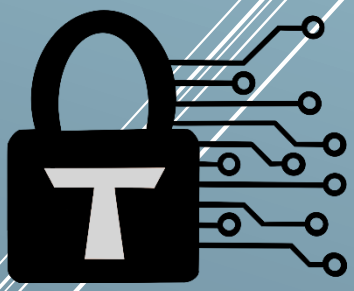
Trust Security

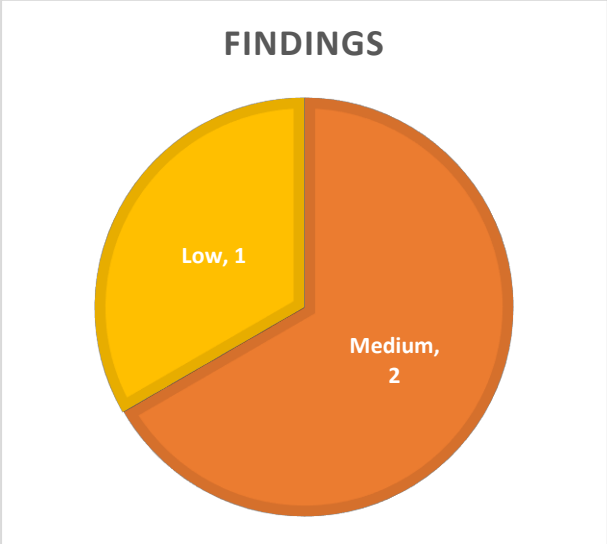


Smart Contract Audit

Clober Rebalancer

31/12/24

Executive summary

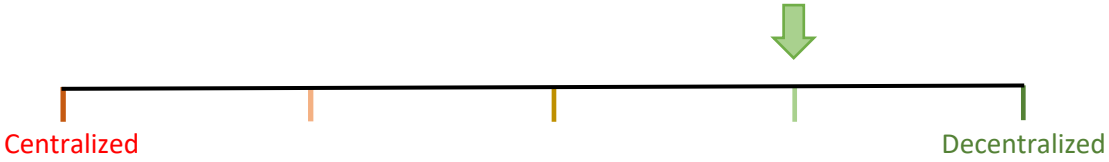


Category	Liquidity Pool
Audited file count	2
Lines of Code	615
Auditor	cccz carrotsmuggler
Time period	23/12/2024-29/12/2024

Findings

Severity	Total	Fixed
High	-	-
Medium	2	2
Low	1	1

Centralization score



Signature

Digitally signed by Trust
Date: 2024.12.31 17:39:36 +02'00'

Document properties

Versioning

Version	Date	Description
0.1	29/12/2024	Client report
0.2	31/12/2024	Mitigation review

Contact

Trust

trust@trust-security.xyz

Introduction

Trust Security has conducted an audit at the customer's request. The audit is focused on uncovering security issues and additional bugs contained in the code defined in scope. Some additional recommendations have also been given when appropriate.

Scope

The following files are in scope of the audit:

- src/Rebalancer.sol
- src/SimpleOracleStrategy.sol

Repository details

- **Repository URL:** https://github.com/clober-dex/clober-rebalancer
- **Commit hash:** b5510b36c4abdad6fdb811dd5d8ba949386514ea
- **Mitigation review hash:** 56729cdae08cd0fa813c691ccf369f45e6af869e

About Trust Security

Trust Security has been established by top-end blockchain security researcher Trust, in order to provide high quality auditing services. Trust is the leading auditor at competitive auditing service Code4rena, reported several critical issues to Immunefi bug bounty platform and is currently a Code4rena judge.

About the Auditors

A top competitor in audit contests, cccz has achieved superstar status in the security space. He is a Black Hat / DEFCON speaker with rich experience in both traditional and blockchain security.

Carrotsmuggler competes in public audit contests on various platforms with multiple Top 3 finishes. He has experience reviewing contracts on diverse EVM and non-EVM platforms.

Disclaimer

Smart contracts are an experimental technology with many known and unknown risks. Trust Security assumes no responsibility for any misbehavior, bugs or exploits affecting the audited code or any part of the deployment phase.

Furthermore, it is known to all parties that changes to the audited code, including fixes of issues highlighted in this report, may introduce new issues and require further auditing.

Methodology

In general, the primary methodology used is manual auditing. The entire in-scope code has been deeply looked at and considered from different adversarial perspectives. Any additional dependencies on external code have also been reviewed. Fuzz tests and unit tests have also been used as needed.

Qualitative analysis

Metric	Rating	Comments
Code complexity	**Good**	Project kept code as simple as possible, despite implementing custom data structures for efficiency.
Documentation	**Moderate**	Project is still under active development and currently lacks documentation.
Best practices	**Good**	Project generally follows best practices.
Centralization risks	**Good**	Project does not introduce significant unnecessary centralization risks.

