AstraSec

Hercules LimitOrder

Security Audit Report

August 26, 2024

Contents

1 Introduction

1.1 About Hercules

`Hercules` is a deep liquidity infrastructure project that aims to leverage `Camelot`'s successful DEX model to achieve deep and adaptable liquidity for Metis-based projects. The project boasts a suite of innovative features and liquidity strategies to foster long-term market vitals for projects while incentivizing users with real yield possibilities.

1.2 Source Code

The following source code was reviewed during the audit:

- https://github.com/HerculesDeFiLabs/limit-order-contract.git
- CommitID: 7c45ae5

And this is the final version representing all fixes implemented for the issues identified in the audit:

- https://github.com/HerculesDeFiLabs/limit-order-contract.git
- CommitID: abf46b6

2 | Overall Assessment

This report has been compiled to identify issues and vulnerabilities within the `Hercules LimitOrder` protocol. Throughout this audit, we identified a total of 4 issues spanning various severity levels. By employing auxiliary tool techniques to supplement our thorough manual code review, we have discovered the following findings.

Severity	Count	Acknowledged	Won't Do	Addressed
Critical	-	-	-	-
High	-	-	-	-
Medium	-	-	-	-
Low	3	1	-	2
Informational	1	-	-	1
Undetermined	-	-	-	-

3 Vulnerability Summary

3.1 Overview

Click on an issue to jump to it, or scroll down to see them all.

~~L-1~~ Incompatibility with Deflationary Tokens

~~L-2~~ Improved Logic of LimitOrder::createOrder()

L-3 Potential Risks Associated with Centralization

~~I-1~~ Redundant State/Code Removal

3.2 Security Level Reference

In web3 smart contract audits, vulnerabilities are typically classified into different severity levels based on the potential impact they can have on the security and functionality of the contract. Here are the definitions for critical-severity, high-severity, medium-severity, and low-severity vulnerabilities:

Severity	Description
C-X (Critical)	A severe security flaw with immediate and significant negative consequences. It poses high risks, such as unauthorized access, financial losses, or complete disruption of functionality. Requires immediate attention and remediation.
H-X (High)	Significant security issues that can lead to substantial risks. Although not as severe as critical vulnerabilities, they can still result in unauthorized access, manipulation of contract state, or financial losses. Prompt remediation is necessary.
M-X (Medium)	Moderately impactful security weaknesses that require attention and remediation. They may lead to limited unauthorized access, minor financial losses, or potential disruptions to functionality.
L-X (Low)	Minor security issues with limited impact. While they may not pose significant risks, it is still recommended to address them to maintain a robust and secure smart contract.
I-X (Informational)	Warnings and things to keep in mind when operating the protocol. No immediate action required.
U-X (Undetermined)	Identified security flaw requiring further investigation. Severity and impact need to be determined. Additional assessment and analysis are necessary.

3.3 Vulnerability Details

3.3.1 [L-1] Incompatibility with Deflationary Tokens

Target	Category	IMPACT	LIKELIHOOD	STATUS
LimitOrder.sol	Business Logic	Low	Low	Addressed

The `createOrder()` function facilitates the creation of a limit order, allowing a user to exchange a specified amount of `sellToken` for a specified amount of `buyToken`. However, the function assumes that `sellToken` strictly adheres to the ERC-20 standard, particularly in how tokens are transferred. This assumption creates a vulnerability when interacting with tokens that implement non-standard transfer mechanisms, such as "transfer-on-fee" tokens. These tokens deduct a fee during the transfer, resulting in the contract receiving fewer tokens than expected. As a result, the recorded `sellAmount` may be inaccurate, resulting in potential losses for the protocol.

