability that could affect the desired outcome of executing the contract with high impact; needs to be fixed with high priority.
MEDIUM	A vulnerability that could affect the desired outcome of executing the contract with medium impact in a specific scenario; needs to be fixed.
LOW	An issue that does not have a significant impact, can be considered as less important.

Table 1. Severity levels

1.4. Disclaimer

Please note that security auditing cannot uncover all existing vulnerabilities, and even an audit in which no vulnerabilities are found is not a guarantee for a 100% secure smart contract. However, auditing allows discovering vulnerabilities that were unobserved, overlooked during development and areas where additional security measures are necessary.

2. AUDIT RESULT

2.1. Overview

The Holdstation Dex was written in `Solidity` language. The source code was written based on OpenZeppelin's library.

2.2. Findings

Note: this is just a summary of noticeable findings we discovered during the early time of the audit.

2.2.1. Incompatible function parameter HIGH

Affected files:

- HSTrading.sol
- HSTradingStorage.sol

There is an incompatible in struct `OpenLimitOrder` between `HSTrading` and `HSTradingStorage`. In `openTrade` function in `HSTrading`, it calls `storeOpenLimitOrder` function in `HSTradingStorage` contract but the passed in parameter `OpenLimitOrder` (from `StorageInterfaceV5`) is incompatible with `OpenLimitOrder` in `HSTradingStorage` so this function call will be reverted. There are some other places which are using wrong `OpenLimitOrder` struct too.

```
function openTrade(
    StorageInterfaceV5.Trade memory t,
    NftRewardsInterfaceV6.OpenLimitOrderType orderType, // LEGACY => market
    uint256 spreadReductionId,
    uint256 slippageP, // for market orders only
    address referrer
) external notContract notDone checkWhitelist {
    ...
    storageT.storeOpenLimitOrder(
        StorageInterfaceV5.OpenLimitOrder( // incompatible function parameter
            StorageInterfaceV5.OrderInfo(
                t.pairIndex,
                t.positionSizeUsdc,
                t.buy,
                t.leverage,
                t.tp,
                t.sl,
                t.openPrice,
                t.openPrice
            ),
            sender,
            index,
            spreadReductionId > 0 ? storageT.spreadReductionsP(spreadReductionId - 1) : 0,
            block.number
        )
    );
    ...
}

// StorageInterfaceV5.sol
struct OrderInfo {
    uint256 pairIndex;
    uint256 positionSize;
    bool buy;
    uint256 leverage;
    uint256 tp;
    uint256 sl;
    uint256 minPrice;
    uint256 maxPrice;
}
struct OpenLimitOrder {
    OrderInfo orderInfo;
    address trader;
    uint256 index;
    uint256 spreadReductionP;
    uint256 block;
}

// HSTradingStorage.sol
struct OpenLimitOrder {
    address trader;
    uint256 pairIndex;
    uint256 index;
    uint256 positionSize; // 1e18 (USDC or GFARM2)
    uint256 spreadReductionP;
    bool buy;
    uint256 leverage;
    uint256 tp; // PRECISION (%)
    uint256 sl; // PRECISION (%)
    uint256 minPrice; // PRECISION
    uint256 maxPrice; // PRECISION
    uint256 block;
    uint256 tokenId; // index in supportedTokens
}

function storeOpenLimitOrder(OpenLimitOrder memory o) external onlyTrading {
    o.index = firstEmptyOpenLimitIndex(o.trader, o.pairIndex);
    o.block = block.number;
    openLimitOrders.push(o);
    openLimitOrderIds[o.trader][o.pairIndex][o.index] = openLimitOrders.length - 1;
    openLimitOrdersCount[o.trader][o.pairIndex]++;
}
```

RECOMMENDATION

Fix `StorageInterfaceV5` to reflect the right struct for `HSTradingStorage` and fix the function call in `openTrade` function.

2.2.2. Test functions must be removed HIGH

Affected files:

- HSTrading.sol
- HSPriceAggregatorV1.sol

There are many places are using `aggregator.emptyNodeFulFill` to test without oracle nodes in testnet. This test function must be removed before deploying in mainnet or `gov` can control `fakeFeedPrice` to manipulate users' positions through fake price.

```
// HSTrading.sol
function openTrade(
    StorageInterfaceV5.Trade memory t,
    NftRewardsInterfaceV6.OpenLimitOrderType orderType, // LEGACY => market
    uint256 spreadReductionId,
    uint256 slippageP, // for market orders only
    address referrer
) external notContract notDone checkWhitelist {
    ...
    storageT.storePendingMarketOrder(
        StorageInterfaceV5.PendingMarketOrder(
            StorageInterfaceV5.Trade(sender, t.pairIndex, 0, 0, t.positionSizeUsdc, 0,
t.buy, t.leverage, t.tp, t.sl),
            0,
            t.openPrice,
            slippageP,
            spreadReductionId > 0 ? storageT.spreadReductionsP(spreadReductionId - 1) : 0,
            0
        ),
        orderId,
        true
    );

    aggregator.emptyNodeFulFill(t.pairIndex, orderId,
AggregatorInterfaceV6.OrderType.MARKET_OPEN); // test function must be removed.

    emit MarketOrderInitiated(orderId, sender, t.pairIndex, true);
    ...
}

// HSPriceAggregatorV1.sol
function emptyNodeFulFill(uint256 pairIndex, uint256 orderId, OrderType orderType) external
onlyTrading {
    if (nodes.length != 0) {
        return;
    }
    PairsStorageInterfaceV6.Feed memory f = pairsStorage.pairFeed(pairIndex);

    uint256 feedPrice;
    if (fakeFeedPrice == 0) {
        (, int256 feedPrice1, , , ) = ChainlinkFeedInterfaceV5(f.feed1).latestRoundData();
        if (f.feedCalculation == PairsStorageInterfaceV6.FeedCalculation.DEFAULT) {
            feedPrice = uint256((feedPrice1 * int256(PRECISION)) / 1e8);
        } else if (f.feedCalculation == PairsStorageInterfaceV6.FeedCalculation.INVERT) {
            feedPrice = uint256((int256(PRECISION) * 1e8) / feedPrice1);
        } else {
            (, int256 feedPrice2, , , ) =
ChainlinkFeedInterfaceV5(f.feed2).latestRoundData();
            feedPrice = uint256((feedPrice1 * int256(PRECISION)) / feedPrice2);
        }
    } else feedPrice = fakeFeedPrice;

    CallbacksInterfaceV6_2.AggregatorAnswer memory a;

    a.orderId = orderId;
    a.price = feedPrice;
    a.spreadP = pairsStorage.pairSpreadP(pairIndex);

    CallbacksInterfaceV6_2 c = CallbacksInterfaceV6_2(storageT.callbacks());

    if (orderType == OrderType.MARKET_OPEN) {
        c.openTradeMarketCallback(a);
    } else if (orderType == OrderType.MARKET_CLOSE) {
        c.closeTradeMarketCallback(a);
    } else if (orderType == OrderType.LIMIT_OPEN) {
        c.executeNftOpenOrderCallback(a);
    } else if (orderType == OrderType.LIMIT_CLOSE) {
        c.executeNftCloseOrderCallback(a);
    } else {
        c.updateSlCallback(a);
    }

    emit PriceReceived(bytes32(block.timestamp), orderId, msg.sender, pairIndex, feedPrice,
feedPrice, 0);
}
```

RECOMMENDATION

Remove all the test functions and parameters.

2.2.3. High gas cost with larger number of pending orders MEDIUM

Affected files:

- HSTradingStorage.sol

The contract is using an array to store pending orders and the pending order array is looped to remove pending order while using `currentPendingOrderIds.indexOf(_id)`. The more the number of pending orders are, the more gas will cost for running this function.

```
function unregisterPendingMarketOrder(uint256 _id, bool _open) external onlyTrading {
    PendingMarketOrder memory _order = reqID_pendingMarketOrder[_id];
    uint256[] storage orderIds = pendingOrderIds[_order.trade.trader];

    for (uint256 i = 0; i < orderIds.length; i++) {
        if (orderIds[i] == _id) {
            if (_open) {
                pendingMarketOpenCount[_order.trade.trader][_order.trade.pairIndex]--;
            } else {
                pendingMarketCloseCount[_order.trade.trader][_order.trade.pairIndex]--;

openTradesInfo[_order.trade.trader][_order.trade.pairIndex][_order.trade.index].beingMarket
Closed = false;
            }

            orderIds[i] = orderIds[orderIds.length - 1];
            orderIds.pop();

            delete reqID_pendingMarketOrder[_id];
            if (currentPendingOrderIds.length > 0) {
                int256 index = currentPendingOrderIds.indexOf(_id); // High gas cost with
larger number of pending orders.
                if (index >= 0) {
                    currentPendingOrderIds.remove(uint256(index));
                }
            }
            return;
        }
    }
}
```

RECOMMENDATION

Use a mapping to store index of `_id` in `currentPendingOrderIds` array and look it up instead of using `indexOf`.

3. VERSION HISTORY

Version	Date	Status/Change	Created by
1.0	*May 10, 2023*	Draft Report	Verichains Lab

Table 2. Report versions history