Findings

Medium severity findings

TRST-M-1 Rebalance may not always work even if rebalanceThreshold is 100%

- **Category:** Logical issues
- **Source:** SimpleOracleStrategy.sol
- **Status:** Fixed

Description

In *SimpleOracleStrategy*, rebalance is not allowed when cancelable (unfilled) orders are greater than **rebalanceThreshold * lastRawAmounts**.

```
            if (
                lastRawAmounts > 0
                    && (
                        liquidityA.cancelable
                            > (lastRawAmounts >> 128) * bookKeyA.unitSize *
config.rebalanceThreshold / RATE_PRECISION
                            || liquidityB.cancelable
                                > (lastRawAmounts & LAST_RAW_AMOUNT_MASK) *
bookKeyB.unitSize * config.rebalanceThreshold
                                    / RATE_PRECISION
                    )
            ) {
                return (ordersA, ordersB);
            }
```

The **lastRawAmounts** only includes the **rawAmount** of orders.

```
    function rebalanceHook(address, bytes32 key, Order[] memory liquidityA, Order[]
memory liquidityB) external {
        if (msg.sender != address(rebalancer)) revert InvalidAccess();
        uint256 lastRawAmountA;
        uint256 lastRawAmountB;
        for (uint256 i = 0; i < liquidityA.length; ++i) {
            IStrategy.Order memory order = liquidityA[i];
            lastRawAmountA += order.rawAmount;
        }

        for (uint256 i = 0; i < liquidityB.length; ++i) {
            IStrategy.Order memory order = liquidityB[i];
            lastRawAmountB += order.rawAmount;
        }
        _lastRawAmounts[key] = (lastRawAmountA << 128) + lastRawAmountB;
    }
```

However, the cancelable amount may include **makerFee**.

```
    function _getLiquidity(FeePolicy makerPolicy, uint64 unitSize, OrderId orderId)
        internal
        view
        returns (uint256 cancelable, uint256 claimable)
    {
        IBookManager.OrderInfo memory orderInfo = bookManager.getOrder(orderId);
        cancelable = uint256(orderInfo.open) * unitSize;
```

```
        claimable = orderId.getTick().quoteToBase(uint256(orderInfo.claimable) *
unitSize, false);
        if (makerPolicy.usesQuote()) {
            int256 fee = makerPolicy.calculateFee(cancelable, true);
            cancelable = uint256(int256(cancelable) + fee);
```

Consider Base token is ETH, Quote token is USDC, ETH/USDC = 3000, and *UsesQuote()* is true, **makerFee** is 10%, **rebalanceThreshold** is 20%.

Rebalancer offers 3300 USDC (300 USDC of makerFee) to trade 1 ETH. **lastRawAmounts** will be 3000 USDC, but cancelable amount will be 3300 USDC.

When 2400 USDC (80%) is filled and 600 USDC (20%) is left cancelable, the cancelable amount will be 600 * 1.1 = 660 USDC instead of the expected 600 USDC, which results in a larger **rebalanceThreshold** than expected.

Since the max **rebalanceThreshold** is 100%, and the fact that the **rebalanceThreshold** may be 100% means that the rebalance can be performed at any time. But this would lead to even if the **rebalanceThreshold** is 100%, the rebalance can only be performed after the order has been partially fulfilled.

Recommended mitigation

It is recommended to return **rawCancelable** without **makerFee** in *getLiquidity()* and use it to calculate **rebalanceThreshold**.

Team response

Fixed.

Mitigation Review

The fix changes **lastRawAmount** to **lastAmount** and stores the amount that includes fees in **lastAmount**, which solves the issue.

TRST-M-2 Incorrect price check in *updatePosition()* may lead to protocol loss

- **Category:** Logical issues
- **Source:** SimpleOracleStrategy.sol
- **Status:** Fixed

Description

Operator will call *updatePosition()* to update the trading price and it will check that the trading price is set to make the protocol profitable.

For example, the protocol will be set to buy ETH for 3000 USDC and sell ETH for 3100 USDC, and the protocol will profit from the difference.

Considering that the **makerFee** is charged or compensated when making orders, the **priceWithFee**, i.e., the price including the **makerFee**, is checked.

But the problem here is that **priceWithFee** is calculated incorrectly - it doesn't consider *usesQuote()* and the direction is incorrect, which may result in incorrect prices being set, thus making the protocol lose on the trade.

For example, consider Base token is ETH, Quote token is USDC, ETH/USDC = 3000, and **makerFee** is 10%.

When *bookA.usesQuote()* is true, the maker offers 3300 USDC to trade 1 ETH.

When *bookA.usesQuote()* is false, the maker offers 3000 USDC to trade 0.9 ETH.

The value of the **makerFee** is the same, 300 USDC, but the **priceWithFee** is different. The former is 3300/1 = 3300 USDC and the latter is 3000/0.9 = 3333 USDC. But the protocol incorrectly calculates **priceWithFee** as 3000 *(1-10%) = 2700 for both cases.