LimitOrder::createOrder()

```
function createOrder(
    address buyToken,
    address sellToken,
    uint256 buyAmount,
    uint256 sellAmount,
    uint256 expiry
) public nonReentrant whenNotPaused returns(uint256) {
    ...
    // Create a new order
    Order memory order = Order({
        maker: msg.sender,
        buyToken: buyToken,
        sellToken: sellToken,
        buyAmount: buyAmount,
        sellAmount: sellAmount,
        expiry: expiry,
        createdAt: block.timestamp,
        status: Status.ACTIVE
    });

    orderIndex++;
    _activeOrderIndexes.add(orderIndex);
    _userOrderIndexesActive[msg.sender].add(orderIndex);
    orders[orderIndex] = order;

    // Transfer the sellToken from the user to this contract
    IERC20(sellToken).safeTransferFrom(
        msg.sender,
        address(this),
```

```
        sellAmount
    );

    emit OrderCreated(
        msg.sender,
        buyToken,
        sellToken,
        buyAmount,
        sellAmount,
        expiry
    );

    return orderIndex;
}
```

Remediation Implement a whitelist to restrict the `sellToken` and `buyToken` to standard ERC-20 tokens only; Instead of relying on the transfer amount specified by the user, use the actual amount of `sellToken` received by the contract inside `createOrder()`.

3.3.2 [L-2] Improved Logic of LimitOrder::createOrder()

Target	Category	IMPACT	LIKELIHOOD	STATUS
LimitOrder.sol	Business Logic	Low	Low	Addressed

In the `LimitOrder` contract, `EnumerableSet.UintSet` is used to manage the sets of `_activeOrderIndexes`, `_executedOrderIndexes`, and `_cancelledOrderIndexes`, which respectively track active, executed, and canceled limit orders. When a user creates a new limit order via the `createOrder()` function, the order's index is intended to be added to the `_activeOrderIndexes` set.

However, the current implementation does not check the return value of `_activeOrderIndexes.add(orderIndex)` (line 81) to ensure the addition was successful. If the addition fails and this failure is not handled, the order index will not be included in `_activeOrderIndexes`, yet the transaction would still succeed. This could lead to a situation where the limit order is not properly recorded in the active orders set, making it impossible to cancel or execute the order through the contract.

LimitOrder::createOrder()

```
60 function createOrder(
61     address buyToken,
62     address sellToken,
63     uint256 buyAmount,
64     uint256 sellAmount,
65     uint256 expiry
```

```
) public nonReentrant whenNotPaused returns(uint256) {
    ...
    // Create a new order
    Order memory order = Order({
        maker: msg.sender,
        buyToken: buyToken,
        sellToken: sellToken,
        buyAmount: buyAmount,
        sellAmount: sellAmount,
        expiry: expiry,
        createdAt: block.timestamp,
        status: Status.ACTIVE
    });

    orderIndex++;
    _activeOrderIndexes.add(orderIndex);
    _userOrderIndexesActive[msg.sender].add(orderIndex);
    orders[orderIndex] = order;

    ...

    return orderIndex;
}
```

Remediation Verify the return values of all `EnumerableSet.UintSet` operations. If any operation fails, the transaction should be reverted.

3.3.3 [L-3] Potential Risks Associated with Centralization

Target	Category	IMPACT	LIKELIHOOD	STATUS
LimitOrder.sol	Security	Low	Low	Acknowledged

In the `Hercules LimitOrder` protocol, the existence of a privileged account introduces centralization risks, as it holds significant control and authority over critical operations governing the protocol. In the following, we show the representative function potentially affected by the privileges associated with the privileged account.

Examples of Privileged Operations

```
function setYakRouter(
    address _yakRouter
) external onlyRole(DEFAULT_ADMIN_ROLE) {
    address oldYakRouter = address(yakRouter);
    yakRouter = IYakRouter(_yakRouter);
```

```
    emit SetYakRouter(oldYakRouter, _yakRouter);
}

function recoverERC20(
    address tokenAddress,
    uint256 tokenAmount
)
    external
    onlyRole(DEFAULT_ADMIN_ROLE) // Only the admin can call this function
{
    IERC20(tokenAddress).safeTransfer(msg.sender, tokenAmount); // Transfer tokens to the admin
}
```

Remediation To mitigate the identified issue, it is recommended to introduce multi-sig mechanism to undertake the role of the privileged account. Moreover, it is advisable to implement timelocks to govern all modifications to the privileged operations.

