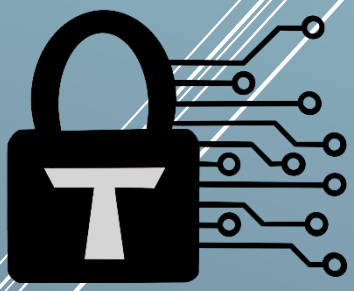
Trust Security



Smart Contract Audit

Clober Rebalancer

26/07/24

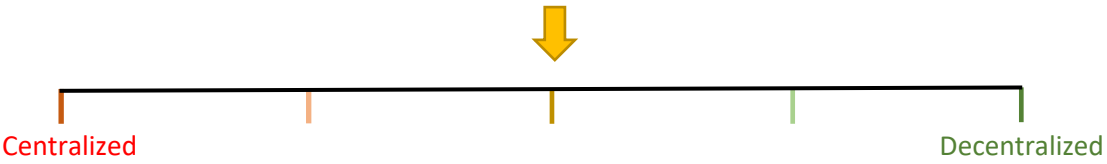
Executive summary

Category	Liquidity Pool
Audited file count	3
Lines of Code	456
Auditor	cccz carrotsmuggler
Time period	15/07/2024-26/07/2024

Findings

Severity	Total	Fixed	Acknowledged
High	2	2	-
Medium	1	1	-
Low	3	3	-

Centralization score



Signature

Digitally signed by Trust
Date: 2024.07.26 12:38:49 +03'00'

Document properties

Versioning

Version	Date	Description
0.1	23/07/2024	Client report
0.2	25/07/2024	Mitigation review
0.3	26/07/2024	Final 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
- src/libraries/ERC6909Supply.sol

Repository details

- **Repository URL:** https://github.com/clober-dex/clober-rebalancer
- **Commit hash:** 8a3a6952aa4e4296a9e12a9a9b4f20e290bbbe8a
- **Mitigation review hash:** a73dc2245c1a85d16fea5a75ae18b4cd6cb2eb2e
- **Final mitigation review hash:** 1d37d761621561239ed9ee079167425683f4f56e

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	**Moderate**	Project relies on admin to set correct parameters. A compromised admin account could risk the safety of funds.

Findings

High severity findings

TRST-H-1 Depositors may lose deposited native tokens

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

Description

When the user calls the *mint()* function to deposit, the contract transfers the user's tokens to the pool based on the pool's liquidity ratio, and mints shares to the user. The pool liquidity ratio changes as orders are filled.

When the user provides native tokens to mint shares, the contract uses the **refund** variable to calculate the native tokens to be refunded but does not return them to the user. Given the frequent changes in the liquidity ratio, it is likely that a user will provide too many native tokens, and these excess native tokens will not be refunded.

Consider that currently **liquidityA** = 3 ETH, **liquidityB** = 9000 USDC, the user offers 1 ETH and 3000 USDC to mint shares, and due to the orders being filled, the liquidity changes and **liquidityA** = 2 ETH, **liquidityB** = 12000 USDC. Then, 0.5 ETH and 3000 USDC will be used to mint shares, but the excess 0.5 ETH will not be refunded, resulting in losses for the user.

Recommended mitigation

It is recommended to refund the excess native tokens to the user in the *mint()* function.

Team response

Fixed.

Mitigation Review

The fix implements the recommendation.

TRST-H-2 Funds can be drained with native or ERC777 tokens

- **Category:** Reentrancy attacks
- **Source:** Rebalancer.sol
- **Status:** Fixed

Description

When the user withdraws tokens from the contract using the *burn()* function, the contract transfers out tokens one at a time. If these tokens have transfer hooks like in the case of ERC777 tokens or call the receiving contract functions like in the case of native token transfers, the rebalancer contract can be re-entered. The issue is that the Rebalancer contract burns the

LP tokens first, does the token transfer next and finally updates the pool reserves, allowing a reentry before all values have been updated.

```
    if (burnAmount > 0) {
            //…
            _burn(user, uint256(key), burnAmount);
            //…
        }
        //…
        if (withdrawalA > 0) {
            bookKeyA.quote.transfer(user, withdrawalA);
            pool.reserveA -= withdrawalA;
        }
        if (withdrawalB > 0) {
            bookKeyA.base.transfer(user, withdrawalB);
            pool.reserveB -= withdrawalB;
        }
```

When the code hits the *quote.transfer()* call, the contract is in a state where the LP tokens have been burnt up but the reserves have not been changed. If users re-enter the *burn()* function at this point, they can withdraw more tokens than they deserve, enabling them to drain the contract.

Consider the current reserves are 1e18 ETH and 4000e6 USDC and the total supply is 1e18 LP tokens. Say the user calls *burn()* with 0.25e18 LP tokens. **withdrawalA = 0.25e18 ETH** and **withdrawalB = 1000e6 USDC**. When the *transfer()* function is called, the receiver is allowed a chance to re-enter and call *burn()* again with 0.25e18 LP tokens. This time around, for the 0.25e6 LP tokens being burnt, **withdrawalA** = 1e18 * 0.25e18 / 0.75e18 = 0.33e18 ETH. **withdrawalB** = 4000e6 * 0.25e18 / 0.75e18 = 1333e6 USDC. This is because the **totalSupply** has already reduced to 0.75e18, but the reserves has not yet reduced to 0.75e18 and is still at 1e18, resulting in losses for all other users in the pool and allowing the draining of the contract.