The correct calculation should be as follows:

bookA.usesQuote() is true, **priceWithFeeA = priceA * (1+makerFee)**

bookA.usesQuote() is false, **priceWithFeeA = priceA / (1-makerFee)**

bookB.usesQuote() is true, **priceWithFeeB = priceB / (1+makerFee)**

bookB.usesQuote() is false, **priceWithFeeB = priceB * (1-makerFee)**

Recommended mitigation

It is recommended to change as follows.

```
-       uint256 priceWithFee = uint256(int256(priceA) -
bookKeyA.makerPolicy.calculateFee(priceA, false));
+       uint256 priceWithFeeA = bookKeyA.makerPolicy.usesQuote() ?
uint256(int256(priceA) + bookKeyA.makerPolicy.calculateFee(priceA, false)) :
bookKeyA.makerPolicy.calculateOriginalAmount(priceA, true);
-       priceWithFee = uint256(
-           int256(priceWithFee) -
bookManager.getBookKey(bookIdB).makerPolicy.calculateFee(priceWithFee, false)
-       );
+       IBookManager.BookKey memory bookKeyB = bookManager.getBookKey(bookIdB);
+       uint256 priceWithFeeB = bookKeyB.makerPolicy.usesQuote() ?
bookKeyB.makerPolicy.calculateOriginalAmount(priceB, false) : uint256(int256(priceB) -
bookKeyB.makerPolicy.calculateFee(priceB, false);
-       if (priceWithFee >= priceB) revert InvalidPrice();
+       if (priceWithFeeA >= priceWithFeeB) revert InvalidPrice();
```

Team response

Fixed.

Mitigation Review

The fix implements the recommendation.

Low severity findings

TRST-L-1 Orders can be canceled intentionally

- **Category:** Logical issues
- **Source:** Rebalancer.sol
- **Status:** Fixed

Description

When users withdraw funds, orders are canceled in proportion to the shares burned, this allows malicious users to cancel orders by immediately depositing and withdrawing.

Consider the following scenario:

1. Alice deposits 1000 USDT and 1000 USDC and gets 1000 shares.

2. *rebalance()* is called and 500 USDT and 500 USDC are used to make orders (50% utilization).

3. Bob deposits 4000 USDT and 4000 USDC, gets 4000 shares, and takes them out immediately.

4. In *_burn()*, 400 USDC and 400 USDT orders will be cancelled.

In the example above, when Bob withdraws, only 100 USDT and 100 USDC are in orders, and if the **rebalanceThreshold** is 10%, the new *rebalance()* call is allowed only after 90 USDT and 90 USDC have been traded, which may cause the protocol to miss some trading opportunities.

Recommended mitigation

Consider adding lockout periods/deposit withdrawal fees and other measures to prevent users from intentionally canceling orders.

Team response

Fixed.

Mitigation Review

The fix adds **burnFee** to increase the cost of intentional order cancellations by malicious users.

Additional recommendations

TRST-R-1 Users will lose on withdrawal depending on unitSize precision

When users withdraw funds, orders are canceled in proportion to the shares burned. The amount of canceled orders is rounded down to intentionally return less assets.

The problem is that orders use **unitSize** as the minimal unit, so rounding down may cause the withdrawer to lose up to a **unitSize** of assets.

```
            if (orderInfo.open > 0) {
                canceledAmount += bookManager.cancel(
                    IBookManager.CancelParams({
                        id: orderId,
                        toUnit: (orderInfo.open - orderInfo.open * cancelNumerator /
cancelDenominator).toUint64()
                    }),
                    ""
                );
            }
```

Consider that the Quote token is ETH, the amount of ETH that can be canceled in orders is 1e18, **unitSize** is 1e12, and **totalSupply** is 10001. When the user withdraws 120 shares, the protocol cancels 1e18 / 1e12 * 120 / 10001 = 11998 (11998.8 rounded down to 11998), 11998 * 1e12 = 0.011998 ETH to the user, due to rounding, the user loses 0.0000008 ETH (0.003 USD).

Since this would potentially affect every withdrawal, and the **unitSize** value in USD is not capped, there is risk for users to lose significant value in the platform due to rounding, especially over time.

One option is to calculate the assets amount to be taken out based on all liquidity and send them directly to the user, not based on **canceledAmount**.

Another option is to document the following assumptions:

- Up to **unitSize** of value can be lost on withdrawal.
- The **unitSize** of any future market should be set to a negligible amount to avoid loss of value.

TRST-R-2 Add tests for oracle with 18 decimals

MockOracle has 8 decimals, however *DatastreamOracle* with 18 decimals is already being used on-chain. It is recommended to update tests to apply *MockOracle* with 18 decimals.

```
contract MockOracle is IOracle {
    mapping(address => uint256) private _priceMap;

    bool public isValid = true;

    function decimals() external pure returns (uint8) {
        return 8;
```

```
    }
```

Centralization risks

TRST-CR-1 The owner of SimpleOracleStrategy can make orders at a very low price

In *SimpleOracleStrategy*, owner can set a large **referenceThreshold**, then malicious operator can set a very low price to sell depositors' assets.

Systemic risks

TRST-SR-1 The rebalanceThreshold prevents new deposits from making orders

The protocol only allows *rebalance()* to be called to create new orders once the current order has been filled over **rebalanceThreshold**. It requires BOTH order A and B to be filled, if order A is completely filled but order B is untouched, rebalance still doesn't happen.

This may prevent new deposits from creating orders.