Response By Team This issue has been confirmed by the team. The multi-sig mechanism will be introduced to mitigate this issue.

3.3.4 [I-1] Redundant State/Code Removal

Target	Category	IMPACT	LIKELIHOOD	STATUS
LimitOrder.sol	Coding Practices	N/A	N/A	Addressed

By design, any user is allowed to execute an active limit order via the `executeOrder()` function. At the beginning of the function, there is a check to verify whether the provided order index corresponds to an active order (line 179). However, after thoroughly reviewing the code, it was observed that the same check is redundantly performed within the `checkOrderExecutable()` function (line 160), which is subsequently called within the `executeOrder()` function (line 188).

LimitOrder::checkOrderExecutable()&&executeOrder()

```
157 function checkOrderExecutable(
158     uint256 index
159 ) public view returns (bool, IYakRouter.FormattedOffer memory) {
160     if (!_activeOrderIndexes.contains(index)) {
161         revert IndexDoesNotExist(index);
162     }

164     Order memory order = orders[index];
```

```
    IYakRouter.FormattedOffer memory offer;
    if (block.timestamp > order.expiry) {
        return (false, offer);
    }

    ...
}

/**
 * @dev Execute an order if it is executable.
 * @param index The index of the order to be executed.
 */
function executeOrder(uint256 index) external nonReentrant whenNotPaused {
    if (!_activeOrderIndexes.contains(index)) {
        revert IndexDoesNotExist(index);
    }

    Order memory order = orders[index];

    (
        bool executable,
        IYakRouter.FormattedOffer memory offer
    ) = checkOrderExecutable(index);

    if (!executable) {
        revert NotExecutable();
    }

    ...
}
```

Moreover, we identified that the `getOrdersBatch()` function is redundant, as it duplicates the functionality of the existing `getOrders()` function. To improve code maintainability and reduce the risk of future inconsistencies, we recommend safely removing the `getOrdersBatch()` function.

LimitOrder::getOrders()&&getOrdersBatch()

```
function getOrders(
    uint256[] memory indexes
) public view returns (Order[] memory) {
    Order[] memory _orders = new Order[](indexes.length); // Create an array to
        hold the orders
    for (uint256 i = 0; i < indexes.length; i++) {
        _orders[i] = orders[indexes[i]]; // Retrieve each order by index
    }
    return _orders; // Return the array of orders
}
```

```
function getOrdersBatch(
    uint256[] memory indexes
) external view returns (Order[] memory) {
    Order[] memory _orders = new Order[](indexes.length); // Create an array to
        hold the orders
    for (uint256 i = 0; i < indexes.length; i++) {
        _orders[i] = orders[indexes[i]]; // Retrieve each order by index
    }
    return _orders; // Return the batch of active orders
}
```

Remediation Safely remove the redundant code to enhance the contract's efficiency and maintainability.

4 Appendix

4.1 About AstraSec

`AstraSec` is a blockchain security company that serves to provide high-quality auditing services for blockchain-based protocols. With a team of blockchain specialists, `AstraSec` maintains a strong commitment to excellence and client satisfaction. The audit team members have extensive audit experience for various famous DeFi projects. `AstraSec`'s comprehensive approach and deep blockchain understanding make it a trusted partner for the clients.

4.2 Disclaimer

The information provided in this audit report is for reference only and does not constitute any legal, financial, or investment advice. Any views, suggestions, or conclusions in the audit report are based on the limited information and conditions obtained during the audit process and may be subject to unknown risks and uncertainties. While we make every effort to ensure the accuracy and completeness of the audit report, we are not responsible for any errors or omissions in the report.

We recommend users to carefully consider the information in the audit report based on their own independent judgment and professional advice before making any decisions. We are not responsible for the consequences of the use of the audit report, including but not limited to any losses or damages resulting from reliance on the audit report.

This audit report is for reference only and should not be considered a substitute for legal documents or contracts.

4.3 Contact

Phone	+86 156 0639 2692
Email	contact@astrasec.ai
Twitter	https://twitter.com/AstraSecAI