Recommended mitigation

It is recommended to add non-reentrant modifiers on the user facing functions to stop users from reentering the contract. Also, CEI (checks-effects-interactions) pattern should be followed in places where callbacks are possible. Consider moving both the reserve value updates to be done before any external tokens are interacted with.

Team response

Fixed.

Mitigation Review

The fix implements the recommendation.

Medium severity findings

TRST-M-1 Tokens offered in the order may be greater than the balance, thus reverting the rebalance process

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

Description

SimpleOracleStrategy.computeOrders() calculates tokens to be offered in the order.

The problem is that when using quote tokens as fee, the rounding up in *calculateOriginalAmount()* will result in the calculated token amount being greater than the token amount in the contract, causing the rebalance process to revert due to insufficient tokens.

Consider **makerFee** is positive, 10%, **unitSize** = 1. In *computeOrders()*, **amountA** = 120, *calculateOriginalAmount()* will round up, **rawAmount** = 120 / 1.1 = 110.

```
    function calculateOriginalAmount(FeePolicy self, uint256 amount, bool reverseFee)
        internal
        pure
        returns (uint256 originalAmount)
    {
        int24 r = rate(self);

        bool positive = r > 0;
        uint256 divider;
        assembly {
            if reverseFee { r := sub(0, r) }
            divider := add(RATE_PRECISION, r)
        }
        originalAmount = Math.divide(amount * RATE_PRECISION, divider, positive);
    }
```

In *bookManager.make()*, **quoteAmount** = 110, in *calculateFee()*, **absFee** = 110 * 0.1 = 11, **quoteDelta** = 110 + 11 = 121.

```
    function calculateFee(FeePolicy self, uint256 amount, bool reverseRounding)
internal pure returns (int256 fee) {
        int24 r = rate(self);

        bool positive = r > 0;
        uint256 absRate;
        unchecked {
            absRate = uint256(uint24(positive ? r : -r));
        }
        // @dev absFee must be less than type(int256).max
        uint256 absFee = Math.divide(amount * absRate, RATE_PRECISION, reverseRounding
? !positive : positive);
        fee = positive ? int256(absFee) : -int256(absFee);
    }
```

That is, *rebalancer* needs to provide 121 tokens A. If rateA is 100%, then there will be no more tokens A in the contract.

Recommended mitigation

It is recommended to implement the new *calculateOriginalAmount()* function and always round down when calculating.

Team response

Fixed.

Mitigation Review

The fix implements the recommendation.

Low severity findings

TRST-L-1 Users cannot withdraw funds when strategy is address(0)

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

Description

When the user calls *burn()* to make a withdrawal, in *_burnAndRebalance()*, if the **strategy** is **address(0)**, the transaction will be revert and the user will not be able to make a withdrawal. However, when the **strategy** is address(0), the user can continue to make deposits.

```
    function _burnAndRebalance(bytes32 key, address user, uint256 burnAmount)
        public
        selfOnly
        returns (uint256 withdrawalA, uint256 withdrawalB)
    {
        Pool storage pool = _pools[key];
        if (pool.strategy == IStrategy(address(0))) revert InvalidBookPair();
```

Recommended mitigation

It is recommended to skip *computeOrders()* instead of reverting if **strategy** is **address(0)** when withdrawing.

Team response

Fixed.

Mitigation Review

The fix implements the recommendation.

TRST-L-2 Missing slippage in mint() can lead to unexpected token transfers

- **Category:** Slippage control issues
- **Source:** Rebalancer.sol
- **Status:** Fixed

Description

The *mint()* and *burn()* functions lack slippage checks, leading to potential discrepancies in token and LP token amounts for users, depending on the current pool state. When using *mint()*, users specify **amountA** and **amountB** as the maximum tokens they intend to mint with. However, the actual amount of LP tokens minted depends on the fraction of the user's deposit relative to the total liquidity in the pool. If the pool composition changes, users may receive fewer LP tokens than anticipated.

Say the pool has 1000e6 USDC and 1000e6 USDT tokens, some of which are being used for current orders and **totalSupply** of the LP token is also 1000e6. User wants to add liquidity with 1000e6 USDC and 1000e6 USDT as well. If an order gets matched and the pool composition

changes now to 1500e6 USDC and 500e6 USDT, then only a part of the user's deposit will be used to mint LP. Depositor will now be charged only 1000e6 USDC and 333.33e6 USDT tokens and be minted only 666.67e6 LP tokens instead of the expected 1000e6 LP tokens.

When using *burn()*, if the pool composition changes, ratio of tokens withdrawn will be inconsistent with expectations. For example, a user burns shares and hopes to withdraw 1 ETH and 3000 USDC but may withdraw 1.5 ETH and 1500 USDC due to the pool composition changes.

Recommended mitigation

Consider allowing the user to pass in a minimum mint amount, to make sure they mint a minimum amount of LP tokens and lock in the token ratios when providing liquidity. Also, allow users to pass in the desired tokens amount when burning to make sure the tokens received are consistent with expectations.

Team response

Fixed.

Mitigation Review

The fix implements the recommendation for *mint()*, but not for *burn()*. It is recommended to allow users to pass in the desired tokens amount when burning to make sure the tokens received are consistent with expectations.

Team response

Fixed.

Mitigation Review

The fix implements the recommendation for *burn()*.

TRST-L-3 _open() may overwrite existing pools

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

Description

When owner calls *open()* to create a pool, it does not check if the pool already existed. A harmful case is that if the owner accidentally creates a pool using **bookKeyB**/**bookKeyA** in reverse order, the existing **bookKeyA**/**bookKeyB** pool will be overwritten because they have the same key, which will replace the existing pool's **bookIdA**/**bookIdB** with **bookIdB**/**bookIdA**. This will cause incorrect bookIDs to be used when making orders.

```
    function _open(IBookManager.BookKey calldata bookKeyA, IBookManager.BookKey
calldata bookKeyB, address strategy)
        public
        selfOnly
        returns (bytes32 key)
    {
        if (!(bookKeyA.quote.equals(bookKeyB.base) &&
bookKeyA.base.equals(bookKeyB.quote))) revert InvalidBookPair();
```

```
        if (address(bookKeyA.hooks) != address(0) || address(bookKeyB.hooks) !=
address(0)) revert InvalidHook();

        BookId bookIdA = bookKeyA.toId();
        BookId bookIdB = bookKeyB.toId();
        if (!bookManager.isOpened(bookIdA)) bookManager.open(bookKeyA, "");
        if (!bookManager.isOpened(bookIdB)) bookManager.open(bookKeyB, "");

        key = _encodeKey(bookIdA, bookIdB);
        _pools[key].bookIdA = bookIdA;
        _pools[key].bookIdB = bookIdB;
```

Recommended mitigation

It is recommended to check if the pool already exists in *_open()*.

Team response

Fixed.

Mitigation Review

If the owner calls setStrategy() to set _pool.strategy to address(0), then the pool may still be overwritten. It is recommended to check if _pools[key].bookIdA or _pools[key].bookIdB is non zero

Team response

Acknowledged, the *setStrategy()* function will only exist in the testnet. It will be removed before deploying to the mainnet.

Additional recommendations

TRST-R-1 The first depositor can control the liquidity ratio

For example, for the USDC/USDT pool, the first depositor can provide x wei USDC and 1 wei USDT, in which x * **minRateA** +1 = **bookKeyA.unitSize**, thus ensuring that **rawAmount** = 0 when making orders. This will prevent rebalance from making orders and the pool will always remain USDC/USDT = x:1.

In this way, the USDC: USDT provided by the subsequent depositors must be x:1, which may make the pool insufficiently liquid for market making.

It is recommended that the pool creator make the initial deposit to control the ratio, or require that the liquidity value on both sides of the first deposit is not much different.

TRST-R-2 The bookPair mapping can get overwritten

The contract uses a mapping **bookPair** to store the addresses of the pairs of orderbooks which make up a rebalancing pool, so **bookPair[bookA] = bookB** and **bookPair[bookB] = bookA.** However, in the way this is implemented in the *_open()* function, this mapping can easily get overwritten if any book is used to create multiple book pairs. So, if bookA and bookB are used to create a pair, and then bookA and bookC is used to create a pair, the **bookPair[bookA]** will be set to bookC only. The bookA-bookB pair rebalancing mechanism will still work, however this mapping will only show bookC as the pair for bookA.

It is recommended to not use same books in different pairs. Otherwise, the **bookPair** mapping cannot be relied upon to always give the complete list of opened pairs in the contract.

Centralization risks

TRST-CR-1 The owner of the Rebalancer can set any strategy

In *Rebalancer*, owner can set any **strategy**, malicious strategies may use low prices to sell depositors' assets.

TRST-CR-2 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 Using fixed ticks to make orders can be arbitraged

Anyone can perform a rebalance to make orders at a fixed tick, and the deviation of the fixed tick from the market price can be used for arbitrage. It's not safe for the protocol to rely on operator price feeds, which may have the risk of delayed price feeds and frontrunning.

TRST-SR-2 Rebalance will cancel old orders and make new orders

When the prices of the orders are the same, the new order will be further down in the take queue, i.e., it will be taken later, so malicious users can exploit this to get their orders taken earlier.

TRST-SR-3 Total MakerFee greater than 0 will lose depositors

If makerFee is greater than 0, both bookA's quote-to-base and bookB's base-to-quote will be charged. When rebalance makes orders and orders are taken, the total assets are reduced.

For example, if makerFee is 0.1%, the user deposits 10,000 USD tokens A and 10,000 USD tokens B, rebalance make the order, the order is taken, 10,000 USD tokens A go to 9,990 USD tokens B, and 10,000 USD tokens B go to 9,900 USD tokens A.

A malicious user could call rebalance frequently to compromise depositors.